

Programming Languages

CONCEPTS AND IMPLEMENTATION

Saverio Perugini



Programming Languages

CONCEPTS AND IMPLEMENTATION

Saverio Perugini

Department of Computer Science
University of Dayton



JONES & BARTLETT
LEARNING



World Headquarters

Jones & Bartlett Learning
25 Mall Road, Suite 600
Burlington, MA 01803
978-443-5000
info@jblearning.com
www.jblearning.com

Jones & Bartlett Learning books and products are available through most bookstores and online booksellers. To contact Jones & Bartlett Learning directly, call 800-832-0034, fax 978-443-8000, or visit our website, www.jblearning.com.

Substantial discounts on bulk quantities of Jones & Bartlett Learning publications are available to corporations, professional associations, and other qualified organizations. For details and specific discount information, contact the special sales department at Jones & Bartlett Learning via the above contact information or send an email to specialsales@jblearning.com.

Copyright © 2023 by Jones & Bartlett Learning, LLC, an Ascend Learning Company

All rights reserved. No part of the material protected by this copyright may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

The content, statements, views, and opinions herein are the sole expression of the respective authors and not that of Jones & Bartlett Learning, LLC. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not constitute or imply its endorsement or recommendation by Jones & Bartlett Learning, LLC and such reference shall not be used for advertising or product endorsement purposes. All trademarks displayed are the trademarks of the parties noted herein. *Programming Languages: Concepts and Implementation* is an independent publication and has not been authorized, sponsored, or otherwise approved by the owners of the trademarks or service marks referenced in this product.

There may be images in this book that feature models; these models do not necessarily endorse, represent, or participate in the activities represented in the images. Any screenshots in this product are for educational and instructive purposes only. Any individuals and scenarios featured in the case studies throughout this product may be real or fictitious but are used for instructional purposes only.

23862-4

Production Credits

VP, Content Strategy and Implementation: Christine Emerton	Product Fulfillment Manager: Wendy Kilborn
Product Manager: Ned Hinman	Composition: S4Carlisle Publishing Services
Content Strategist: Melissa Duffy	Cover Design: Michael O'Donnell
Project Manager: Jessica deMartin	Media Development Editor: Faith Brosnan
Senior Project Specialist: Jennifer Risden	Rights Specialist: James Fortney
Digital Project Specialist: Rachel DiMaggio	Cover Image: © javarman/Shutterstock.
Marketing Manager: Suzy Balk	Printing and Binding: McNaughton & Gunn

Library of Congress Cataloging-in-Publication Data

Names: Perugini, Saverio, author.

Title: Programming languages : concepts and implementation / Saverio Perugini, Department of Computer Science, University of Dayton.

Description: First edition. | Burlington, MA : Jones & Bartlett Learning, [2023] | Includes bibliographical references and index.

Identifiers: LCCN 2021022692 | ISBN 9781284222722 (paperback)

Subjects: LCSH: Computer programming. | Programming languages (Electronic computers)

Classification: LCC QA76.6 .P47235 2023 | DDC 005.13-dc23

LC record available at <https://lcn.loc.gov/2021022692>

6048

Printed in the United States of America

25 24 23 22 21 10 9 8 7 6 5 4 3 2 1

✠
✠ JMJ ✠

Ad majorem Dei gloriam.

Omnia in Christo.

Sancte Ioseph, ExémpLAR opíficUM, Ora pro nobis.

Sancte Thoma de Aquino, Patronus academicorum, Ora pro nobis.

Sancte Francisce de Sales, Patronus scriptorum, Ora pro nobis.

Sancta Rita, Patrona impossibilium, Ora pro nobis.

In loving memory of
George Daloia,
Nicola and Giuseppina Perugini, and
Bob Twarek.

*Requiem aeternam dona eis, Domine, et lux perpetua luceat eis.
Requiescant in pace. Amen.*

Contents

Preface	xvii
About the Author	xxix
List of Figures	xxxi
List of Tables	xxxv

Part I Fundamentals 1

1 Introduction	3
1.1 Text Objectives	3
1.2 Chapter Objectives	4
1.3 The World of Programming Languages	4
1.3.1 Fundamental Questions	4
1.3.2 Bindings: Static and Dynamic	6
1.3.3 Programming Language Concepts	7
1.4 Styles of Programming	8
1.4.1 Imperative Programming	8
1.4.2 Functional Programming	11
1.4.3 Object-Oriented Programming	12
1.4.4 Logic/Declarative Programming	13
1.4.5 Bottom-up Programming	15
1.4.6 Synthesis: Beyond Paradigms	16
1.4.7 Language Evaluation Criteria	19
1.4.8 Thought Process for Problem Solving	20
1.5 Factors Influencing Language Development	21
1.6 Recurring Themes in the Study of Languages	25
1.7 What You Will Learn	27
1.8 Learning Outcomes	27
1.9 Thematic Takeaways	30
1.10 Chapter Summary	31
1.11 Notes and Further Reading	32

2	Formal Languages and Grammars	33
2.1	Chapter Objectives	33
2.2	Introduction to Formal Languages	34
2.3	Regular Expressions and Regular Languages	35
2.3.1	Regular Expressions	35
2.3.2	Finite-State Automata	38
2.3.3	Regular Languages	39
2.4	Grammars and Backus–Naur Form	40
2.4.1	Regular Grammars	41
2.5	Context-Free Languages and Grammars	42
2.6	Language Generation: Sentence Derivations	44
2.7	Language Recognition: Parsing	47
2.8	Syntactic Ambiguity	48
2.8.1	Modeling Some Semantics in Syntax	49
2.8.2	Parse Trees	51
2.9	Grammar Disambiguation	57
2.9.1	Operator Precedence	57
2.9.2	Associativity of Operators	57
2.9.3	The Classical Dangling <code>else</code> Problem	58
2.10	Extended Backus–Naur Form	60
2.11	Context-Sensitivity and Semantics	64
2.12	Thematic Takeaways	67
2.13	Chapter Summary	68
2.14	Notes and Further Reading	69
3	Scanning and Parsing	71
3.1	Chapter Objectives	71
3.2	Scanning	72
3.3	Parsing	74
3.4	Recursive-Descent Parsing	76
3.4.1	A Complete Recursive-Descent Parser	76
3.4.2	A Language Generator	79
3.5	Bottom-up, Shift-Reduce Parsing and Parser Generators	80
3.5.1	A Complete Example in <code>lex</code> and <code>yacc</code>	82
3.6	PLY: Python Lex-Yacc	84
3.6.1	A Complete Example in PLY	84
3.6.2	Camille Scanner and Parser Generators in PLY	86
3.7	Top-down Vis-à-Vis Bottom-up Parsing	89
3.8	Thematic Takeaways	100
3.9	Chapter Summary	100
3.10	Notes and Further Reading	101
4	Programming Language Implementation	103
4.1	Chapter Objectives	103
4.2	Interpretation Vis-à-Vis Compilation	103
4.3	Run-Time Systems: Methods of Executions	109

4.4	Comparison of Interpreters and Compilers	114
4.5	Influence of Language Goals on Implementation	116
4.6	Thematic Takeaways	121
4.7	Chapter Summary	122
4.8	Notes and Further Reading	123
5	Functional Programming in Scheme	125
5.1	Chapter Objectives	126
5.2	Introduction to Functional Programming	126
5.2.1	Hallmarks of Functional Programming	126
5.2.2	Lambda Calculus	126
5.2.3	Lists in Functional Programming	127
5.3	Lisp	128
5.3.1	Introduction	128
5.3.2	Lists in Lisp	128
5.4	Scheme	129
5.4.1	An Interactive and Illustrative Session with Scheme	129
5.4.2	Homoiconicity: No Distinction Between Program Code and Data	133
5.5	<code>cons</code> Cells: Building Blocks of Dynamic Memory Structures	135
5.5.1	List Representation	136
5.5.2	List-Box Diagrams	136
5.6	Functions on Lists	141
5.6.1	A List <code>length</code> Function	141
5.6.2	Run-Time Complexity: <code>append</code> and <code>reverse</code>	141
5.6.3	The Difference Lists Technique	144
5.7	Constructing Additional Data Structures	149
5.7.1	A Binary Tree Abstraction	150
5.7.2	A Binary Search Tree Abstraction	151
5.8	Scheme Predicates as Recursive-Descent Parsers	153
5.8.1	<code>atom?</code> , <code>list-of-atoms?</code> , and <code>list-of-numbers?</code>	153
5.8.2	Factoring out the <code>list-of</code> Pattern	154
5.9	Local Binding: <code>let</code> , <code>let*</code> , and <code>letrec</code>	156
5.9.1	The <code>let</code> and <code>let*</code> Expressions	156
5.9.2	The <code>letrec</code> Expression	158
5.9.3	Using <code>let</code> and <code>letrec</code> to Define a Local Function	158
5.9.4	Other Languages Supporting Functional Programming: ML and Haskell	161
5.10	Advanced Techniques	166
5.10.1	More List Functions	166
5.10.2	Eliminating Expression Recomputation	167
5.10.3	Avoiding Repassing Constant Arguments Across Recur- sive Calls	167
5.11	Languages and Software Engineering	174
5.11.1	Building Blocks as Abstractions	174

5.11.2	Language Flexibility Supports Program Modification	175
5.11.3	Malleable Program Design	175
5.11.4	From Prototype to Product	175
5.12	Layers of Functional Programming	176
5.13	Concurrency	177
5.14	Programming Project for Chapter 5	178
5.15	Thematic Takeaways	179
5.16	Chapter Summary	180
5.17	Notes and Further Reading	182
6	Binding and Scope	185
6.1	Chapter Objectives	185
6.2	Preliminaries	186
6.2.1	What Is a Closure?	186
6.2.2	Static Vis-à-Vis Dynamic Properties	186
6.3	Introduction	186
6.4	Static Scoping	187
6.4.1	Lexical Scoping	187
6.5	Lexical Addressing	193
6.6	Free or Bound Variables	196
6.7	Dynamic Scoping	200
6.8	Comparison of Static and Dynamic Scoping	202
6.9	Mixing Lexically and Dynamically Scoped Variables	207
6.10	The FUNARG Problem	213
6.10.1	The Downward FUNARG Problem	214
6.10.2	The Upward FUNARG Problem	215
6.10.3	Relationship Between Closures and Scope	224
6.10.4	Uses of Closures	225
6.10.5	The Upward and Downward FUNARG Problem in a Single Function	225
6.10.6	Addressing the FUNARG Problem	226
6.11	Deep, Shallow, and Ad Hoc Binding	233
6.11.1	Deep Binding	234
6.11.2	Shallow Binding	235
6.11.3	Ad Hoc Binding	236
6.12	Thematic Takeaways	240
6.13	Chapter Summary	240
6.14	Notes and Further Reading	242
Part II	Types	243
7	Type Systems	245
7.1	Chapter Objectives	245
7.2	Introduction	245
7.3	Type Checking	246

7.4	Type Conversion, Coercion, and Casting	249
7.4.1	Type Coercion: Implicit Conversion	249
7.4.2	Type Casting: Explicit Conversion	252
7.4.3	Type Conversion Functions: Explicit Conversion	252
7.5	Parametric Polymorphism	253
7.6	Operator/Function Overloading	263
7.7	Function Overriding	267
7.8	Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing	268
7.9	Type Inference	268
7.10	Variable-Length Argument Lists in Scheme	274
7.11	Thematic Takeaways	281
7.12	Chapter Summary	281
7.13	Notes and Further Reading	283
8	Currying and Higher-Order Functions	285
8.1	Chapter Objectives	285
8.2	Partial Function Application	285
8.3	Currying	292
8.3.1	Curried Form	292
8.3.2	Currying and Uncurrying	294
8.3.3	The <code>curry</code> and <code>uncurry</code> Functions in Haskell	295
8.3.4	Flexibility in Curried Functions	297
8.3.5	All Built-in Functions in Haskell Are Curried	301
8.3.6	Supporting Curried Form Through First-Class Closures	307
8.3.7	ML Analogs	308
8.4	Putting It All Together: Higher-Order Functions	313
8.4.1	Functional Mapping	313
8.4.2	Functional Composition	315
8.4.3	Sections in Haskell	316
8.4.4	Folding Lists	319
8.4.5	Crafting Cleverly Conceived Functions with Curried HOFs	324
8.5	Analysis	334
8.6	Thematic Takeaways	335
8.7	Chapter Summary	335
8.8	Notes and Further Reading	336
9	Data Abstraction	337
9.1	Chapter Objectives	337
9.2	Aggregate Data Types	338
9.2.1	Arrays	338
9.2.2	Records	338
9.2.3	Undiscriminated Unions	341
9.2.4	Discriminated Unions	343
9.3	Inductive Data Types	344
9.4	Variant Records	347
9.4.1	Variant Records in Haskell	348

9.4.2	Variant Records in Scheme: (define-datatype ...) and (cases ...)	352
9.5	Abstract Syntax	356
9.6	Abstract-Syntax Tree for Camille	359
9.6.1	Camille Abstract-Syntax Tree Data Type: <code>TreeNode</code>	359
9.6.2	Camille Parser Generator with Tree Builder	360
9.7	Data Abstraction	366
9.8	Case Study: Environments	366
9.8.1	Choices of Representation	367
9.8.2	Closure Representation in Scheme	367
9.8.3	Closure Representation in Python	371
9.8.4	Abstract-Syntax Representation in Python	372
9.9	ML and Haskell: Summaries, Comparison, Applications, and Analysis	382
9.9.1	ML Summary	382
9.9.2	Haskell Summary	382
9.9.3	Comparison of ML and Haskell	383
9.9.4	Applications	383
9.9.5	Analysis	385
9.10	Thematic Takeaways	385
9.11	Chapter Summary	386
9.12	Notes and Further Reading	387

Part III Interpreter Implementation 389

10	Local Binding and Conditional Evaluation	391
10.1	Chapter Objectives	391
10.2	Checkpoint	391
10.3	Overview: Learning Language Concepts Through Interpreters	393
10.4	Preliminaries: Interpreter Essentials	394
10.4.1	Expressed Values Vis-à-Vis Denoted Values	394
10.4.2	Defined Language Vis-à-Vis Defining Language	395
10.5	The Camille Grammar and Language	395
10.6	A First Camille Interpreter	396
10.6.1	Front End for Camille	396
10.6.2	Simple Interpreter for Camille	399
10.6.3	Abstract-Syntax Trees for Arguments Lists	401
10.6.4	REPL: Read-Eval-Print Loop	403
10.6.5	Connecting the Components	404
10.6.6	How to Run a Camille Program	404
10.7	Local Binding	405
10.8	Conditional Evaluation	410
10.9	Putting It All Together	411
10.10	Thematic Takeaways	419

10.11	Chapter Summary	419
10.12	Notes and Further Reading	421
11	Functions and Closures	423
11.1	Chapter Objectives	423
11.2	Non-recursive Functions	423
11.2.1	Adding Support for User-Defined Functions to Camille	423
11.2.2	Closures	426
11.2.3	Augmenting the <code>evaluate_expr</code> Function	427
11.2.4	A Simple Stack Object	430
11.3	Recursive Functions	440
11.3.1	Adding Support for Recursion in Camille	440
11.3.2	Recursive Environment	441
11.3.3	Augmenting <code>evaluate_expr</code> with New Variants	445
11.4	Thematic Takeaways	450
11.5	Chapter Summary	451
11.6	Notes and Further Reading	456
12	Parameter Passing	457
12.1	Chapter Objectives	457
12.2	Assignment Statement	457
12.2.1	Use of Nested <code>lets</code> to Simulate Sequential Evaluation	458
12.2.2	Illustration of Pass-by-Value in Camille	459
12.2.3	Reference Data Type	460
12.2.4	Environment Revisited	462
12.2.5	Stack Object Revisited	463
12.3	Survey of Parameter-Passing Mechanisms	467
12.3.1	Pass-by-Value	467
12.3.2	Pass-by-Reference	472
12.3.3	Pass-by-Result	477
12.3.4	Pass-by-Value-Result	478
12.3.5	Summary	481
12.4	Implementing Pass-by-Reference in the Camille Interpreter	485
12.4.1	Revised Implementation of References	486
12.4.2	Reimplementation of the <code>evaluate_operand</code> Function	487
12.5	Lazy Evaluation	492
12.5.1	Introduction	492
12.5.2	β -Reduction	492
12.5.3	C Macros to Demonstrate Pass-by-Name: β -Reduction Examples	495
12.5.4	Two Implementations of Lazy Evaluation	499
12.5.5	Implementing Lazy Evaluation: Thunks	501
12.5.6	Lazy Evaluation Enables List Comprehensions	506
12.5.7	Applications of Lazy Evaluation	511
12.5.8	Analysis of Lazy Evaluation	511
12.5.9	Purity and Consistency	512

12.6	Implementing Pass-by-Name/Need in Camille: Lazy Camille	522
12.7	Sequential Execution in Camille	527
12.8	Camille Interpreters: A Retrospective	533
12.9	Metacircular Interpreters	539
12.10	Thematic Takeaways	542
12.11	Chapter Summary	543
12.12	Notes and Further Reading	544

Part IV Other Styles of Programming 545

13	Control and Exception Handling	547
13.1	Chapter Objectives	548
13.2	First-Class Continuations	548
13.2.1	The Concept of a Continuation	548
13.2.2	Capturing First-Class Continuations: <code>call/cc</code>	550
13.3	Global Transfer of Control with Continuations	556
13.3.1	Nonlocal Exits	556
13.3.2	Breakpoints	560
13.3.3	First-Class Continuations in Ruby	562
13.4	Other Mechanisms for Global Transfer of Control	570
13.4.1	The <code>goto</code> Statement	570
13.4.2	Capturing and Restoring Control Context in C: <code>setjmp</code> and <code>longjmp</code>	571
13.5	Levels of Exception Handling in Programming Languages: A Summary	579
13.5.1	Function Calls	580
13.5.2	Lexically Scoped Exceptions: <code>break</code> and <code>continue</code>	581
13.5.3	Stack Unwinding/Crawling	581
13.5.4	Dynamically Scoped Exceptions: Exception-Handling Systems	582
13.5.5	First-Class Continuations	583
13.6	Control Abstraction	585
13.6.1	Coroutines	586
13.6.2	Applications of First-Class Continuations	589
13.6.3	The Power of First-Class Continuations	590
13.7	Tail Recursion	594
13.7.1	Recursive Control Behavior	594
13.7.2	Iterative Control Behavior	596
13.7.3	Tail-Call Optimization	598
13.7.4	Space Complexity and Lazy Evaluation	601
13.8	Continuation-Passing Style	608
13.8.1	Introduction	608
13.8.2	A Growing Stack or a Growing Continuation	610
13.8.3	An All-or-Nothing Proposition	613

13.8.4	Trade-off Between Time and Space Complexity	614
13.8.5	<code>call/cc</code> Vis-à-Vis CPS	617
13.9	Callbacks	618
13.10	CPS Transformation	620
13.10.1	Defining <code>call/cc</code> in Continuation-Passing Style	622
13.11	Thematic Takeaways	635
13.12	Chapter Summary	636
13.13	Notes and Further Reading	640
14	Logic Programming	641
14.1	Chapter Objectives	642
14.2	Propositional Calculus	642
14.3	First-Order Predicate Calculus	644
14.3.1	Representing Knowledge as Predicates	645
14.3.2	Conjunctive Normal Form	646
14.4	Resolution	648
14.4.1	Resolution in Propositional Calculus	648
14.4.2	Resolution in Predicate Calculus	649
14.5	From Predicate Calculus to Logic Programming	651
14.5.1	Clausal Form	651
14.5.2	Horn Clauses	653
14.5.3	Conversion Examples	654
14.5.4	Motif of Logic Programming	656
14.5.5	Resolution with Propositions in Clausal Form	657
14.5.6	Formalism Gone Awry	660
14.6	The Prolog Programming Language	660
14.6.1	Essential Prolog: Asserting Facts and Rules	662
14.6.2	Casting Horn Clauses in Prolog Syntax	663
14.6.3	Running and Interacting with a Prolog Program	663
14.6.4	Resolution, Unification, and Instantiation	665
14.7	Going Further in Prolog	667
14.7.1	Program Control in Prolog: A Binary Tree Example	667
14.7.2	Lists and Pattern Matching in Prolog	672
14.7.3	List Predicates in Prolog	674
14.7.4	Primitive Nature of <code>append</code>	675
14.7.5	Tracing the Resolution Process	676
14.7.6	Arithmetic in Prolog	677
14.7.7	Negation as Failure in Prolog	678
14.7.8	Graphs	679
14.7.9	Analogs Between Prolog and an RDBMS	681
14.8	Imparting More Control in Prolog: Cut	691
14.9	Analysis of Prolog	701
14.9.1	Prolog Vis-à-Vis Predicate Calculus	701
14.9.2	Reflection in Prolog	703
14.9.3	Metacircular Prolog Interpreter and WAM	704

14.10	The CLIPS Programming Language	705
14.10.1	Asserting Facts and Rules	705
14.10.2	Variables	706
14.10.3	Templates	707
14.10.4	Conditional Facts in Rules	708
14.11	Applications of Logic Programming	709
14.11.1	Natural Language Processing	709
14.11.2	Decision Trees	710
14.12	Thematic Takeaways	710
14.13	Chapter Summary	710
14.14	Notes and Further Reading	712
15	Conclusion	713
15.1	Language Themes Revisited	714
15.2	Relationship of Concepts	714
15.3	More Advanced Concepts	716
15.4	Bottom-up Programming	716
15.5	Further Reading	719
Appendix A	Python Primer	721
A.1	Appendix Objective	721
A.2	Introduction	722
A.3	Data Types	722
A.4	Essential Operators and Expressions	725
A.5	Lists	731
A.6	Tuples	733
A.7	User-Defined Functions	734
A.7.1	Simple User-Defined Functions	734
A.7.2	Positional Vis-à-Vis Keyword Arguments	735
A.7.3	Lambda Functions	738
A.7.4	Lexical Closures	739
A.7.5	More User-Defined Functions	740
A.7.6	Local Binding and Nested Functions	742
A.7.7	Mutual Recursion	744
A.7.8	Putting It All Together: Mergesort	744
A.8	Object-Oriented Programming in Python	748
A.9	Exception Handling	750
A.10	Thematic Takeaway	754
A.11	Appendix Summary	754
A.12	Notes and Further Reading	755
Appendix B	Introduction to ML (Online)	757
B.1	Appendix Objective	757
B.2	Introduction	757
B.3	Primitive Types	758
B.4	Essential Operators and Expressions	758

B.5	Running an ML Program	760
B.6	Lists	762
B.7	Tuples	763
B.8	User-Defined Functions	764
B.8.1	Simple User-Defined Functions	764
B.8.2	Lambda Functions	764
B.8.3	Pattern-Directed Invocation	765
B.8.4	Local Binding and Nested Functions: <code>let</code> Expressions . . .	768
B.8.5	Mutual Recursion	770
B.8.6	Putting It All Together: Mergesort	770
B.9	Declaring Types	774
B.9.1	Inferred or Deduced	774
B.9.2	Explicitly Declared	774
B.10	Structures	775
B.11	Exceptions	776
B.12	Input and Output	776
B.12.1	Input	776
B.12.2	Parsing an Input File	777
B.12.3	Output	778
B.13	Thematic Takeaways	781
B.14	Appendix Summary	782
B.15	Notes and Further Reading	782
Appendix C Introduction to Haskell (Online)		783
C.1	Appendix Objective	783
C.2	Introduction	783
C.3	Primitive Types	784
C.4	Type Variables, Type Classes, and Qualified Types	785
C.5	Essential Operators and Expressions	787
C.6	Running a Haskell Program	789
C.7	Lists	791
C.8	Tuples	792
C.9	User-Defined Functions	793
C.9.1	Simple User-Defined Functions	793
C.9.2	Lambda Functions	794
C.9.3	Pattern-Directed Invocation	795
C.9.4	Local Binding and Nested Functions: <code>let</code> Expressions . . .	799
C.9.5	Mutual Recursion	801
C.9.6	Putting It All Together: Mergesort	802
C.10	Declaring Types	806
C.10.1	Inferred or Deduced	806
C.10.2	Explicitly Declared	806
C.11	Thematic Takeaways	810
C.12	Appendix Summary	810
C.13	Notes and Further Reading	810

Appendix D	Getting Started with the Camille Programming Language (Online)	811
D.1	Appendix Objective	811
D.2	Grammar	811
D.3	Installation	813
D.4	Git Repository Structure and Setup	813
D.5	How to Use Camille in a Programming Languages Course	814
D.5.1	Module 0: Front End (Scanner and Parser)	814
D.5.2	Chapter 10 Module: Introduction (Local Binding and Conditionals)	814
D.5.3	Configuring the Language	815
D.5.4	Chapter 11 Module: Intermediate (Functions and Closures)	816
D.5.5	Chapter 12 Modules: Advanced (Parameter Passing, Including Lazy Evaluation) and Imperative (Statements and Sequential Evaluation)	818
D.6	Example Usage: Non-interactively and Interactively (CLI)	818
D.7	Solutions to Programming Exercises in Chapters 10–12	819
D.8	Notes and Further Reading	821
Appendix E	Camille Grammar and Language (Online)	823
E.1	Appendix Objective	823
E.2	Camille 0.1: Numbers and Primitives	823
E.3	Camille 1.x: Local Binding and Conditional Evaluation	824
E.4	Camille 2.x: Non-recursive and Recursive Functions	825
E.5	Camille 3.x: Variable Assignment and Support for Arrays	826
E.6	Camille 4.x: Sequential Execution	827
Bibliography		B-1
Index		I-1

Preface

I hear and I forget, I see and I remember, I do and I understand.

— Confucius

What we have to learn to do, we learn by doing

— Aristotle, *Ethics*

Learning should be an adventure, a quest, a romance.

— Gretchen E. Smalley

THIS text is about programming language concepts. The goal is not to learn the nuances of particular languages, but rather to explore and establish a deeper understanding of the general concepts or principles of programming languages, with a particular emphasis on programming. Such an understanding prepares us to evaluate how a variety of languages address these concepts and to discern the appropriate languages for a given task. It also arms us with a larger toolkit of programming techniques from which to build abstractions. The text’s objectives and the recurring themes and learning outcomes of this course of study are outlined in Sections 1.1, 1.6, and 1.8, respectively.

This text is intended for the student who enjoys problem solving, programming, and exploring new ways of thinking and programming languages that support those views. It exposes readers to alternative styles of programming. The text challenges students to program in languages beyond what they may have encountered thus far in their university studies of computer science studies—specifically, to write programs in languages that do not have variables.

Locus of Focus: Notes for Instructors

This text focuses on the concepts of programming languages that constitute requisite knowledge for undergraduate computer science students. Thus, it is intentionally light on topics that most computing curricula emphasize in other courses. A course in programming languages emphasizes topics that students typically do not experience in other courses: functional programming (Chapter 5), typing (Chapters 7–9), interpreter implementation (Chapters 10–12), control abstraction (Chapter 13), logic/declarative programming (Chapter 14), and, more

generally, formalizing the language concepts and the design/implementation options for those concepts that students experience through programming. We also emphasize eclectic language features and programming techniques that lead to powerful and practical programming abstractions: currying, lazy evaluation, and first-class continuations (e.g., `call/cc`).

Book Overview: At a Glance

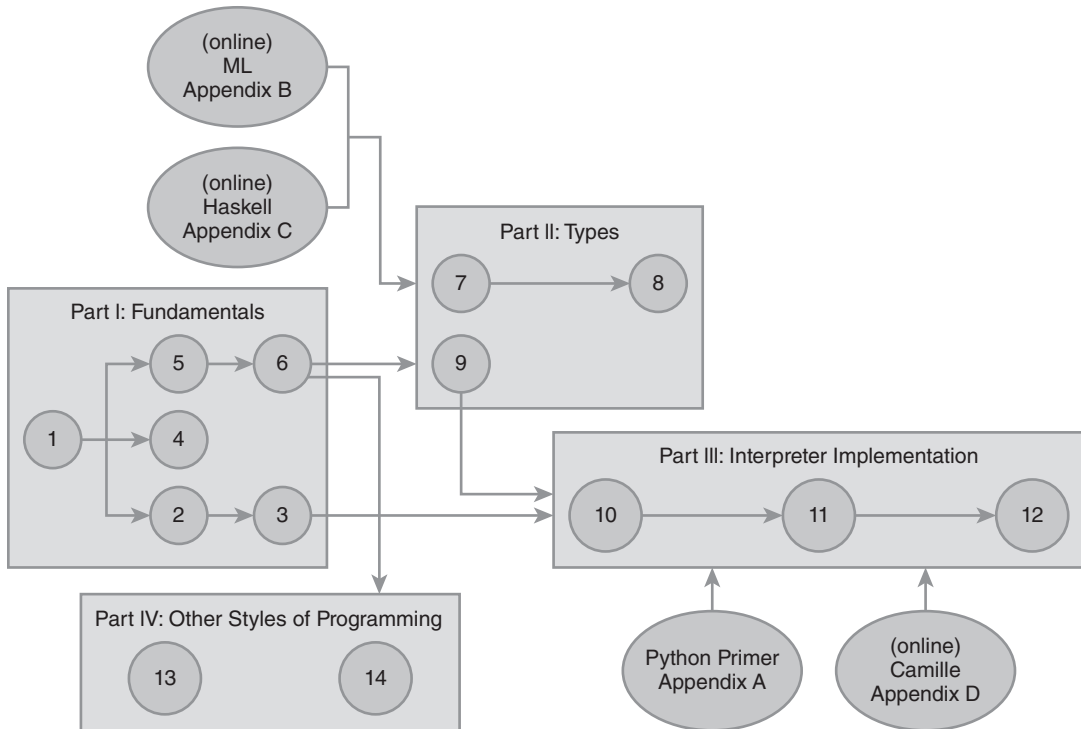
The approach to distilling language concepts is captured in the following sequence of topics:

Module	Chapters	Topic(s)	Language(s) Used
I.	1–6	Fundamentals and Foundational Functional Programming	← Scheme and Python
		↓	
II.	7–9	Typing Concepts and Data Abstraction	← ML, Haskell, and Python
		↓	
III.	10–12	Interpreter Implementation	← Python
		↓	
IV.	13–14	Other Styles of Programming	
		{programming with continuations;	← Scheme and Ruby
		logic/declarative programming}	← Prolog and CLIPS

Before we implement concepts in languages, we commence by studying the most fundamental principles of languages and developing a vocabulary of concepts for subsequent study. Chapter 2 covers language definition and description methods (e.g., grammars). We also explore the fundamentals of functional programming (primarily in Scheme in Chapter 5), which is quite different from the styles of programming predominant in the languages with which students are probably most familiar. To manage the complexity inherent in interpreters, we must make effective use of data abstraction techniques. Thus, we also study data abstraction and, specifically, how to define inductive data types, as well as representation strategies to use in the implementation of abstract data types. In Chapters 10–12, we implement a progressive series of interpreters in Python, using functional and object-oriented techniques, for a language called Camille that operationalize the concepts that we study in the first module, including binding, scope, and recursion, and assess the differences in the resulting versions of Camille. Following the interpreter implementation module, we fan out and explore other styles of programming. A more detailed outline of the topics covered is given in Section 1.7.

Chapter Dependencies

The following figure depicts the dependencies between the chapters of this text.



Instructors can take multiple pathways through this text to customize their languages course. Within each of the following tracks, instructors can add or subtract material based on these chapter dependencies to suit the needs of their students.

Multiple Pathways

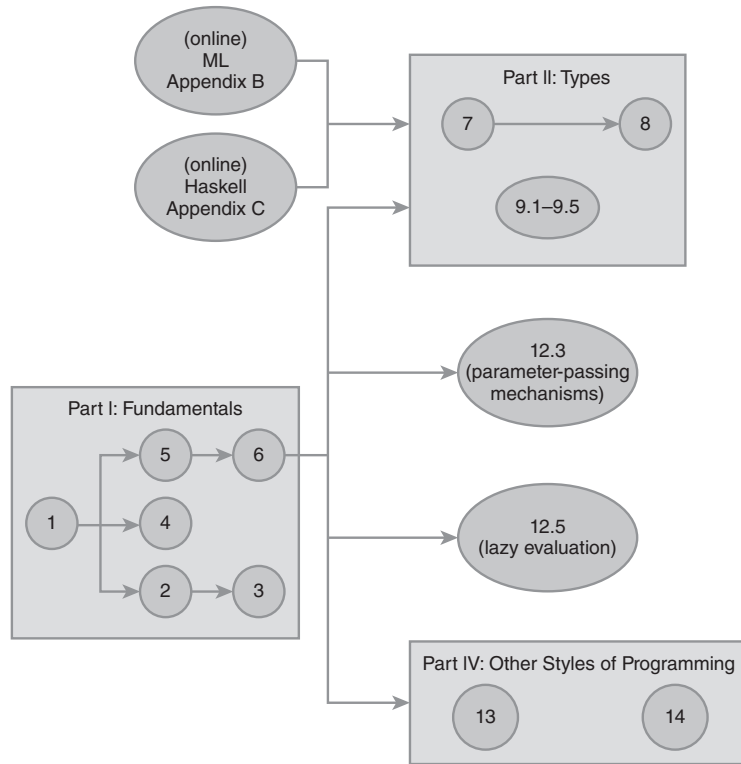
Since the content in this text is arranged in a modular fashion, the pathways through it are customizable.

Customized Courses of Study

Multiple approaches may be taken toward establishing an understanding of programming language concepts. One way to learn language principles is to study how they are supported in a variety of programming languages and to write programs in those languages to probe those concepts. Another approach to learning language concepts is to implement them by building interpreters for computer languages—the focus of Chapters 10–12. Yet another avenue involves a hybrid of these two approaches. The following tracks through the chapters of this text represent the typical approaches to teaching programming languages.

Concepts-Based Approach

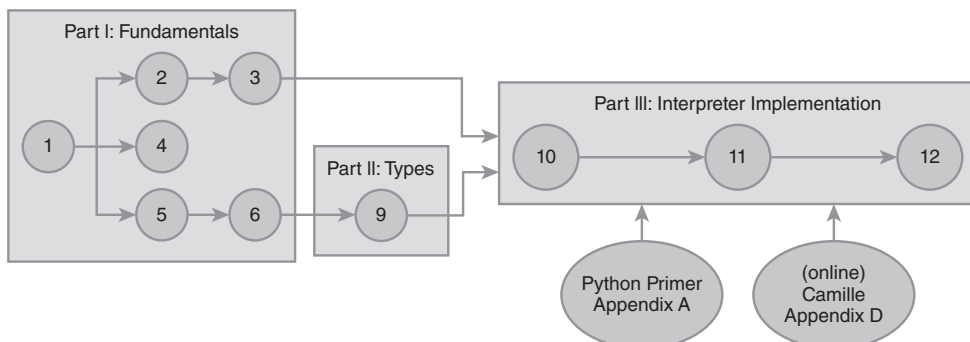
The following figure demonstrates the *concepts-based approach* through the text.



The path through the text modeled here focuses solely on the conceptual parts of Chapters 9 and 10–12, and omits the “Interpreter Implementation” module in favor of the “Other Styles of Programming” module.

Interpreter-Based Approach

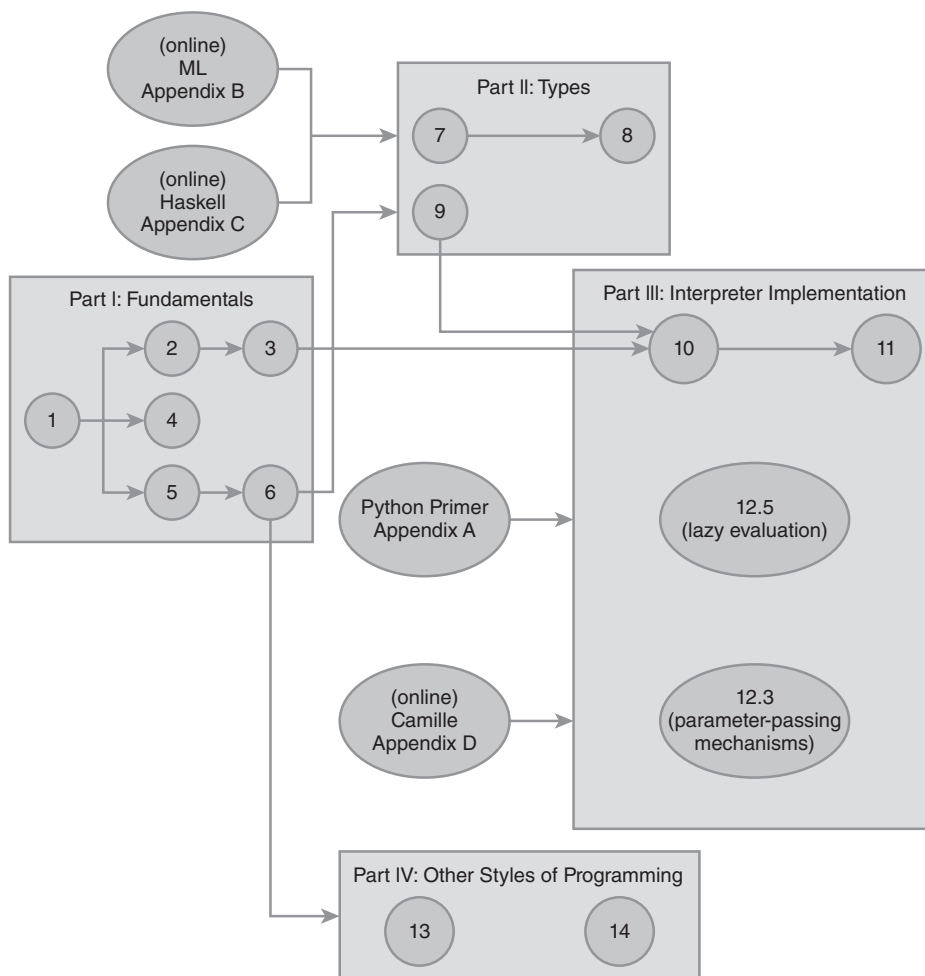
The following figure demonstrates the *interpreter-based approach* using Python.



This approach is the complement of the concepts-only approach, in that it uses the “Interpreter Implementation” module and the entirety of Chapter 9 instead of the “Other Styles of Programming” module and the conceptual chapters of the “Types” module (i.e., Chapters 7–8).

Hybrid Concepts/Interpreter Approach

The following approach involves a *synthesis of the concepts- and interpreter-based approaches*.



The pathway modeled here retains the entirety of each of the “Types” and “Other Styles of Programming” modules, but omits Chapter 12 of the “Interpreter Implementation” module, except for the conceptual parts (i.e., the survey of parameter-passing mechanisms, including lazy evaluation).

Mapping from ACM/IEEE Curriculum to Chapters

Table 1 presents a mapping from the nine draft competencies (A–I) for programming languages in the *ACM/IEEE Computing Curricula 2020* (Computing Curricula 2020 Task Force 2020, p. 113) to the chapters of this text where the material leading to those competencies is addressed or covered.

Table 2 presents a mapping from the 17 topics in the *ACM/IEEE Curriculum Standards for Programming Languages in Undergraduate CS Degree Programs 2013* [The Joint Task Force on Computing Curricula: Association for Computing Machinery (ACM) and IEEE Computer Society 2013, pp. 155–166] to the chapters of this text where they are covered.

Prerequisites for This Course of Study

This book assumes no prior experience with functional or declarative programming or programming in Python, Scheme, Haskell, ML, Prolog, or any of the other languages used in the text. However, we assume that readers are familiar with intermediate imperative and/or object-oriented programming in a block-structured language, such as Java or C++, and have had courses in both data structures and discrete mathematics.

The examples in this text are presented in multiple languages—this is necessary to detach students from an *una lingua* mindset. However, to keep things simple, the only languages students need to know to progress through this text are Python (Appendix A is a primer on Python programming), Scheme (covered in Chapter 5), and either ML or Haskell (covered in the online appendices). Beyond these languages, a working understanding of Java or C/C++ is sufficient to follow the code snippets and examples because they often use a Java/C-like syntax.

Beyond these requisites, an intellectual and scientific curiosity, a thirst for learning new concepts and exploring compelling ideas, and an inclination to experience familiar ideas from new perspectives are helpful dispositions for this course of study.

A message I aspire to convey throughout this text is that programming should be creative, artistic, and a joy, and programs should be beautiful. The study of programming languages ties multiple loose ends in the study of computer science together and helps foster a more holistic view of the discipline of computing. I hope readers experience multiple epiphanies as they work through the concepts presented and are as mystified as I was the first time I explored and discovered this material. Let the journey begin.

Note to Readers

Establishing an understanding of the organization and concepts of programming languages and the elegant programming abstractions/techniques enabled by a mastery of those concepts requires work. This text encourages its reader to learn

Competency	Chapter(s)
A. Present the design and implementation of a class considering object-oriented encapsulation mechanisms (e.g., class hierarchies, interfaces, and private members).	10–12
B. Produce a brief report on the implementation of a basic algorithm considering control flow in a program using dynamic dispatch that avoids assigning to a mutable state (or considering reference equality) for two different languages.	5
C. Present the implementation of a useful function that takes and returns other functions considering variables and lexical scope in a program as well as functional encapsulation mechanisms.	5–6, 8–9
D. Use iterators and other operations on aggregates (including operations that take functions as arguments) in two programming languages and present to a group of professionals some ways of selecting the most natural idioms for each language.	5, 8, 12–13
E. Contrast and present to peers	
(1) the procedural/functional approach (defining a function for each operation with the function body providing a case for each data variant) and	8–9
(2) the object-oriented approach (defining a class for each data variant with the class definition providing a method for each operation).	10–12
F. Write event handlers for a web developer for use in reactive systems such as GUIs.	13
G. Demonstrate program pieces (such as functions, classes, methods) that use generic or compound types, including for collections to write programs.	7–13
H. Write a program for a client to process a representation of code that illustrates the incorporation of an interpreter, an expression optimizer, and a documentation generator.	5, 10–13
I. Use type-error messages, memory leaks, and dangling-pointer to debug a program for an engineering firm.	6–7

Table 1 Mapping from the *ACM/IEEE Computing Curricula 2020* to Chapters of This Text

language concepts much as one learns to swim or drive a car—not just by reading about it, but by *doing* it—and within that space lies the joy. A key theme of this text is the emphasis on implementation. The programming exercises afford the reader ample opportunities to implement the language concepts we discuss and require a fair amount of critical thought and design.

Tier	Topic	Hours	Chapter(s)
1 and 2	Object-Oriented Programming	4 + 6	9–12
1 and 2	Functional Programming	3 + 4	5
2	Event-Driven and Reactive Programming	0 + 2	13
1 and 2	Basic Type Systems	1 + 4	9
2	Program Representation	0 + 1	2, 9
2	Language Translation and Execution	0 + 3	3–4, 10–12
E	Syntax Analysis	—	3
E	Compiler Semantic Analysis	—	6, 10–12
E	Code Generation	—	3–4
E	Runtime Systems	—	4, 10–12
E	Static Analysis	—	10–12
E	Advanced Programming Constructs	—	6, 13
E	Concurrency and Parallelism	—	13
E	Type Systems	—	9
E	Formal Semantics	—	2
E	Language Pragmatics	—	10–12
E	Logic Programming	—	14

Table 2 Mapping from the 2013 ACM/IEEE Computing Curriculum Standards to Chapters of This Text

Moreover, this text is not intended to be read passively. Students are encouraged to read the text with their Python, Racket Scheme, ML, Haskell, or Prolog interpreter open to enter the expressions as they read them so that they can follow along with our discussion. The reward of these mechanics is a more profound understanding of language concepts resulting from having implemented them, and the epiphanies that emerge during the process.

Lastly, I hope to (1) develop and improve readers’ ability to generalize patterns from the examples provided, and subsequently (2) develop their aptitude and intuition for quickly recognizing new instances of these self-learned patterns when faced with similar problems in domains/contexts in which they have little experience. Thus, many of the exercises seek to evaluate how well readers can synthesize the concepts and ideas presented for use when independently approaching and solving unfamiliar problems.

Supplemental Material

Supplemental material for this text, including presentation slides and other instructor-related resources, is available online.

Source Code Availability

The source code of the Camille interpreters in Python developed in Chapters 10–12 is available as a Git repository in BitBucket at <https://bitbucket.org/camilleinterpreter/camille-interpreter-in-python-release/>.

Solutions to Conceptual and Programming Exercises

Solutions to all of the conceptual and programming exercises are available only to instructors at go.jblearning.com/Peruginile or by contacting your Jones & Bartlett Learning sales representative.

Programming Language Availability

C	http://www.open-std.org/jtc1/sc22/wg14/
C++	https://isocpp.org
CLIPS	http://www.clipsrules.net/
Common Lisp	https://clisp.org
Elixir	https://elixir-lang.org
Go	https://golang.org
Java	https://java.com
JavaScript	https://www.javascript.com
Julia	https://julialang.org
Haskell	https://www.haskell.org
Lua	https://lua.org
ML	https://smlnj.org
Perl	https://www.perl.org
Prolog	https://www.swi-prolog.org
Python	https://python.org
Racket	https://racket-lang.org
Ruby	https://www.ruby-lang.org
Scheme	https://www.scheme.com
Smalltalk	https://squeak.org

Acknowledgments

With a goal of nurturing students, and with an abiding respect for the craft of teaching and professors who strive to teach well, I have sought to produce a text that both illuminates language concepts that are enlightening to the mind and is faithful and complete as well as useful and practical. Doing so has been a labor of love. This text would not have been possible without the support and inspiration from a variety of sources.

I owe a debt of gratitude to the computer scientists with expertise in languages who, through authoring the beautifully crafted textbooks from which I originally

learned this material, have broken new ground in the pedagogy of programming languages: Abelson and Sussman (1996); Friedman, Wand, and Haynes (2001); and Friedman and Felleisen (1996a, 1996b). I am particularly grateful to the scholars and educators who originally explored the language landscape and how to most effectively present the concepts therein. They shared their results with the world through the elegant and innovative books they wrote with precision and flair. You are truly inspirational. My view of programming languages and how best to teach languages has been informed and influenced by these seminal books. In writing this text, I was particularly inspired by *Essentials of Programming Languages* (Friedman, Wand, and Haynes 2001). Chapters 10–11 and Sections 12.2, 12.4, 12.6, and 12.7 of this text are inspired by their Chapter 3. Our contribution is the use of Python to build EOPL-style interpreters. *The Little Schemer* (Friedman and Felleisen 1996a) and *The Seasoned Schemer* (Friedman and Felleisen 1996b) were a delight to read and work through, and *The Structure and Interpretation of Computer Programs* (Abelson and Sussman 1996) will always be a classic. These books are gifts to our field.

Other books have also been inspiring and influential in forming my approach to teaching and presenting language concepts, including Dybvig 2009, Graham (2004b, 1993), Kamin (1990), Hutton (2007), Krishnamurthi (2003, 2017), Thompson (2007), and Ullman (1997). Readers familiar with these books will observe their imprint here. I have attempted to weave a new tapestry here from the palette set forth in these books through my synthesis of a conceptual/principles-based approach with an interpreter-based approach. I also thank James D. Arthur, Naren Ramakrishnan, and Stephen H. Edwards at Virginia Tech, who first shared this material with me.

I have also been blessed with bright, generous, and humble students who have helped me with the development of this text in innumerable ways. Their help is heartfelt and very much appreciated. In particular, Jack Watkin, Brandon Williams, and Zachary Rowland have contributed significant time and effort. I am forever thankful to and for you. I also thank other University of Dayton students and alumni of the computer science program for helping in various ways, including Travis Suel, Patrick Marsee, John Cresencia, Anna Duricy, Masood Firoozabadi, Adam Volk, Stephen Korenewych, Joshua Buck, Tyler Masthay, Jonathon Reinhart, Howard Poston, and Philip Bohun.

I thank my colleagues Phu Phung and Xin Chen for using preliminary editions of this text in their courses. I also thank the students at the University of Dayton who used early manuscripts of this text in their programming languages courses and provided helpful feedback.

Thanks to John Lewis at Virginia Tech for putting me in contact with Jones & Barlett Learning and providing guidance throughout the process of bringing this text to production. I thank Simon Thompson at the University of Kent (in the United Kingdom) for reviewing a draft of this manuscript and providing helpful feedback. I am grateful to Doug Hodson at the Air Force Institute of Technology and Kim Conde at the University of Dayton for providing helpful

editorial comments. Thanks to Julianne Morgan at the University of Dayton for being generous with her time and helping in a variety of ways.

Many thanks to the University of Dayton and the Department of Computer Science, in particular, for providing support, resources, and facilities, including two sabbaticals, to make this text possible. I also thank the team at Jones & Bartlett Learning, especially Ned Hinman, Melissa Duffy, Jennifer Ridsen, Sue Boshers, Jill Hobbs, and James Fortney for their support throughout the entire production process.

I thank Camille and Carla for their warm hospitality and the members of the Corpus Christi FSSP Mission in Naples, Florida, especially Father Dorsa, Katy Allen, Connor DeLoach, Rosario Sorrentino, and Michael Piedimonte, for their prayers and support.

I thank my parents, Saverio and Georgeanna Perugini, and grandmother, Lucia Daloia, for love and support. Thank you to Mary and Patrick Sullivan; Matthew and Hilary Barhorst and children; Ken and Mary Beth Artz; and Steve and Mary Ann Berning for your friendship and the kindness you have shown me. Lastly, I thank my best friends—my *Holy Family* family—for your love, prayers, and constant supportive presence in my life: Dimitri Savvas; Dan Warner; Jim and Christina Warner; Maria, Angela, Rosa, Patrick, Joseph, Carl, and Gina Hess; Vince, Carol, and Tosca. I love you. *Deo gratias. Ave Maria.*

Saverio Perugini
April 2021

About the Author

Saverio Perugini is a professor in the Department of Computer Science at the University of Dayton. He has a PhD in computer science from Virginia Tech.

List of Figures

1.1	Conceptual depiction of a set of objects communicating by message passing.	13
1.2	Within the context of their support for a variety of programming styles, all languages involve a core set of universal concepts.	20
1.3	Programming languages and the styles of programming therein are conduits into computation.	22
1.4	Evolution of programming languages across a time axis.	24
1.5	Factors influencing language design.	25
2.1	A finite-state automaton for a legal identifier and positive integer in C.	38
2.2	The dual nature of grammars as generative and recognition devices.	47
2.3	Two parse trees for the expression $x + y * z$	51
2.4	Parse trees for the expression x	52
2.5	Parse tree for the expression 132.	53
2.6	Parse trees for the expression $1 + 3 + 2$	53
2.7	Parse trees for the expression $1 + 3 * 2$	54
2.8	Parse trees for the expression $6 - 3 - 2$	54
2.9	Parse trees for the sentence <code>if (a < 2) if (b > 3) x else y</code>	59
3.1	Simplified view of scanning and parsing: the front end.	73
3.2	More detailed view of scanning and parsing.	74
3.3	A finite-state automaton for a legal identifier and positive integer in C.	74
4.1	Execution by interpretation.	104
4.2	Execution by compilation.	105
4.3	Interpreter for language <i>simple</i>	108
4.4	Low-level view of execution by compilation.	110
4.5	Alternative view of execution by interpretation.	112
4.6	Four different approaches to language implementation.	113
4.7	Mutually dependent relationship between compilers and interpreters.	114

5.1	List box representation of a <code>cons</code> cell.	136
5.2	' (a b) = ' (a . (b))	137
5.3	' (a b c) = ' (a . (b c)) = ' (a . (b . (c)))	137
5.4	' (a . b)	137
5.5	' ((a) (b) ((c))) = ' ((a) . ((b) ((c))))	138
5.6	' (((a) b) c)	138
5.7	' ((a b) c) = ' (((a) b) . (c)) = ' (((a) . (b)) . (c))	138
5.8	' ((a . b) . c)	138
5.9	Graphical depiction of the foundational nature of <code>lambda</code>	160
5.10	Layers of functional programming.	177
6.1	Run-time call stack at the time the expression <code>(+ a b x)</code> is evaluated.	189
6.2	Static call graph of the program illustrating dynamic scoping in Section 6.7.	200
6.3	Two run-time call stacks possible from dynamic scoping program in Section 6.7.	201
6.4	Run-time stack at <code>print</code> call on line 37 of program of Listing 6.2.	209
6.5	Illustration of the upward FUNARG problem.	215
6.6	The heap in a process from which dynamic memory is allocated.	227
7.1	Hierarchy of concepts to which the study of typing leads.	283
8.1	<code>foldr</code> using the right-associative <code>:</code> <code>cons</code> operator.	320
8.2	<code>foldl</code> in Haskell (left) vis-à-vis <code>foldl</code> in ML (right).	321
9.1	Abstract-syntax tree for <code>((lambda (x) (f x)) (g y))</code>	358
9.2	(left) Visual representation of <code>TreeNode</code> Python class. (right) A value of type <code>TreeNode</code> for an identifier.	363
9.3	An abstract-syntax representation of a named environment in Python.	373
9.4	An abstract-syntax representation of a named environment in Racket Scheme.	376
9.5	A list-of-lists representation of a named environment in Scheme.	378
9.6	A list-of-vectors representation of a nameless environment in Scheme.	379
9.7	A list-of-lists representation of a named environment in Python	380
9.8	A list-of-lists representation of a nameless environment in Python	380
9.9	An abstract-syntax representation of a nameless environment in Racket Scheme	381
9.10	An abstract-syntax representation of a nameless environment in Python	382
10.1	Execution by interpretation.	396
10.2	Abstract-syntax tree for the Camille expression <code>* (7, x)</code>	402

10.3	Abstract-syntax tree for the Camille expression let x = 1 y = 2 in *(x,y).	409
10.4	Dependencies between Camille interpreters thus far.	420
11.1	Abstract-syntax representation of our Closure data type in Python.	426
11.2	An abstract-syntax representation of a non-recursive, named environment	428
11.3	A list-of-lists representation of a non-recursive, named environment	429
11.4	Dependencies between Camille interpreters thus far.	433
11.5	A list-of-lists representation of a non-recursive, nameless environment.	438
11.6	An abstract-syntax representation of a non-recursive, nameless environment	439
11.7	An abstract-syntax representation of a circular, recursive, named environment.	444
11.8	A list-of-lists representation of a circular, recursive, named environment.	445
11.9	Dependencies between Camille interpreters supporting functions thus far.	448
11.10	An abstract-syntax representation of a circular, recursive, nameless environment.	449
11.11	A list-of-lists representation of a circular, recursive, nameless environment.	449
11.12	Dependencies between Camille interpreters thus far.	453
12.1	A primitive reference to an element in a Python list.	460
12.2	Passing arguments by value in C.	468
12.3	Passing of references (to objects) by value in Java.	472
12.4	Passing arguments by value in Scheme.	473
12.5	The pass-by-reference parameter-passing mechanism in C++.	476
12.6	Passing memory-address arguments by value in C.	477
12.7	Passing arguments by result.	479
12.8	Passing arguments by value-result.	480
12.9	Summary of parameter-passing concepts in Java, Scheme, C, and C++	482
12.10	Three layers of references to indirect and direct targets represent- ing parameters to functions.	491
12.11	Passing variables by reference in Camille.	491
12.12	Dependencies between Camille interpreters.	534
12.13	Dependencies between Camille interpreters thus far.	536
13.1	The general call/cc continuation capture and invocation process.	553
13.2	Example of call/cc continuation capture and invocation process.	554

13.3	The run-time stack during the continuation replacement process depicted in Figure 13.2.	554
13.4	The run-time stacks in the <code>factorial</code> example in C.	574
13.5	The run-time stacks in the <code>jumpstack.c</code> example.	576
13.6	Data and procedural abstraction with control abstraction as an afterthought.	585
13.7	Recursive control behavior (left) vis-à-vis iterative control behavior (right).	596
13.8	Decision tree for the use of <code>foldr</code> , <code>foldl</code> , and <code>foldl'</code> in designing functions.	607
13.9	Both <code>call/cc</code> and CPS involve reification and support control abstraction.	618
13.10	<i>Program readability/writability</i> vis-à-vis <i>space complexity</i>	621
13.11	CPS transformation and subsequent low-level <code>let-to-lambda</code> transformations.	622
14.1	The theoretical foundations of <i>functional</i> and <i>logic programming</i> are λ -calculus and <i>first-order predicate calculus</i> , respectively.	645
14.2	A search tree illustrating the resolution process.	669
14.3	An alternative search tree illustrating the resolution process.	670
14.4	Search tree illustrating an infinite expansion of the <code>path</code> predicate in the resolution process used to satisfy the goal <code>path(X, c)</code>	671
14.5	The branch of the resolution search tree for the <code>path(X, c)</code> goal that the cut operator removes in the first <code>path</code> predicate.	692
14.6	The branch of the resolution search tree for the <code>path(X, c)</code> goal that the cut operator removes in the second <code>path</code> predicate.	694
14.7	The branch of the resolution search tree for the <code>path(X, c)</code> goal that the cut operator removes in the third <code>path</code> predicate.	695
15.1	The relationships between some of the concepts we studied.	715
15.2	Interplay of advanced concepts of programming languages.	717
C.1	A portion of the Haskell type class inheritance hierarchy.	786
D.1	The grammar in EBNF for the Camille programming language.	812

List of Tables

1	Mapping from the <i>ACM/IEEE Computing Curricula 2020</i> to Chapters of This Text	xxiii
2	Mapping from the <i>2013 ACM/IEEE Computing Curriculum Standards</i> to Chapters of This Text	xxiv
1.1	Static Vis-à-Vis Dynamic Bindings	7
1.2	Expressions Vis-à-Vis Statements	9
1.3	Purity in Programming Languages	15
1.4	Practical/Conceptual/Theoretical Basis for Common Styles of Programming	16
1.5	Key Terms Discussed in Section 1.4	16
2.1	Progressive Types of Sentence Validity	35
2.2	Progressive Types of Program Expression Validity	35
2.3	Regular Expressions	36
2.4	Examples of Regular Expression	37
2.5	Relationship of Regular Expressions, Regular Grammars, and Finite-State Automata to Regular Languages	42
2.6	Formal Grammars Vis-à-Vis BNF Grammars	49
2.7	The Dual Use of Grammars: For <i>Generation</i> (Constructing a Derivation) and <i>Recognition</i> (Constructing a Parse Tree)	52
2.8	Effect of Ambiguity on Semantics	52
2.9	Syntactic Ambiguity Vis-à-Vis Semantic Ambiguity	56
2.10	Polysemes, Homonyms, and Synonyms	56
2.11	Interplay Between and Interdependence of Types of Ambiguity	56
2.12	Formal Grammar Capabilities Vis-à-Vis Programming Language Constructs	68
2.13	Summary of Formal Languages and Grammars, and Models of Computation	69
3.1	Parceling Lexemes into Tokens in the Sentence <code>int i = 20; . . .</code>	72
3.2	Two-Dimensional Array Modeling a Finite-State Automaton.	75
3.3	(Concrete) Lexemes and Parse Trees Vis-à-Vis (Abstract) Tokens and Abstract-Syntax Trees, Respectively	75

3.4	Implementation Differences in Top-down Parsers: Table-Driven Vis-à-Vis Recursive-Descent	77
3.5	Top-down Vis-à-Vis Bottom-up Parsers	89
3.6	LL Vis-à-Vis LR Grammars	90
3.7	Parsing Programming Exercises in This Chapter, Including Their Essential Properties and Dependencies.	91
4.1	Advantages and Disadvantages of Compilers and Interpreters . . .	115
4.2	Interpretation Programming Exercises in This Chapter Annotated with the Prior Exercises on Which They Build.	117
4.3	Features of the Parsers Used in Each Subpart of the Programming Exercises in This Chapter.	117
5.1	Examples of Shortening <code>car-cdr</code> Call Chains with Syntactic Sugar	151
5.2	Binding Approaches Used in <code>let</code> and <code>let*</code> Expressions	157
5.3	Reducing <code>let</code> to <code>lambda</code>	161
5.4	Reducing <code>let*</code> to <code>lambda</code>	161
5.5	Reducing <code>letrec</code> to <code>lambda</code>	162
5.6	Semantics of <code>let</code> , <code>let*</code> , and <code>letrec</code>	163
5.7	Functional Programming Design Guidelines	181
6.1	Static Vis-à-Vis Dynamic Bindings	186
6.2	Static Scoping Vis-à-Vis Dynamic Scoping	188
6.3	Lexical Depth and Position in a Referencing Environment	194
6.4	Definitions of <i>Free</i> and <i>Bound</i> Variables in λ -Calculus	197
6.5	Advantages and Disadvantages of Static and Dynamic Scoping . . .	203
6.6	Example Data Structure Representation of Closures	216
6.7	Scoping Vis-à-Vis Environment Binding	238
7.1	Features of Type Systems Used in Programming Languages	248
7.2	The General Form of a <i>Qualified Type</i> or <i>Constrained Type</i> and an Example	257
7.3	Parametric Polymorphism Vis-à-Vis Function Overloading	259
7.4	Scheme Vis-à-Vis ML and Haskell for Fixed- and Variable-Sized Argument Lists	278
7.5	Scheme Vis-à-Vis ML and Haskell for Reception and Decomposi- tion of Argument(s)	278
8.1	Type Signatures and λ -Calculus for a Variety of Higher-Order Functions.	286
8.2	Definitions of <code>papply1</code> and <code>papply</code> in Scheme	287
8.3	Definitions of <code>curry</code> and <code>uncurry</code> in Curried Form in Haskell for Binary Functions	296
8.4	Definitions of <code>curry</code> and <code>uncurry</code> in Scheme for Binary Functions	297

9.1	Support for C/C++ Style <code>structs</code> and <code>unions</code> in ML, Haskell, Python, and Java	352
9.2	Support for Composition and Decomposition of Variant Records in a Variety of Programming Languages.	354
9.3	Summary of the Programming Exercises in This Chapter Involving the Implementation of a Variety of Representations for an Environment	375
9.4	The Variety of Representations of Environments in Racket Scheme and Python Developed in This Chapter	375
9.5	List-of-Lists/Vectors Representations of an Environment Used in Programming Exercise 9.8.4.	377
9.6	List-of-Lists Representations of an Environment Used in Programming Exercise 9.8.5.	379
9.7	Comparison of the Main Concepts and Features of ML and Haskell	384
10.1	New Versions of Camille, and Their Essential Properties, Created in the Chapter 10 Programming Exercises.	418
10.2	Versions of Camille.	420
10.3	Concepts and Features Implemented in Progressive Versions of Camille.	421
10.4	Configuration Options in Camille	421
11.1	New Versions of Camille, and Their Essential Properties, Created in the Section 11.2.4 Programming Exercises.	432
11.2	New Versions of Camille, and Their Essential Properties, Created in the Section 11.3.3 Programming Exercises.	447
11.3	Variety of Environments in Python Developed in This Text.	452
11.4	Camille Interpreters in Python Developed in This Text Using All Combinations of Non-recursive and Recursive Functions, and Named and Nameless Environments.	452
11.5	Versions of Camille.	454
11.6	Concepts and Features Implemented in Progressive Versions of Camille.	455
11.7	Configuration Options in Camille	456
12.1	New Versions of Camille, and Their Essential Properties, Created in the Programming Exercises of This Section.	465
12.2	Relationship Between Denoted Values, Dereferencing, and Parameter-Passing Mechanisms in Programming Languages Discussed in This Section.	481
12.3	Terms Used to Refer to Evaluation Strategies for Function Arguments in Three Progressive Contexts	494
12.4	Terms Used to Refer to Forming and Evaluating a Thunk	502
12.5	New Versions of Camille, and Their Essential Properties, Created in the Sections 12.6 and 12.7 Programming Exercises.	526
12.6	Complete Suite of Camille Languages and Interpreters.	535

12.7	Concepts and Features Implemented in Progressive Versions of Camille.	538
12.8	Complete Set of Configuration Options in Camille	539
12.9	Approaches to Learning Language Semantics Through Interpreter Implementation	539
13.1	Mapping from the Greatest Common Divisor Exercises in This Section to the Essential Aspects of First-Class Continuations and <code>call/cc</code>	566
13.2	Facilities for Global Transfer of Control in Scheme Vis-à-Vis C	572
13.3	Summary of Methods for Nonlocally Transferring Program Control	577
13.4	Mechanisms for Handling Exceptions in Programming Languages .	584
13.5	Levels of Data and Control Abstraction in Programming Languages	586
13.6	Different Sides of the Same Coin: Call-By-Name/Need Parameters, Continuations, and Coroutines Share Conceptually Common Complementary Operations	590
13.7	Non-tail Calls/Recursive Control Behavior Vis-à-Vis Tail Calls/Iterative Control Behavior	600
13.8	Summary of Higher-Order <code>fold</code> Functions with Respect to Eager and Lazy Evaluation	606
13.9	Properties of the Four Versions of <code>fact-cps</code> Presented in Section 13.8.2.	614
13.10	Interplay of Tail Recursion/Calls, Recursive/Iterative Control Behavior, Tail-Call Optimization, and Continuation-Passing Style .	614
13.11	Properties Present and Absent in the <code>call/cc</code> and CPS Versions of the <code>product</code> Function	616
13.12	Advantages and Disadvantages of Functions Exhibiting Recursive Control Behavior, Iterative Control Behavior, and Recursive Control Behavior with CPS Transformation	622
13.13	Mapping from the Greatest Common Divisor Exercises in This Section to the Essential Aspects of Continuation-Passing Style	627
13.14	The Approaches to Function Definition as Related to Control Presented in This Chapter Based on the Presence and Absence of a Variety of Desired Properties	637
13.15	Effects of the Techniques Discussed in This Chapter	640
14.1	Logical Concepts and Operators or Connectors	643
14.2	Truth Table Proof of the <i>Logical Equivalence</i> $p \supset q \equiv \neg p \vee q$. . .	643
14.3	Truth Table Illustration of the Concept of <i>Entailment</i> in $p \wedge q \models p \vee q$	644
14.4	<i>Quantifiers</i> in Predicate Calculus	646
14.5	The Commutative, Associative, and Distributive Rules of Boolean Algebra as Well as <i>DeMorgan's Laws</i> Are Helpful for Rewriting Propositions in CNF.	647
14.6	An Example Application of Resolution	650

14.7	An Example of a Resolution Proof by Refutation, Where the Propositions Therein Are Represented in CNF	652
14.8	Types of Horn Clauses with Forms and Examples	654
14.9	An Example Application of Resolution, Where the Propositions Therein Are Represented in Clausal Form	658
14.10	An Example of a Resolution Proof Using Backward Chaining	661
14.11	Mapping of Types of Horn Clauses to Prolog Clauses	662
14.12	Predicates for Interacting with the SWI-Prolog Shell (i.e., REPL) . .	664
14.13	A Comparison of Prolog and Datalog	672
14.14	Example List Patterns in Prolog Vis-à-Vis the Equivalent List Patterns in Haskell	672
14.15	Analogs Between a Relational Database Management System (RDBMS) and Prolog	686
14.16	Summary of the Mismatch Between Predicate Calculus and Prolog .	703
14.17	A Suite of Built-in Reflective Predicates in Prolog	704
14.18	Essential CLIPS Shell Commands	707
15.1	Reflection on Styles of Programming	714
C.1	Conceptual Equivalence in Type Mnemonics Between Java and Haskell	785
C.2	The General Form of a Qualified Type or Constrained Type and an Example	786
D.1	Configuration Options in Camille	815
D.2	Design Choices and Implemented Concepts in Progressive Versions of Camille	817
D.3	Solutions to the Camille Interpreter Programming Exercises in Chapters 10–12	820

PART I

FUNDAMENTALS

Chapter 1

Introduction

Language to the mind is more than light is to the eye.

— Anne Sullivan in William Gibson’s *The Miracle Worker* (1959)

That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted.

— George Boole (1854)

A language that doesn’t affect the way you think about programming, is not worth knowing.

— Alan Perlis (1982)

“I don’t see how he can ever finish, if he doesn’t begin.”

— Alice, in *Alice’s Adventures in Wonderland* (1895) by Lewis Carroll

WELCOME to the study of programming languages. This book and course of study is about *programming language concepts*—the building blocks of languages.

1.1 Text Objectives

The objectives of this text are to:

- Establish an understanding of fundamental and universal language concepts and design/implementation options for them.
- Improve readers’ ability to understand new programming languages and enhance their background for selecting appropriate languages.
- Expose readers to alternative styles of programming and exotic ways of performing computation so to establish an increased capacity for describing computation in a program, a richer toolbox of techniques from which to solve problems, and a more holistic picture of computing.

Since language concepts are the building blocks from which all languages are constructed and organized, an understanding of the concepts implies that, given a (new) language, one can:

- Deconstruct it into its essential concepts and determine the implementation options for these concepts.
- Focus on the big picture (i.e., core concepts/features and options) and not language nuisances or minutia (e.g., syntax).
- Discern in which contexts (e.g., application domains) it is an appropriate or ideal language of choice.
- In turn, learn to use, assimilate, and harness the strengths of the language more quickly.

1.2 Chapter Objectives

- Establish a foundation for the study of concepts of programming languages.
- Introduce a variety of styles of programming.
- Establish the historical context in which programming languages evolved.
- Establish an understanding of the factors that influence language design and development and how those factors have changed over time.
- Establish objectives and learning outcomes for the study of programming languages.

1.3 The World of Programming Languages

1.3.1 Fundamental Questions

This text is about programming language concepts. In preparation for a study of language concepts, we must examine some fundamental questions:

- What is a language (not necessarily a programming language)? A *language* is simply a medium of communication (e.g., a whale's song).
- What is a program? A *program* is a set of instructions that a computer understands and follows.
- What is a programming language? A *programming language* is a system of data-manipulation rules for *describing computation*.
- What is a *programming language concept*? It is best defined by example. Perhaps the language concept that resonates most keenly with readers at this point in their formal study of computer science is that of parameter passing. Some languages implement parameter passing with pass-by-value, while others use pass-by-reference, and still other languages implement both mechanisms. In a general sense, a language concept is typically a universal principle of languages, for which individual languages differ in their implementation approach to that principle. The way a concept is implemented in a particular language helps define the semantics of the

language. In this text, we will demonstrate a variety of language concepts and implement some of them.

- What influences language design? How did programming languages evolve and why? Which factors form the basis for programming languages' evolution: industrial/commercial problems, hardware capabilities/limitations, or the abilities of programmers?

Since a programming language is a system for describing computation, a natural question arises: What exactly is the computation that a programming language describes? While this question is studied formally in a course on computability theory, some brief remarks will be helpful here. The notion of mechanical *computation* (or an algorithm) is formally defined by the abstract mathematical model of a computer called a *Turing machine*. A Turing machine is a universal computing model that establishes the notion of what is computable. A programming language is referred to as *Turing-complete* if it can describe any computational process that can be described by a Turing machine. The notion of *Turing-completeness* is a way to establish the power of a programming language in describing computation: If the language can describe all of the computations that a Turing machine can carry out, then the language is Turing-complete. Support for sequential execution of both *variable assignment* and *conditional-branching* statements (e.g., *if* and *while*, and *if* and *goto*) is sufficient to describe computation that a Turing machine can perform. Thus, a programming language with those facilities is considered Turing-complete.

Most, but not all, programming languages are Turing-complete. In consequence, the more interesting and relevant question as it relates to this course of study is not what is or is not formally computable through use of a particular language, but rather which types of programming abstractions are or are not available in the language for describing computation in a more practical sense. Larry Wall, who developed Perl, said:

Computer languages differ not so much in what they make possible, but in what they make easy. (Christiansen, Foy, Wall, and Orwant, 2012, p. xxiii)

"Languages are *abstractions*: ways of seeing or organizing the world according to certain patterns, so that a task becomes easier to carry out. ... [For instance, a] loop is an abstraction: a reusable pattern" (Krishnamurthi 2003, p. 315). Furthermore, programming languages affect (or should affect) the way we think about describing ideas about computation. Alan Perlis (1982) said: "A language that doesn't affect the way you think about programming, is not worth knowing" (Epigraph 19, p. 8). In psychology, it is widely believed that one's capacity to think is limited by the language through which one communicates one's thoughts. This belief is known as the *Sapir-Whorf hypothesis*. George Boole (1854) said: "Language is an instrument of human reason, and not merely a medium for the expression of thought[; it] is a truth generally admitted" (p. 24). As we will see, some programming idioms cannot be expressed as easily or at all in certain languages as they can in others.

A universal lexicon has been established for discussing the concepts of languages and we must understand some of these fundamental/universal terms for engaging in this course of study. We encounter these terms throughout this chapter.

1.3.2 Bindings: Static and Dynamic

Bindings are central to the study of programming languages. *Bindings* refer to the association of one aspect of a program or programming language with another. For instance, in C the reserved word `int` is a mnemonic bound to mean “integer” by the language designer. A programmer who declares `x` to be of type `int` in a program (i.e., `int x;`) binds the identifier `x` to be of type integer. A program containing the statement `x = 1;` binds the value 1 to the variable represented by the identifier `x`, and 1 is referred to as the *denotation* of `x`. Bindings happen at particular times, called *binding times*. Six progressive binding times are identified in the study of programming languages:

1. *Language definition time* (e.g., the keyword `int` bound to the meaning of integer)
2. *Language implementation time* (e.g., `int` data type bound to a storage size such as four bytes)
3. *Compile time* (e.g., identifier `x` bound to an integer variable)
4. *Link time* (e.g., `printf` is bound to a definition from a library of routines)
5. *Load time* (e.g., variable `x` bound to memory cell at address `0x7cd7`—can happen at run-time as well; consider a variable local to a function)

↑ *static bindings* ↑

↓ *dynamic bindings* ↓

6. *Run-time* (e.g., `x` bound to value 1)

Language definition time involves defining the *syntax* (i.e., form) and *semantics* (i.e., meaning) of a programming language. (Language definition and description methods are the primary topic of Chapter 2.) Language implementation time is the time at which a compiler or interpreter for the language is built. (Building language interpreters is the focus of Chapters 10–12.) At this time some of the semantics of the implemented language are bound/defined as well. The examples given in the preceding list are not always performed at the particular time in which they are classified. For instance, binding the variable `x` to the memory cell at address `0x7cd7` can also happen at run-time in cases where `x` is a variable local to a function or block.

The aforementioned bindings are often broadly categorized as either static or dynamic (Table 1.1). A *static* binding happens before run-time (usually at compile time) and often remains unchangeable during run-time. A *dynamic* binding happens at run-time and can be changed at run-time. Dynamic binding is also

Static	bindings occur <i>before</i> run-time and are <i>fixed</i> during run-time.
Dynamic	bindings occur <i>at</i> run-time and are <i>changeable</i> during run-time.

Table 1.1 Static Vis-à-Vis Dynamic Bindings

referred to as *late binding*. It is helpful to think of an analogy to human beings. Our date of birth is bound statically at birth and cannot change throughout our life. Our height, in contrast, is (re-)bound dynamically—it changes throughout our life. Earlier times imply safety, reliability, predictability (i.e., no surprises at run-time), and efficiency. Later times imply flexibility. In interpreted languages, such as Scheme, most bindings are dynamic. Conversely, most bindings are static in compiled languages such as C, C++, and Fortran. Given the central role of bindings in the study of programming languages, we examine both the types of bindings (i.e., what is being bound to what) as well as the binding times involved in the language concepts we encounter in our progression through this text, particularly in Chapter 6.

1.3.3 Programming Language Concepts

Let us demonstrate some language concepts by example, and observe that they often involve options. You may recognize some of the following language concepts (though you may not have thought of them as language concepts) from your study of computing:

- language implementation (e.g., interpreted or compiled)
- parameter passing (e.g., by-value or by-reference)
- abstraction (e.g., procedural or data)
- typing (e.g., static or dynamic)
- scope (e.g., static or dynamic)

We can draw an analogy between language concepts and automobile concepts. Automobile concepts include make (e.g., Honda or Toyota), model (e.g., Accord or Camry), engine type (e.g., gasoline, diesel, hybrid, or electric), transmission type (e.g., manual or automatic), drivetrain (e.g., front wheel, rear wheel, or all wheel), and options (e.g., rear camera, sensors, Bluetooth, satellite radio, and GPS navigation). With certain concepts of languages, their options are so ingrained into the fiber of computing that we rarely ever consider alternative options. For instance, most languages provide facilities for procedural and data abstraction. However, most languages do not provide (sophisticated) facilities for control abstraction (i.e., developing new control structures). The traditional *if*, *while*, and *for* are not the only control constructs for programming. Although some languages, including Go and C++, provide a *goto* statement for transfer of control, a *goto* statement is not sufficiently powerful to design new control structures. (Control abstraction is the topic of Chapter 13.)

The options for language concepts are rarely binary or discretely defined. For instance, multiple types of parameter passing are possible. The options available

and the granularity of those options often vary from language to language and depend on factors such as the application domain targeted by the language and the particular problem to be solved. Some concepts, including control abstraction, are omitted in certain languages.

Beyond these fundamental/universal language concepts, an exploration of a variety of programming styles and language support for these styles leads to a host of other important principles of programming languages and language constructs/abstractions (e.g., closures, higher-order functions, currying, and first-class continuations).

1.4 Styles of Programming

We use the term “*styles of programming*” rather than perhaps the more common/conventional, but antiquated, term “*paradigm of programming*.” See Section 1.4.6 for an explanation.

1.4.1 Imperative Programming

The primary method of describing/affecting computation in an *imperative style of programming* is through the execution of a sequence of commands or *imperatives* that use assignment to modify the values of variables—which are themselves abstractions of memory cells. In C and Fortran, for example, the primary mode of programming is imperative in nature. The imperative style of programming is a natural consequence of basing a computer on the *von Neumann architecture*, which is defined by its uniform representation of both instructions and data in main memory and its use of a fetch–decode–execute cycle. (While the *Turing machine* is an abstract model that captures the notion of mechanical computation, the von Neumann architecture is a practical design model for actual computers. The concept of a Turing machine was developed in 1935–1937 by Alan Turing and published in 1937. The von Neumann architecture was articulated by John von Neumann in 1945.)

The main mechanism used to effect computation in the imperative style is the assignment operator. A discussion of the difference between *statements* and *expressions* in programs helps illustrate alternative ways to perform such computation. Expressions are evaluated for their value, which is returned to the next encompassing expression. For instance, the subexpression $(3 * 4)$ in the expression $2 + (3 * 4)$ returns the integer 12, which becomes the second operand to the addition operator. In contrast, the statement $i = i + 1$ has no return value.¹ After that statement is executed, evaluation proceeds with the following statement (i.e., sequential execution). *Expressions* are evaluated for *values* while *statements* are executed for *side effect* (Table 1.2). A *side effect* is a modification of a parameter to a function or operator, or an entity in the external environment (e.g., a change to a global variable or performing I/O, which changes the nature of the input

1. In C, such statements return the value of i after the assignment takes place.

Expressions are evaluated for *value*.
Statements are executed for *side effect*.

Table 1.2 Expressions Vis-à-Vis Statements

stream/file). The primary way to perform computation in an imperative style of programming is through side effect. The assignment statement inherently involves a side effect. For instance, the execution of statement `x = 1` changes the first parameter (i.e., `x`) to the `=` assignment operator to 1. I/O also inherently involves a side effect. For instance, consider the following Python program:

```
x = int(input())
print (x + x)
```

If the input stream contains the integer 1 followed by the integer 2, readers accustomed to imperative programming might predict the output of this program to be 2 because the `input` function executes only once, reads the value 1,² and stores it in the variable `x`. However, one might interpret the line `print (x + x)` as `print (int(input()) + int(input()))`, since `x` stands for `int(input())`. With this interpretation, one might predict the output of the program to be 3, where the first and second invocations to `input()` read 1 and 2, respectively. While mathematics involves binding (e.g., *let x = 1 in ...*), mathematics does not involve assignment.³

The aforementioned interpretation of the statement `print (x + x)` as `print (int(input()) + int(input()))` might seem unnatural to most readers. For those readers who are largely familiar with the imperative style of programming, describing computation through side effect is so fundamental to and ingrained into their view of programming and so unconsciously integrated into their programming activities that the prior interpretation is viewed as entirely foreign. However, that interpretation might seem entirely natural to a mathematician or someone who has no experience with programming.

Side effects also make a program difficult to understand. For instance, consider the following Python program:

```
def f():
    global x
    x = 2
    return x
# main program
x = 1
print (x + f())
```

Function `f` has a side effect: After `f` is called, the `global` variable `x` has value 2, which is different than the value it had prior to the call to `f`. As a result, the output of this program depends on the order in which the operands to the

2. The Python `int` function used here converts the string read with the `input` function to an integer.

3. The common programming idiom `x=x+1` can be confusing to nonprogrammers because it appears to convey that two entities are equal that are clearly not equal.

addition operator are evaluated. However, the result of a commutative operation, like addition, is not dependent on the order in which its operands are evaluated (i.e., $1 + 2 = 2 + 1 = 3$). If the operands are evaluated from left to right (i.e., Python semantics), the output of this program is 3. If the operands are evaluated from right to left, the output is 4.

The concept of side-effect is closely related to, yet distinct from, the concept of *referential transparency*. Expressions and languages are said to be *referentially transparent* (i.e., independent of evaluation order) if the same arguments/operands to a function/operator yield the same output irrespective of the context/environment in which the expression applying the function/operator is evaluated. The function Python `⊕` given previously has a side effect and the expression `x + ⊕()` is not referential transparent. The absence of side effects is not sufficient to guarantee referential transparency (Conceptual Exercise 1.8).

Since the von Neumann architecture gave rise to an imperative mode of programming, most early programming languages (e.g., Fortran and COBOL), save for Lisp, supported primarily that style of programming. Moreover, programming languages evolved based on the von Neumann model. However, the von Neumann architecture has certain inherent limitations. Since a processor can execute program instructions much faster than program instructions and program data can be moved from main memory to the processor, I/O between the processor and memory—referred to as the *von Neumann bottleneck*—affects the speed of program execution. Moreover, the reality that computation must be described as a sequence of instructions operating on a single piece of data that is central to the von Neumann architecture creates another limitation. The von Neumann architecture is not a natural model for other non-imperative styles of describing computation. For instance, recursion, nondeterministic computation, and parallel computation do not align with the von Neumann model.^{4,5}

Imperative programming is programming by side effect; functional programming is programming without side effect. *Functional programming* involves describing and performing computation by calling functions that return values. Programmers from an imperative background may find it challenging to conceive of writing a program without variables and assignment statements. Not only is such a mode of programming possible, but it leads to a compelling higher-order style of program construction, where functions accept other functions as arguments and can return a function as a return value. As a result, a program is conceived as a collection of highly general, abstract, and reusable functions that build other functions, which collectively solve the problem at hand.

4. Ironically, John Backus, the recipient of the 1977 ACM A. M. Turing Award for contributions to the primarily imperative programming language Fortran, titled his Turing Award paper “Can Programming Be Liberated from the von Neumann Style?: A Functional Style and Its Algebra of Programs.” This paper introduced the functional programming language FP through which Backus (1978) cast his argument. While FP was never fully embraced by the industrial programming community, it ignited both debate and interest in functional programming and subsequently influenced multiple languages supporting a functional style of programming (*Interview with Simon Peyton-Jones* 2017).

5. Computers have been designed for these inherently non-imperative styles as well (e.g., Lisp machine and Warren Abstract Machine).

1.4.2 Functional Programming

While the essential element in imperative programming is the assignment statement, the essential ingredient in functional programming is the function. Functions in languages supporting a functional style of programming are first-class entities. In programming languages, a *first-class entity* is a program object that has privileges that other comparable program entities do not have.⁶ The designation of a language entity as first-class generally means that the entity can be expressed in the source code of the program and has a value at run-time that can be manipulated programmatically (i.e., within the source code of the program). Traditionally, this has meant that a first-class entity can be stored (e.g., in a variable or data structure), passed as an argument, and returned as a value. For instance, in many modern programming languages, functions are first-class entities because they can be created and manipulated at run-time through the source code. Conversely, labels in C passed to `goto` do not have run-time values and, therefore, are not first-class entities. Similarly, a class in Java does not have a manipulatable value at run-time and is not a first-class entity. In contrast, a class in Smalltalk does have a value that can be manipulated at run-time, so it is a first-class entity.

In a functional style of programming, the programmer describes computation primarily by calling a series of functions that cascade a set of return values to each other. Functional programming typically does not involve variables and assignment, so side effects are absent from programs developed using a functional style. Since side effect is fundamental to sequential execution, statement blocks, and iteration, a functional style of programming utilizes recursion as a primary means of repetition. The functional style of programming was pioneered in the Lisp programming language, designed by John McCarthy in 1958 at MIT (1960). Scheme and Common Lisp are dialects of Lisp. Scheme, in particular, is an ideal vehicle for exploring language semantics and implementing language concepts. For instance, we use Scheme in this text to implement recursion from first principles, as well as a variety of other language concepts. In contrast to the von Neumann architecture, the Lisp *machine* is a predecessor to modern single-user workstations. ML, Haskell, and F# also primarily support a functional style of programming.

Functional programming is based on *lambda-calculus* (hereafter referred to as λ -calculus)—a mathematical theory of functions developed in 1928–1929 by Alonzo Church and published in 1932.⁷ Like the Turing machine, λ -calculus is an abstract mathematical model capturing the notion of mechanical computation (or an algorithm). Every function that is *computable*—referred to as *decidable*—by Turing machines is also computable in (untyped) λ -calculus. One goal of functional programming is to bring the activity of programming closer to mathematics, especially to formally guarantee certain safety properties and constraints. While the criterion of sequential execution of assignment and conditional statements is sufficient to determine whether a language is Turing-complete, languages without support for sequential execution and variable assignment can also be

6. Sometimes entities in programming languages are referred to as *second-class* or even *third-class* entities. However, these distinctions are generally not helpful.

7. Alonzo Church was Alan Turing's PhD advisor at Princeton University from 1936 to 1938.

Turing-complete. Support for (1) arithmetic operations on integer values, (2) a selection operator (e.g., `if ... then ... else ...`), and (3) the ability to define new recursive functions from existing functions/operators are alternative and sufficient criteria to describe the computation that a Turing machine can perform. Thus, a programming language with those facilities is also Turing-complete.

The concept of *purity* in programming languages also arises with respect to programming style. A language without support for side effect, including no side effect for I/O, can be considered to support a pure form of functional programming. Scheme is not pure in its support for functional programming because it has an assignment operator and I/O operators. By comparison, Haskell is nearly pure. Haskell has no support for variables or assignment, but it supports I/O in a carefully controlled way through the use of *monads*, which are functions that have side effects but cannot be called by functions without side effects.

Again, programming without variables or assignment may seem inconceivable to some programmers, or at least seem to be an ascetical discipline. However, modification of the value of a variable through assignment accounts for a large volume of bugs in programs. Thus, without facilities for assignment one might write less buggy code. “Ericsson’s AXD301 project, a couple million lines of Erlang code,⁸ has achieved 99.999999% reliability. How? ‘No shared state and a sophisticated error-recovery model,’ Joe [Armstrong, who was a designer of Erlang] says” (Swaine 2009, p. 16). Moreover, parallelization and synchronization of single-threaded programs is easier in the absence of variables whose values change over time since there is no shared state to protect from corruption. Chapter 5 introduces the details of the functional style of programming. The imperative and functional modes of programming are not entirely mutually exclusive, as we see in Section 1.4.6.

1.4.3 Object-Oriented Programming

In object-oriented programming, a programmer develops a solution to a problem as a collection of *objects* communicating by passing *messages* to each other (Figure 1.1):

I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning—it took a while to see how to do messaging in a programming language efficiently enough to be useful). (Kay 2003)

Objects are program entities that encapsulate data and functionality. An object-oriented style of programming typically unifies the concepts of data and procedural abstraction through the constructs of classes and objects. The object-oriented style of programming was pioneered in the Smalltalk programming language, designed by Alan Kay and colleagues in the early 1970s at Xerox PARC.

⁸ Erlang is a language supporting concurrent and functional programming that was developed by the telecommunications company Ericsson.

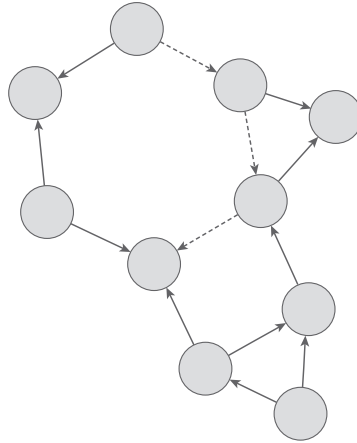


Figure 1.1 Conceptual depiction of a set of objects communicating by passing messages to each other to collaboratively solve a problem.

While there are imperative aspects involved in object-oriented programming (e.g., assignment), the concept of a *closure* from functional programming (i.e., a first-class function with associated bindings) is an early precursor to an *object* (i.e., a program entity encapsulating behavior and state). Alan Kay (2003) has expressed that Lisp influenced his thoughts in the development of object orientation and Smalltalk. Languages supporting an object-oriented style of programming include Java, C++, and C#. A language supporting a pure style of object-oriented programming is one where all program entities are objects—including primitives, classes, and methods—and where all computation is described by passing messages between these objects. Smalltalk and languages based on the Common Lisp Object System (CLOS), including Dylan, support a pure form of object-oriented programming.

Lisp (and the Lisp machine) and Smalltalk were the experimental platforms that gave birth to many of the commonly used and contemporary language features, including implicit pointer dereferencing, automatic garbage collection, run-type typing, and associated tools (e.g., interactive programming environments and pointing devices such as the mouse). Both languages significantly influenced the subsequent evolution of programming languages and, indeed, personal computing. Lisp, in particular, played an influential role in the development of other important programming languages, including Smalltalk (Kay 2003).

1.4.4 Logic/Declarative Programming

The defining characteristic of a *logic* or *declarative* style of programming is description of *what* is to be computed, not *how* to compute it. Thus, declarative programming is largely an activity of specification, and languages supporting declarative programming are sometimes called *very-high-level languages* or *fifth-generation languages*. Languages supporting a logic/declarative style of programming have support for reasoning about facts and rules; consequently,

this style of programming is sometimes referred to as *rule-based*. The basis of the logic/declarative style of programming is *first-order predicate calculus*.

Prolog is a language supporting a logic/declarative style of programming. In contrast to the von Neumann architecture, the *Warren Abstract Machine* is a target platform for Prolog compilers. CLIPS is also a language supporting logic/declarative programming. Likewise, programming in SQL is predominantly done in a declarative manner. A SQL query describes what data is desired, not how to find that data (i.e., developing a plan to answer the query). Usually language support for declarative programming implies an inefficient language implementation since declarative specification occurs at a very high level. In turn, interpreters for languages that support declarative programming typically involve multiple layers of abstraction.

An objective of logic/declarative programming is to support the specification of both what you want and the knowledge base (i.e., the facts and rules) from which what you want is to be inferred without regard to how the system will deduce the result. In other words, the programmer should not be required or permitted to codify the facts and rules in the program in a form that imparts control over or manipulates the built-in deduction algorithm for producing the desired result. No control information or procedural directives should be woven into the knowledge base so to direct the interpreter's deduction process. Specification (or declaration) should be order-independent. Consider the following two logical propositions:

If it is raining and windy, I carry an umbrella. $(R \wedge W) \supset U$
 If it is windy and raining, I carry an umbrella. $(W \wedge R) \supset U$

Since the conjunction logical operator ($\wedge$) is commutative, these two propositions are semantically equivalent and, thus, it should not matter which of the two forms we use in a program. However, since computers are deterministic systems, the interpreter for a language supporting declarative programming typically evaluates the terms on the left-hand side of these propositions (i.e., R and W) in a left-to-right or right-to-left order. Thus, the desired result of the program can—due to side effect and other factors—depend on that evaluation order, akin to the evaluation order of the terms in the Python expression $x + f()$ described earlier. Languages supporting logic/declarative programming as the primary mode of performing computation often equip the programmer with facilities to impart control over the search strategy used by the system (e.g., the cut operator in Prolog). These control facilities violate a defining principle of a declarative style—that is, the programmer need only be concerned with the logic and can leave the control (i.e., the inference methods used to produce program output) up to the system. Unlike Prolog, the Mercury programming language is nearly pure in its support for declarative programming because it does not support control facilities intended to circumvent or direct the search strategy built into the system (Somogyi, Henderson, and Conway 1996). Moreover, the form of the specification of the facts and rules in a logic/declarative program should have no bearing on the output of the program. Unfortunately, it often does. Mercury is the closest to a language

Style of Programming	Purity Indicates	(Near-)Pure Language(s)
Functional programming	No provision for side effect	Haskell
Logic/declarative programming	No provision for control	Mercury
Object-oriented programming	No provision for performing computation without message passing; all program entities are objects	Smalltalk, Ruby, and CLOS-based languages

Table 1.3 Purity in Programming Languages

supporting a purely logic/declarative style of programming. Table 1.3 summarizes purity in programming styles. Chapter 14 discusses the logic/declarative style of programming.

1.4.5 Bottom-up Programming

A compelling style of programming is to use a programming language not to develop a solution to a problem, but rather to build a language specifically tailored to solving a family of problems for which the problem at hand is an instance. The programmer subsequently uses this language to write a program to solve the problem of interest. This process is called *bottom-up programming* and the resulting language is typically either an *embedded* or a *domain-specific language*. Bottom-up programming is not on the same conceptual level as the other styles of programming discussed in this chapter—it is on more of a *meta-level*. Similarly, Lisp is not just a programming language or a language supporting multiple styles of programming. From its origin, Lisp was designed as a language to be extended (Graham 1993, p. vi), or “a programmable programming language” (Foderaro 1991, p. 27), on which the programmer can build layers of languages supporting multiple styles of programming. For instance, the abstractions in Lisp can be used to extend the language with support for object-oriented programming (Graham 1993, p. ix). This style of programming or metaprogramming, called bottom-up programming, involves using a programming language not as a tool to write a target program, but to define a new targeted (or domain-specific) language and then develop the target program in that language (Graham 1993, p. vi). In other words, bottom-up programming involves “changing the language to suit the problem” (Graham 1993, p. 3). “Not only can you program *in* Lisp (that makes it a programming language) but you can program the language itself” (Foderaro 1991, p. 27). It has been said that “[i]f you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases” (Friedman and Felleisen 1996b, p. 207).

Style of Programming	Practical/Conceptual/Theoretical Foundation	Defining/Pioneering Language
Imperative programming	von Neumann architecture	Fortran
Functional programming	λ -calculus; Lisp machine	Lisp
Logic/declarative programming	First-order Predicate Calculus; Warren Abstract Machine	Prolog
Object-oriented programming	Lisp, biological cells, individual computers on a network	Smalltalk

Table 1.4 Practical/Conceptual/Theoretical Basis for Common Styles of Programming

syntax: <i>form</i> of language semantics: <i>meaning</i> of language first-class entity side effect referential transparency

Table 1.5 Key Terms Discussed in Section 1.4

Other programming languages are also intended to be used for bottom-up programming (e.g., Arc⁹). While we do return to the idea of bottom-up programming in Section 5.12 in Chapter 5, and in Chapter 15, the details of bottom-up programming are beyond the scope of this text. For now it suffices to say that bottom-up design can be thought of as building a library of functions followed by writing a concise program that calls those functions. “However, Lisp gives you much broader powers in this department, and augmenting the language plays a proportionately larger role in Lisp style—so much so that [as mentioned previously] Lisp is not just a different language, but a whole different way of programming” (Graham 1993, p. 4).

A host of other styles of programming are supported by a variety of other languages: concatenative programming (e.g., Factor, Joy) and dataflow programming (e.g., LabView). Table 1.4 summarizes the origins of the styles of programming introduced here. Table 1.5 presents the terms introduced in this section that are fundamental/universal to the study of programming languages.

1.4.6 Synthesis: Beyond Paradigms

Most languages have support for imperative (e.g., assignment, statement blocks), object-oriented (e.g., objects, classes), and functional (e.g., λ /anonymous [and

9. <http://arclanguage.org>

first-class] functions) programming. Some languages even have, to a lesser extent, support for declarative programming (e.g., pattern-directed invocation).

What we refer to here as *styles of programming* was once—and in many cases still is—referred to as *paradigms of languages*.¹⁰ Imperative, functional, logic/declarative, and object-oriented have been, traditionally, the four classical paradigms of languages. However, historically, other paradigms have emerged for niche application domains,¹¹ including languages for business applications (e.g., COBOL), *hardware description languages* (e.g., Verilog, VHDL), and *scripting languages* (e.g., awk, Rexx, Tcl, Perl). Traditional scripting languages are typically interpreted languages supporting an imperative style of programming with an easy-to-use command-and-control-oriented syntax and ideal for processing strings and generating reports. The advent of the web ignited the evolution of languages used for traditional scripting-type tasks into languages supporting multiple styles of programming (e.g., JavaScript, Python, Ruby, PHP, and Tcl/Tk). As the web and its use continued to evolve, the programming tasks common to web programming drove these languages to continue to grow and incorporate additional features and constructs supporting more expressive and advanced forms of functional, object-oriented, and concurrent programming. (Use of these languages with associated development patterns [e.g., Model-View-Controller] eventually evolved into *web frameworks* [e.g., Express, Django Rails, Lavavel].)

The styles of programming just discussed are not mutually exclusive, and language support for multiple styles is not limited to those languages used solely for web applications. Indeed, one can write a program with a functional motif while sparingly using imperative constructs (e.g., assignment) for purposes of pragmatics. Scheme and ML primarily support a functional style of programming, but have some imperative features (e.g., assignment statements and statement blocks). Alternatively, one can write a primarily imperative program using some functional constructs (e.g., λ /anonymous functions). Dylan, which was influenced by Scheme and Common Lisp, is a language that adds support for object-oriented programming to its functional programming roots. Similarly, the *pattern-directed invocation* built into languages such as ML and Haskell is declarative in nature and resembles the rule-based programming, at least syntactically, in Prolog. Curry is a programming language derived from Haskell and, therefore, supports functional programming; however, it also includes support for logic programming. In contrast, POP-11 primarily facilitates a declarative style of programming, but

10. A *paradigm* is a worldview—a model. A *model* is a simplified view of some entity in the real world (e.g., a model airplane) that is simpler to interact with. A *programming language paradigm* refers to a style of performing computation from which programming in a language adhering to the tenets of that style proceeds. A language paradigm can be thought of as a family of natural languages, such as the Romance languages or the Germanic languages.

11. In the past, even the classical functional and logic/declarative paradigms, and specifically the languages Lisp and Prolog, respectively, were considered paradigms primarily for artificial intelligence applications even though the *emacs* text editor for UNIX and *Autocad* are two non-AI applications that are more than 30 years old and were developed in Lisp. Now there are Lisp and Prolog applications in a variety of other domains (e.g., *Orbitz*). We refer the reader to Graham (1993, p. 1) for the details of the origin of the (accidental) association between Lisp and AI. Nevertheless, certain languages are still ideally suited to solve problems in a particular niche application domain. For instance, C is a language for systems programming and continues to be the language of choice for building operating systems.

supports first-class functions. Scala is a language with support for functional programming that runs on the Java virtual machine.

Moreover, some languages support database connectivity to make (declaratively written) queries to a database system. For instance, C# supports “Language-INtegrated Queries” (LINQ), where a programmer can embed SQL-inspired declarative code into programs that otherwise use a combination of imperative, functional, object-oriented, and concurrent programming constructs. Despite this phenomenon in language evolution, both the concept and use of the term *paradigm* as well as the classical boundaries were still rigorously retained. These languages are referred to as either *web programming languages* (i.e., a new paradigm was invented) or *multi-paradigm languages*—an explicit indication of the support for multiple paradigms needed to maintain the classical paradigms.

Almost no languages support only one style of programming. Even Fortran and BASIC, which were conceived as imperative programming languages, now incorporate object-oriented features. Moreover, Smalltalk, which supports a pure form of object-oriented programming, has support for closures from functional programming—though, of course, they are accessed and manipulated through object orientation and message passing. Similarly, Mercury, which is considered nearly a pure logic/declarative language, also supports functional programming. For example, while based on Prolog, Mercury marries Prolog with the Haskell type system (Somogyi, Henderson, and Conway 1996). Conversely, almost all languages support some form of concurrent programming—an indication of the influence of multicore processors on language evolution (Section 1.5). Moreover, many languages now support some form of λ /anonymous functions. Languages supporting more than one style of programming are now the norm; languages supporting only one style of programming are now the exception.¹²

Perhaps this is partial acknowledgment from the industry that concepts from functional (e.g., first-class functions) and object-oriented programming (e.g., reflection) are finding their way from research languages into mainstream languages (see Figure 1.4 and Section 1.5 later in this chapter). It also calls the necessity of the concept of *language paradigm* into question. If all languages are multi-paradigm languages, then the concept of language paradigm is antiquated. Thus, the boundaries of the classical (and contemporary) paradigms are by now thoroughly blurred, rendering both the boundaries and the paradigms themselves irrelevant: “Programming language ‘paradigms’ are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them?” (Krishnamurthi 2008). The terms originally identifying language paradigms (e.g., imperative, object-oriented, functional, and declarative) are more *styles of programming*^{13,14} than descriptors for languages or patterns for languages to follow. Thus, instead of talking about

12. The miniKanren family of languages primarily supports logic programming.

13. John Backus (1978) used the phrase “functional style” in the title of his 1977 Turing Award paper.

14. When we use the phrase “styles of programming” we are not referring to the program formatting guidelines that are often referred to as “program style” (e.g., consistent use of three spaces for indentation or placing the function return type on a separate line) (Kernighan and Plauger 1978), but rather the style of effecting and describing computation.

a “functional language” or an “object-oriented language,” we discuss “functional programming” and “object-oriented programming.”

A style of programming captures the concepts and constructs through which a language provides support for effecting and describing computation (e.g., by assignment and side effect vis-à-vis by functions and return values) and is not a property of a language. The essence of the differences between styles of programming is captured by how *computation* is fundamentally effected and described in each style.¹⁵

1.4.7 Language Evaluation Criteria

As a result of the support for multiple styles of programming in a single language, now, as opposed to 30 years ago, a comparative analysis of languages cannot be fostered using the styles (i.e., “paradigms”) themselves. For instance, since Python and Go support multiple overlapping styles of programming, a comparison of them is not as simple as stating, “Python is an object-oriented language and Go is an imperative language.” Despite their support for a variety of programming styles, all computer languages involve a core set of universal concepts (Figure 1.2), so concepts of languages provide the basis for undertaking comparative analysis. Programming languages differ in terms of the implementation options each employs for these concepts. For instance, Python is a dynamically typed language and Go is a statically typed language. The construction of an interpreter for a computer language operationalizes (or instantiates) the design options or semantics for the pertinent concepts. (*Operational semantics* supplies the meaning of a computer program through its implementation.) One objective of this text is to provide the framework in which to study, compare, and select from the available programming languages.

There are other criteria—sometimes called *nonfunctional requirements*—by which to evaluate languages. Traditionally, these criteria include readability, writability, reliability (i.e., safety), and cost. For instance, all of the parentheses in Lisp affect the readability and writability of Lisp programs.¹⁶ Others might argue that the verbose nature of COBOL makes it a readable language (e.g., `ADD 1 TO X GIVING Y`), but not a writable language. How are readability and writability related? In the case of COBOL, they are inversely proportional to each other. Some criteria are subject to interpretation. For instance, cost (i.e., efficiency) can refer to the cost of execution or the cost of development. Other language evaluation criteria include portability, usability, security, maintainability, modifiability, and manageability.

Languages can be also be compared on the basis of their implementations. Historically, languages that primarily supported imperative programming

15. For instance, the *object-relational impedance mismatch* between relational database systems (e.g., PostgreSQL or MySQL) and languages supporting object-oriented programming—which refers to the challenge in mapping relational schemas and database tables (which are set-, bag-, or list-oriented) in a relational database system to class definitions and objects—is more a reflection of differing levels of granularity in the various data modeling support structures than one fundamental to describing computation.

16. Some have stated that Lisp stands for **Lisp Is Superfluous Parentheses**.

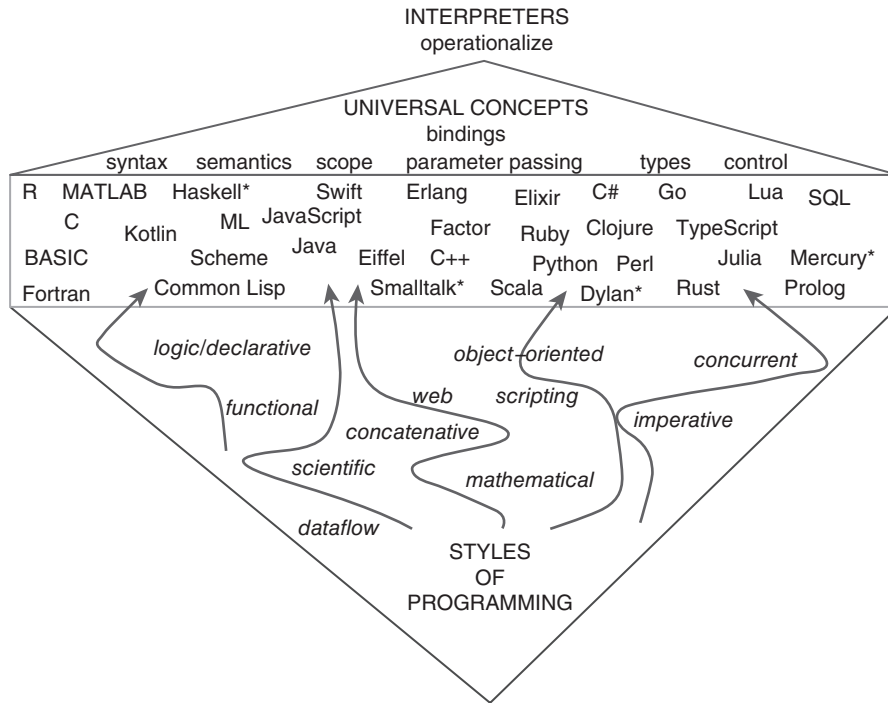


Figure 1.2 Within the context of their support for a variety of programming styles, all languages involve a core set of universal concepts that are operationalized through an interpreter and provide a basis for (comparative) evaluation. Asterisks indicate (near-)purity with respect to programming style.

involved mostly static bindings and, therefore, tended to be compiled. In contrast, languages that support a functional or logic/declarative style of programming involve mostly dynamic bindings and tend to be interpreted. (Chapter 4 discusses strategies for language implementation.)

1.4.8 Thought Process for Problem Solving

While most languages now support multiple styles of programming, use of the styles themselves involves a shift in one's problem-solving thought process. Thinking in one style (e.g., iteration—imperative) and programming in another style (e.g., functional, where recursive thought is fundamental) is analogous to translating into your native language every sentence you either hear from or speak to your conversational partner when participating in a synchronous dialog in a foreign language—an unsustainable strategy. Just as a one-to-one mapping between phrases in two natural languages—even those in the same family of languages (e.g., the Romance languages)—does not exist, it is generally not possible to translate the solution to a problem conceived with thought endemic to

one style (e.g., imperative thought) into another (e.g., functional constructs), and vice versa.

An advantageous outcome of learning to solve problems using an unfamiliar style of programming (e.g., functional, declarative) is that it involves a fundamental shift in one's thought process toward problem decomposition and solving. Learning to think and program in alternative styles typically entails unlearning bad habits acquired unconsciously through the use of other languages to accommodate the lack of support for that style in those languages. Consider how a programmer might implement an inherently recursive algorithm such as mergesort using a language without support for recursion:

Programming languages teach you not to want what they cannot provide. You have to think in a language to write programs in it, and it's hard to want something you can't describe. When I first started writing programs—in Basic—I didn't miss recursion, because I didn't know there was such a thing. I thought in Basic. I could only conceive of iterative algorithms, so why should I miss recursion? (Graham 1996, p. 2)

Paul Graham (2004b, p. 242) describes the effect languages have on thought as the *Blub Paradox*.¹⁷ Programming languages and the use thereof are—perhaps, so far—the only conduit into the science of computing experienced by students. Because language influences thought and capacity for thought, an improved understanding of programming languages and the different styles of programming supported by that understanding result in a more holistic view of computation.¹⁸ Indeed, a covert goal of this text or side effect of this course of study is to broaden the reader's understanding of computation by developing additional avenues through which to both experience and describe/effect computation in a computer program (Figure 1.3). An understanding of Latin—even an elementary understanding—not only helps one learn new languages but also improves one's use and command over their native language. Similarly, an understanding of both Lisp and the linguistic ideas central to it—and, more generally, the concepts of languages—will help you more easily learn new programming languages and make you a better programmer in your language of choice. “[L]earning Lisp will teach you more than just a new language—it will teach you new and more powerful ways of thinking about programs” (Graham 1996, p. 2).

1.5 Factors Influencing Language Development

Surprisingly enough, programming languages did not historically evolve based on the abilities of programmers (Weinberg 1988). (One could argue that programmers'

17. Notice use of the phrase “thinking in” instead of “programming in.”

18. The study of formal languages leads to the concept of a Turing machine; thus, language is integral to the theory of computation.

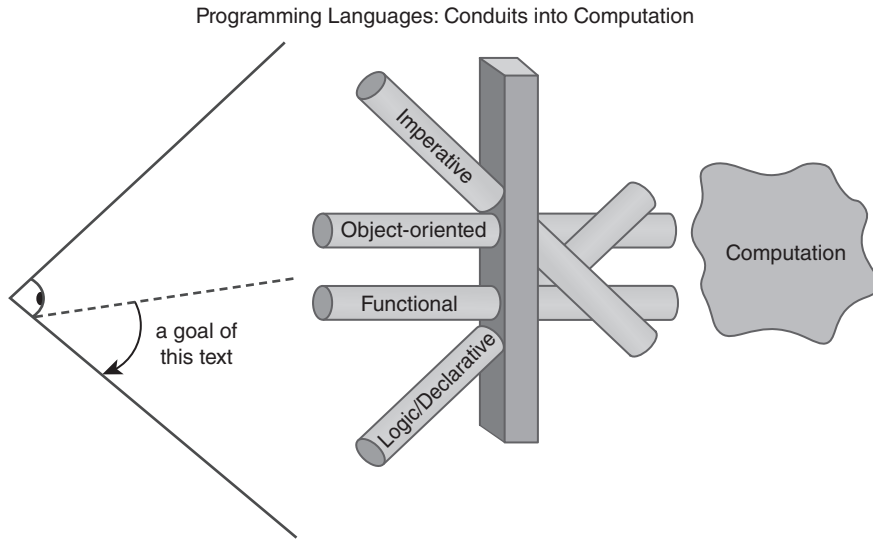


Figure 1.3 Programming languages and the styles of programming therein are conduits into computation.

abilities evolved based on the capabilities and limitations of programming languages.) Historically, computer architecture influenced programming language design and implementation. Use of the *von Neumann architecture* inspired the design of many early programming languages that dovetailed with that model. In the von Neumann architecture, a sequence of program instructions and program data are both stored in main memory. Similarly, the languages inspired by this model view variables (in which to store program data) as abstractions of memory cells. Further, in these languages variables are manipulated through a sequence of commands, including an assignment statement that changes the value of a variable.

Fortran is one of oldest programming languages still in use whose design was based on the von Neumann architecture. The primary design goal of Fortran was speed of execution since Fortran programs were intended for scientific and engineering applications and had to execute fast. Moreover, the emphasis on planning programs in advance advocated by software design methodologies (e.g., *structured programming* or *top-down design*) resulting from the *software crisis*¹⁹ in the 1960s and 1970s promoted the use of static bindings, which then reinforces the use of compiled languages. The need to produce programs that executed fast helped fuel the development of compiled languages such as Fortran, COBOL, and C. Compiled languages with static bindings and top-down design reinforce each other.

Often while developing software we build *throwaway prototypes* solely for purposes of helping us collect, crystallize, and analyze software requirements, candidate designs, and implementation approaches. It is widely believed that

¹⁹ The software crisis in the 1960s and 1970s refers to the software industry's inability to scale the software development process of large systems in the same way as other engineering disciplines.

writing generates and clarifies thoughts (Graham 1993, p. 2). For instance, the process of enumerating a list of groceries typically leads to thoughts of additional items that need to be purchased, which are then listed, and so on. An alternative to structured programming is *literate programming*, a notion introduced by Donald Knuth. Literate programming involves crafting a program as a representation of one's thoughts in natural language rather than based on constraints imposed by computer architecture and, therefore, programming languages.²⁰ Moreover, in the 1980s the discussion around the ideas of object-oriented design emerged through the development of Smalltalk—an interpreted language. Advances in computer hardware, and particularly *Moore's Law*,²¹ also helped reduce the emphasis on speed of program execution as the overriding criterion in the design of programming languages.

While fewer interpreted languages emerged in the 1980s compared to compiled ones, the confluence of literate programming, object-oriented design, and Moore's Law sparked discussion of speed of development as a criterion for designing programming languages.

The advent of the World Wide Web in the late 1990s and early 2000s and the new interactive and networked computing platform on which it runs certainly influenced language design. Language designers had to address the challenges of developing software that was intended to run on a variety of hardware platforms and was to be delivered or interacted with over a network. Moreover, they had to deal with issues of maintaining state—so fundamental to imperative programming—over a stateless (`http`) network protocol. For all these reasons, programming for the web presented a fertile landscape for the practical exploration of issues of language design. Programming languages tended toward the inclusion of more dynamic bindings, so more interpreted languages emerged at this time (e.g., JavaScript).

On the one hand, the need to develop applications with ever-evolving requirements rapidly has attracted attention to the speed of development as a more prominent criterion in the design of programming languages and has continued to nourish the development of languages adopting more dynamic bindings (e.g., Python). The ability, or lack thereof, to delay bindings until run-time affects flexibility of program development. The more dynamic bindings a language supports, the fewer the number of commitments the programmer must make during program development. Thus, dynamic bindings provide for convenient debugging, maintenance, and redesign when dealing with errors or evolving program requirements. For instance, run-time binding of messages to methods in Python allows programs to be more easily designed during their initial development and then subsequently extended during their maintenance.

20. While a novel concept, embraced by tools (e.g., *Noweb*) and languages (e.g., the proprietary language Miranda, which is a predecessor of Haskell and similarly supports a pure form of functional programming), the idea of literate programming never fully caught on.

21. *Moore's Law* states that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years and describes the evolution of computer hardware.

Graham (2004b) describes this process with a metaphor—namely, an oil painting where the painter can smudge the oil to correct any initial flaws. Thus, programming languages that support dynamic bindings are the oil that can reduce the cost of mistakes. There has been an incremental and ongoing shift toward support for more dynamic bindings in programming languages to enable the creation of malleable programs.

On the other hand, static type systems support program evolution by automatically identifying the parts of a program affected by a change in a data structure, for example (Wright 2010). Moreover, program safety and security are new applications of static bindings in languages (e.g., development of TypeScript as JavaScript with a safe type system). Figure 1.4 depicts the (historical) development of contemporary languages with dynamic bindings and languages with static bindings—both supporting multiple styles of programming. Languages

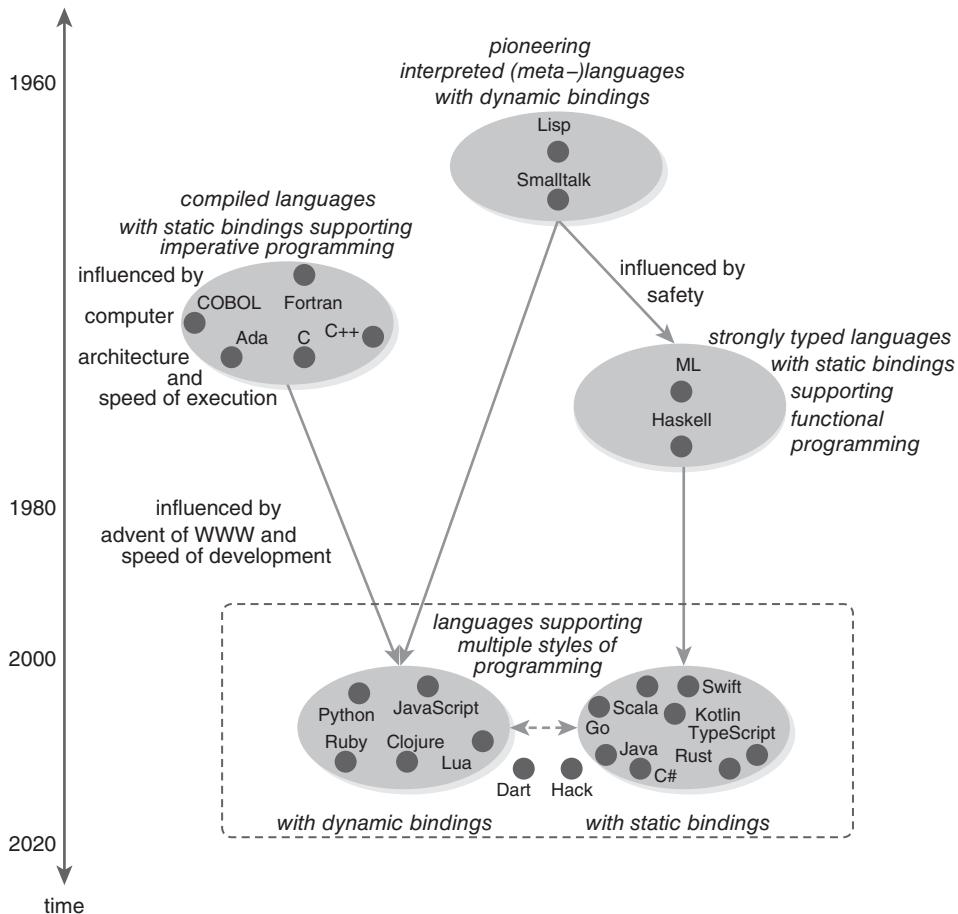


Figure 1.4 Evolution of programming languages emphasizing multiple shifts in language development across a time axis. (Time axis not drawn to scale.)

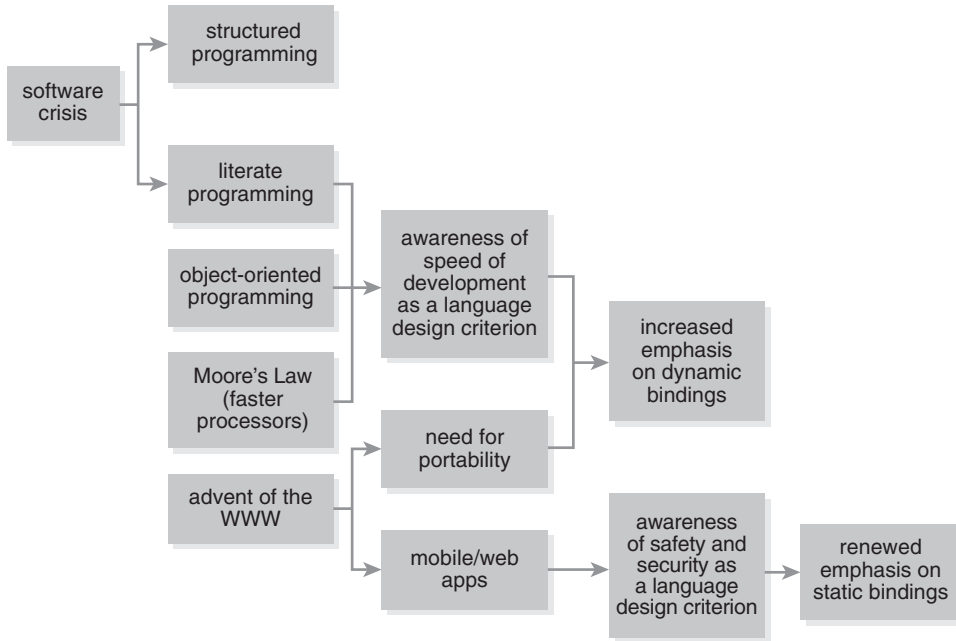


Figure 1.5 Factors influencing language design.

reconciling the need for both safety and flexibility are also starting to emerge (e.g., Hack and Dart). Figure 1.5 summarizes the factors influencing language design discussed here.

With the computing power available today and the time-to-market demands placed on software development, speed of execution is now less emphasized as a design criterion than it once was.²² Software development process methodologies have commensurately evolved in this direction as well and embrace this trend. *Agile methods* such as *extreme programming* involve repeated and rapid tours through the software development cycle, implying that speed of development is highly valued.

1.6 Recurring Themes in the Study of Languages

The following is a set of themes that recur throughout this text:

- A core set of language concepts are universal to all programming languages.
- There are a variety of options for language concepts, and individual languages differ on the design and implementation options for (some of) these concepts.

²². In some engineering applications, speed of execution is still the overriding design criterion.

- The concept of binding is fundamental to many other concepts in programming languages.
- Most issues in the design, implementation, and use of programming languages involve important practical trade-offs. For instance, there is an inverse relationship between static (rigid and fast) and dynamic (flexible and slow) bindings. Reliability, predictability, and safety are the primary motivations for using a statically typed programming language, while flexibility and efficiency are motivations for using a dynamically typed language.
- Side effects are often the underlying culprit of many programming perils.
- Like natural languages, programming languages have exceptions in how a language principle applies to entities in the language. Some languages are consistent (e.g., in Smalltalk everything is an object; Scheme uses prefix notation for built-in and user-defined functions and operators), while others are inconsistent (e.g., Java uses pass-by-value for primitives, but seemingly uses pass-by-reference for objects). There are fewer nuances to learn in consistent languages.
- There is a relationship between languages and the capacity to express ideas about computation.
 - Some idioms cannot be expressed as easily or at all in certain languages as they can in others.
 - Languages, through their support for a variety of programming styles (e.g., functional, declarative), require programmers to undertake a shift in thought process toward problem solving that develops additional avenues through which programmers can describe ideas about computation and, therefore, provides a more holistic view of computer science.
- Languages are built on top of languages.
- Languages evolve: The specific needs of application domains and development models influence language design and implementation options, and vice versa (e.g., speed of execution is less important as a design goal than it once was).
- Programming is an art (Knuth 1974a), and programs are works of art. The goal is not just to produce a functional solution to a problem, but to create a beautiful and reconfigurable program. Consider that architects seek to design not only structurally sound buildings, but buildings and environments that are aesthetically pleasing and foster social interactions.²³ “Great software, likewise, requires a fanatical devotion to beauty” (Graham 2004b, p. 29).

23. Architect Christopher Alexander and colleagues (1977) explored the relationship between (architectural) patterns and languages and, as a result, inspired *design patterns* in software (Gamma et al. 1995).

- Problem solving and subsequent programming implementation require pattern recognition and application, respectively.

To close the loop, we return to these themes in Chapter 15 (Conceptual Exercise 15.3).

1.7 What You Will Learn

The following is a succinct summary of some of the topics about which readers can expect to learn:

- fundamental and universal concepts of programming languages (e.g., scope and parameter passing) and the options available for them (e.g., lexical scoping, pass-by-name/lazy evaluation), especially from an implementation-oriented perspective
- language definition and description methods (e.g., grammars)
- how to design and implement language interpreters, and implementation strategies (e.g., inductive data types, data abstraction and representation)
- different styles of programming (e.g., functional, declarative, concurrent programming) and how to program using languages supporting those styles (e.g., Python, Scheme, ML, Haskell, and Prolog)
- types and type systems (through Python, ML, and Haskell)
- other concepts of programming languages (e.g., type inference, higher-order functions, currying)
- control abstraction, including first-class continuations

One approach to learning language concepts is to implement the studied concepts through the construction of a progressive series of interpreters, and to assess the differences in the resulting languages. One module of this text uses this approach. Specifically, in Chapters 10–12, we implement a programming language, named Camille, supporting functional and imperative programming through the construction of interpreters in Python.

We study and use type systems and other concepts of programming languages (e.g., type inference or currying) through the type-safe languages ML and Haskell in Chapter 7. We discuss a logic/declarative style of programming through use of Prolog in Chapter 14.

1.8 Learning Outcomes

Satisfying the text objectives outlined in Section 1.1 will lead to the following learning outcomes:

- an understanding of fundamental and universal language concepts, and design/implementation options for them
- an ability to deconstruct a language into its essential concepts and determine the implementation options for these concepts

- an ability to focus on the big picture (i.e., core concepts/features and options) and not the minutia (e.g., syntax)
- an ability to (more rapidly) understand (new or unfamiliar) programming languages
- an improved background and richer context for discerning appropriate languages for particular programming problems or application domains
- an understanding of and experience with a variety of programming styles or, in other words, an increased capacity to describe computational ideas
- a larger and richer arsenal of programming techniques to bring to bear upon problem-solving and programming tasks, which will make you a better programmer, in any language
- an increased ability to design and implement new languages
- an improved understanding of the (historical) context in which languages exist and evolve
- a more holistic view of computer science

The study of language concepts involves the development of a methodology and vocabulary for the subsequent comparative study of particular languages and results in both an improved aptitude for choosing the most appropriate language for the task at hand and a larger toolkit of programming techniques for building powerful and programming abstractions.

Conceptual Exercises for Chapter 1

Exercise 1.1 Given the definition of *programming language* presented in this chapter, is HTML a programming language? How about $\text{\LaTeX}$? Explain.

Exercise 1.2 Given the definition of a *programming language* presented in this chapter, is Prolog, which primarily supports a declarative style of programming, a programming language? How about Mercury, which supports a pure form of logic/declarative programming? Explain.

Exercise 1.3 There are many *times* in the study of programming languages. For example, variables are bound to types in C at *compile time*, which means that they remain fixed to their type for the lifetime of the program. In contrast, variables are bound to values at *run-time* (which means that a variable's value is not bound until run-time and can change at any time during run-time). In total, there are six (classic) times in the study of programming languages, of which compile time and run-time are two. Give an alternative time in the study of programming languages, and an example of something in C which is bound at that time.

Exercise 1.4 Are objects *first-class* in Java? C++?

Exercise 1.5 Explain how *first-class functions* can be simulated in C or C++. Write a C or C++ program to demonstrate.

Exercise 1.6 For each of the following entities, give all languages from the set {C++, ML, Prolog, Scheme, Smalltalk} in which the entity is considered *first-class*:

- (a) Function
- (b) Continuation
- (c) Object
- (d) Class

Exercise 1.7 Give a code example of a *side effect* in C.

Exercise 1.8 Are all functions without *side effect* *referentially transparent*? If not, give a function without a side effect that is not referentially transparent.

Exercise 1.9 Are all *referentially transparent* functions without *side effect*? If not, give a function that is referentially transparent, but has a side effect.

Exercise 1.10 Consider the following Java method:

```
1  int f() {
2      int a = 0;
3      a = a + 1;
4      return 10;
5  }
```

This function cannot modify its parameters because it has none. Moreover, it does not modify its external environment because it does not access any global data or perform any I/O. Therefore, the function does not have a side effect. However, the assignment statement on line 3 does have a side effect. How can this be? The function does not have a side effect, yet it contains a statement with a side effect—which seems like a contradiction. Does `f` have a side effect or not, and why?

Exercise 1.11 Identify two language evaluation criteria other than those discussed in this chapter.

Exercise 1.12 List two language evaluation criteria that conflict with each other. Provide two conflicts not discussed in this chapter. Give a specific example of each to illustrate the conflict.

Exercise 1.13 Fill in the blanks in the expressions in the following table with terms from the set:

{Dylan, garbage collection, Haskell,
lazy evaluation, Prolog, Smalltalk, static typing}

Go	=	C	+	_____
Curry	=	_____	+	Prolog
_____	=	Lisp	+	Smalltalk
Objective-C	=	C	+	_____
TypeScript	=	JavaScript	+	_____
Mercury	=	_____	–	impurities
Haskell	=	ML	+	_____

Exercise 1.14 What is *aspect-oriented programming*?

Exercise 1.15 Explore the *Linda* programming language. What styles of programming does it support? For which applications is it intended? What is *Linda-calculus* and how does it differ conceptually from λ -calculus?

Exercise 1.16 Identify a programming language with which you are unfamiliar—perhaps even a language mentioned in this chapter. Try to describe the language through its most defining characteristics.

Exercise 1.17 Read M. Swaine’s 2009 article “It’s Time to Get Good at Functional Programming” in *Dr. Dobbs’s Journal* and write a 250-word commentary on it.

Exercise 1.18 Read N. Savage’s 2018 article “Using Functions for Easier Programming” in *Communications of the ACM*, available at <https://doi.acm.org/10.1145/3193776>, and write a 100-word commentary on it.

Exercise 1.19 Write a 2000-word essay addressing the following questions:

- What interests you in programming languages?
- Which concepts or ideas presented in this chapter do you find compelling? With what do you agree or disagree? Why?
- What are your goals for this course of study?
- What questions do you have?

1.9 Thematic Takeaways

- This course of study is about *concepts* of programming languages.
- There is a universal lexicon for discussing the concepts of languages and for, more generally, engaging in this course of study, including the terms *binding*, *side effect*, and *first-class entity*.
- Programming languages differ in their design and implementation options for supporting a variety of concepts from a host of programming styles, including imperative, functional, object-oriented, and logic/declarative programming.
- The support for multiple styles of programming in a single language provides programmers with a richer palette in that language for expressing ideas about computation.
- Programming languages and the various styles of programming used therein are conduits into computation (Figure 1.3).
- Within the context of their support for a variety of programming styles, all languages involve a core set of universal concepts that are operationalized through an interpreter and provide a basis for (comparative) evaluation (Figure 1.2).
- The diversity of design and implementation options across programming languages provides fertile ground for comparative language analysis.

- A variety of factors influence the design and development of programming languages, including (historically) computer architecture, abilities of programmers, and development methodologies.
- The evolution of programming languages bifurcated into languages involving primarily static binding and those involving primarily dynamic bindings (Figure 1.4).

See also the recurrent themes in Section 1.6.

1.10 Chapter Summary

This text and course of study are about *concepts* of programming languages. There is a universal lexicon for discussing the concepts of languages and for, more generally, engaging in this course of study, including the terms *binding*, *side effect*, and *first-class entity*. Programming languages differ in their design and implementation options for supporting a variety of concepts from a host of programming styles, including imperative, functional, object-oriented, and logic/declarative programming. The *imperative style of programming* is a natural consequence of the *von Neumann architecture*: Instructions are imperative statements that affect, through an assignment operator, the values of variables, which are themselves abstractions of memory locations. Historically, programming languages were designed based on the computer architecture on which the programs written using them were intended to execute. The *functional style of programming* is rooted in λ -calculus—a mathematical theory of functions. The *logic/declarative style of programming* is grounded in *first-order predicate calculus*—a formal system of symbolic logic.

Thirty years ago, programming languages were clearly classified in these discrete categories or language *paradigms*, but that is no longer the case. Now most programming languages support a variety of styles of programming, including imperative, functional, object-oriented, and declarative programming (e.g., Python and Go). This diversity in programming styles supported in individual languages provides programmers with a richer palette in a single language for expressing ideas about computation—programming languages and the styles of programming used in these languages are conduits into computation. A goal of this text is to expose reader to these alternative styles of programming (Figure 1.3).

Within the context of their support for a variety of programming styles, all languages involve a core set of universal concepts (Figure 1.2). Programming languages differ in their design and implementation options for these core concepts as well as in the variety of concepts from the host of programming styles they support. This diversity of options in supporting concepts provides fertile ground for fostering a more meaningful comparative analysis of languages, while rendering the prevalent (and superficial) mode of language comparison of the past—putting languages in paradigms and comparing the paradigms—both irrelevant and nearly impossible. The evolution of programming languages

bifurcated into languages involving primarily static binding and those involving primarily dynamic bindings (Figure 1.4).

Since language concepts are the building blocks from which all languages are constructed/organized, an understanding of the concepts implies that one can focus on the core language principles (e.g., parameter passing) and the particular options (e.g., pass-by-reference) used for those principles in (new or unfamiliar) languages rather than fixating on the details (e.g., syntax), which results in an improved dexterity in learning, assimilating, and using programming languages. Moreover, an understanding and experience with a variety of programming styles and exotic ways of performing computation establishes an increased capacity for describing computation in a program, a richer toolbox of techniques from which to solve problems, and a more well-rounded picture of computing.

1.11 Notes and Further Reading

The term *paradigm* was coined by historian of science Thomas Kuhn. Since most programming languages no longer fit cleanly into the classical language paradigms, the concept of language *purity* (with respect to a particular paradigm) is pragmatically obsolete. The notion of a *first-class entity* is attributed to British computer scientist Christopher Strachey (Abelson and Sussman 1996, p. 76, footnote 64). John McCarthy, the original designer of Lisp, received the ACM A. M. Turing Award in 1971 for contributions to artificial intelligence, including the creation of Lisp.

Chapter 2

Formal Languages and Grammars

[If] one combines the words “to write-while-not-writing”: for then it means, that he has the power to write and not to write at once; whereas if one does not combine them, it means that when he is not writing he has the power to write.

— Aristotle, *Sophistical Refutations*, Book I, Part 4

Never odd or even

Is it crazy how saying sentences backwards creates backwards sentences saying how crazy it is

IN this chapter, we discuss the constructs (e.g., regular expressions and context-free grammars) for defining programming languages and explore their capabilities and limitations. Regular expressions can denote the lexemes of programming languages (e.g., an identifier), but not the higher-order syntactic structures (e.g., expressions and statements) of programming languages. In other words, regular expressions can denote identifiers and other lexemes while context-free grammars can capture the rules for a valid expression or statement. Neither can capture the rule that a variable must be declared before it is used. Context-free grammars are integral to both the definition and implementation of programming languages.

2.1 Chapter Objectives

- Introduce *syntax* and *semantics*.
- Describe formal methods for defining the syntax of a programming language.
- Establish an understanding of *regular languages*, *expressions*, and *grammars*.
- Discuss the use of *Backus–Naur Form* to define grammars.

- Establish an understanding of *context-free languages* and *grammars*.
- Introduce the role of *context* in programming languages and the challenges in modeling context.

2.2 Introduction to Formal Languages

An *alphabet* is a finite set of symbols denoted by Σ . A *string* is a combination of symbols, also called characters, over an alphabet. For instance, strings over the alphabet $\Sigma = \{a, b, c\}$ include a , aa , aaa , bb , aba , and abc . The *empty string* (i.e., a string of zero characters) is represented as ϵ . The *Kleene closure operator* of an alphabet (i.e., Σ^*) represents the set of all possible strings that can be constructed through zero or more concatenations of characters from the alphabet. Thus, the set of all possible strings from the alphabet $\Sigma = \{a, b, c\}$ is Σ^* . While Σ is always finite, Σ^* is always infinite and always contains ϵ . The strings in Σ^* are *candidate sentences*.

A *formal language* is a set of strings. Specifically, a formal language L is a subset of Σ^* , where each string from Σ^* in L is called a *sentence*. Thus, a formal language is a set of sentences. For instance, $\{a, aa, aaa, bb, aba, abc\}$ is a formal language. There are finite and infinite languages. Finite languages have a finite number of sentences. The language described previously is a finite language (i.e., it has six sentences), whereas the Scheme programming language is an infinite language. Most interesting languages are infinite.

Determining whether a string s from Σ^* is in L (i.e., whether the candidate sentence s is a valid sentence) depends on the complexity of L . For instance, determining if a string s from Σ^* is in the language of all three-character strings is simpler than determining if s is in the language of *palindromes* (i.e., strings that read the same both forward and backward; e.g., dad , eye , or $noon$). Thus, determining if a string is a sentence is a set-membership problem.

Recall that *syntax* refers to the *structure or form* of language and *semantics* refers to the *meaning* of language. Formal notational systems are available to define the syntax and semantics of formal languages. This chapter is concerned with establishing an understanding of those formal systems and how they are used to define the syntax of programming languages. Armed with an understanding of the theory of formal language definition mechanisms and methods, we can turn to practice and study how those devices can be used to recognize a valid program prior to interpretation or compilation in Chapter 3.

There are three progressive types of sentence validity. A sentence is *lexically* valid if all the words of the sentence are valid. A sentence is *syntactically* valid if it is lexically valid and the ordering of the words is valid. A sentence is *semantically* valid if it is lexically and syntactically valid and has a valid meaning.

Consider the sentences in Table 2.1. The first candidate sentence is not lexically valid because “saintt” is not a word; therefore, the sentence cannot be syntactically or semantically valid. The second candidate is lexically valid because all of its words are valid, but it is not syntactically valid because the arrangement of those words does not conform to the subject–verb–article–object structure of English sentences; thus, it cannot be semantically valid. The third candidate is

Candidate Sentence	Lexically Valid	Syntactically Valid	Semantically Valid
Augustine is a saintt.	×	×	×
Saint Augustine is a.	✓	×	×
Saint is a Augustine.	✓	✓	×
Augustine is a saint.	✓	✓	✓

Table 2.1 Progressive Types of Sentence Validity

Candidate Expression	Lexically Valid	Syntactically Valid	Semantically Valid
= intt + 3 y x;	×	×	×
= int + 3 y x;	✓	×	×
int 3 = y + x;	✓	✓	×
int y = x + 3;	✓	✓	✓

Table 2.2 Progressive Types of Program Expression Validity

lexically valid because all of its words are valid and syntactically valid because the arrangement of those words conforms to the subject–verb–article–object structure of English sentences, but it is not semantically valid because the sentence does not make sense. The fourth candidate sentence is lexically, syntactically, and semantically valid. Notice that these types of sentence validity are progressive. Once a candidate sentence fails any test for validity, it automatically fails a more stringent test for validity. In other words, if a candidate sentence does not even have valid words, those words can never be arranged correctly. Similarly, if the words of a candidate sentence are not arranged correctly, that sentence can never make semantic sense. For instance, the second sentence in Table 2.1 is not syntactically valid so it can never be semantically valid.

Recall that validating a string as a sentence is a set-membership problem. We saw previously that the first step to determining if a string of words, where a word is a string of non-whitespace characters, is a sentence is to determine if each individual word is a sentence (in a simpler language). Only after the validity of every individual word in the entire string is established can we examine whether the words are arranged in a proper order according to the particular language in which this particular, entire string is a candidate sentence. Notice that these steps are similar to the steps an interpreter or compiler must execute to determine the validity of a program (i.e., to determine if the program has any syntax errors). Table 2.2 illustrates these steps of determining program expression validity. Next, we examine those steps through a formal lens.

2.3 Regular Expressions and Regular Languages

2.3.1 Regular Expressions

Since languages can be infinite, we need a concise, yet formal method of describing languages. A *regular expression* is a pattern represented as a string that concisely

Regular Expression	Denotes	Language
Atomic Regular Expressions		
x	the single character x	$L(x) = x$
ϵ	empty string	$L(\epsilon) = \epsilon$
$\emptyset$	empty set	$L(\emptyset) = \{\}$
Compound Regular Expressions		
(r^*)	zero or more of r_1	$L((r)^*) = L(r)^*$
$(r_1 r_2)$	concatenation of r_1 and r_2	$L(r_1 r_2) = L(r_1)L(r_2)$
$(r_1 + r_2)$	either r_1 or r_2	$L(r_1 + r_2) = L(r_1) \cup L(r_2)$

Table 2.3 Regular Expressions (Key: $x \in \Sigma$.)

and formally denotes the strings of a language. A regular expression is itself a string in a language, albeit a *metalinguage*—a language used to describe a language. Thus, regular expressions have their own alphabet and syntax, not to be confused with the alphabet and syntax of the language that a regular expression is used to define.

Table 2.3 presents the six primitive constructs from which any regular expression can be constructed. These constructs are factored into three *primitive* regular expressions (i.e., x , ϵ , and $\emptyset$) and three *compound* regular expressions (constructed with the $*$, concatenation, and $+$ operators). Thus, some characters in the alphabet of regular expressions are special and called *metacharacters* [e.g., ϵ , $\emptyset$, $*$, $+$, $($, and $)$].¹ In particular, $\Sigma_{RE} = \{\epsilon, \emptyset, *, +, (,)\}$. We have already encountered the $*$ (or Kleene closure) operator as applied to a set of symbols (or alphabet). Here, it is applied to a regular expression r , where r^* denotes zero or more occurrences of r . For instance, the regular expression opus^* defines the language $\{\text{opus}, \text{opus}, \text{opus}, \text{opus}, \dots\}$. The regular expression $(\text{ab})^*$ denotes the language $\{\epsilon, \text{ab}, \text{abab}, \text{ababab}, \dots\}$. In both cases, the set of sentences, and therefore the language, are infinite. In short, a regular expression denotes a set of strings (i.e., the sentences of the language that the regular expression denotes).

The $+$ operator is used to construct a compound regular expression from two subexpressions, where the language denoted by the compound expression contains the strings from the union of the sets denoted by the two subexpressions. For instance, the regular expression “the + Java + programming + language” denotes the language $\{\text{the}, \text{Java} + \text{programming}, \text{language}\}$. Similarly,

$$\text{opus}(1+2+3+4+5+6+7+8+9)(0+1+2+3+4+5+6+7+8+9)^*$$

denotes the language

$$\{\text{opus1}, \text{opus2}, \dots, \text{opus9}, \text{opus10}, \text{opus11}, \dots, \text{opus98}, \text{opus99}\}$$

1. Sometimes some of the characters in the set of metacharacters are also in the alphabet of the language being defined (i.e., $\Sigma_{RE} \cap \Sigma \neq \emptyset$). In these cases, there must be a way to disambiguate the meaning of the overloaded character. For example, a $\backslash$ is used in UNIX to *escape* the special meaning of the metacharacter following it.

and

$(0+1+\dots+8+9)(0+1+\dots+8+9)(0+1+\dots+8+9)-(0+1+\dots+8+9)(0+1+\dots+8+9)-(0+1+\dots+8+9)(0+1+\dots+8+9)(0+1+\dots+8+9)$

which denotes the language of Social Security numbers.

Table 2.4 presents a set of compound regular expressions with the associated language that each denotes. Parentheses in compound regular expressions are used for grouping subexpressions. In the absence of parentheses, highest to lowest precedence proceeds in a top-down manner, as shown in Table 2.3 (e.g., $*$ has the highest precedence and $+$ has the lowest precedence).

An enumeration of the elements of a set of sentences defines a formal language *extensionally*, while a regular expression defines a formal language *intensionally*.

A regular expression is a *denotational* construct for a (certain type of) formal language. In other words, a regular expression denotes sentences from the language it represents. For example, the regular expression opus^* denotes the language $\{\text{opu}, \text{opus}, \text{opuss}, \text{opusss}, \dots\}$.

Regular expressions are implemented in a variety of UNIX tools (e.g., *grep*, *sed*, and *awk*). Most programming languages implement regular expressions

Regular Expression	Denotes	Regular Language
abc	the string abc	$\{\text{abc}\}$
$a + b + c$	any one character in the set $\{a, b, c\}$	$\{a, b, c\}$
$a + e + i + o + u$	any one character in the set $\{a, e, i, o, u\}$	$\{a, e, i, o, u\}$
$\epsilon + a$	"a" or the empty string	$\{\epsilon, a\}$
$a(b + c)$	"a" followed by any character in the set $\{b, c\}$	$\{ab, ac\}$
$ab + cd$	any one string in the set $\{ab, cd\}$	$\{ab, cd\}$
$a(b + c)d$	"a" followed by any character in the set $\{b, c\}$ followed by "d"	$\{abd, acd\}$
a^*	"a" zero or more times	$\{\epsilon, a, aa, aaa, \dots\}$
aa^*	"a" one or more times	$\{a, aa, aaa, \dots\}$
$aaaa^*$	"a" three or more times	$\{aaa, aaaa, aaaaa, \dots\}$
$aaaaaaaa$	"a" exactly eight times	$\{aaaaaaaa\}$
$a + aa + aaa + aaaa + aaaaa$	"a" between one and five times	$\{a, aa, aaa, aaaa, aaaaa\}$
$aaa + aaaa + aaaaa + aaaaaa$	"a" between three and six times	$\{aaa, aaaa, aaaaa, aaaaaa\}$

Table 2.4 Examples of Regular Expression ($\Sigma_{\text{re}} = \{\epsilon, \emptyset, *, +, (,)\}$.)

either natively in the case of scripting languages (e.g., Perl and Tcl) or through a library or package (e.g., Python, Java, Go).²

2.3.2 Finite-State Automata

Recall that a regular expression intensionally *denotes* (the sentences of) a regular language. Now we turn to a computational mechanism that can *decide* whether a string is a sentence in a particular language—the set-membership problem mentioned previously. A *finite-state automaton* (FSA) is a model of computation used to recognize whether a string is a sentence in a particular language. Figure 2.1 presents a finite-state automaton³ that recognizes sentences in the language denoted by the regular expression

$$(1+2+\dots+8+9)(0+1+2+\dots+8+9)^* +$$

$$(\_+a+b+\dots+y+z+A+B+\dots+Y+Z)(\_+a+b+\dots+y+z+A+B+\dots+Y+Z+0+1+\dots+8+9)^*$$

which describes positive integers and legal identifiers in the C programming language.

We can think of an automaton as a simplified computer (Figure 2.1) that, when given a string (i.e., candidate sentence) as input, outputs either yes or no to indicate whether the input string is in the particular language that the machine has been

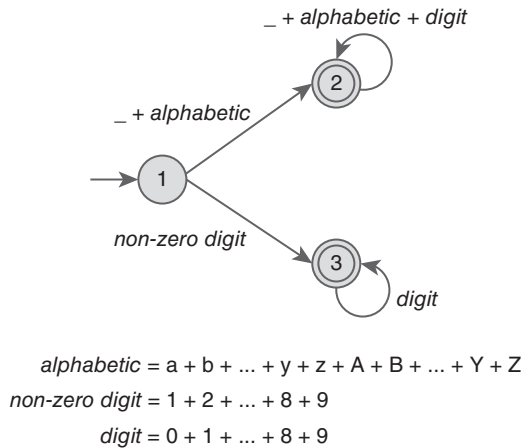


Figure 2.1 A finite-state automaton for a legal identifier and positive integer in the C programming language.

2. The set of metacharacters available to construct regular expressions in most programming languages and UNIX tools has evolved over the years beyond syntactic sugar (for formal regular expressions) and can be used to denote non-regular languages. For instance, the `grep` regular expression `\([a-z]\)\([a-z]\)\([a-z]\)\2\1` matches the language of palindromes of five-character, lowercase letters—a non-regular language.

3. More precisely, this finite-state automaton is a *nondeterministic finite automaton* or NFA. However, the FSA in Figure 2.1 is not a formally a FSA because it has only three transitions, but it should have one for each individual input character that moves the automaton from one state to another. For instance, there should be nine transitions between states 1 and 3—one for each non-zero digit.

constructed to recognize. In particular, if after running the entire string through the machine one character at a time, the automaton is left in an accepting state (i.e., one represented by a double circle, such as states 2 and 3 in Figure 2.1), the string is a sentence. If after running the string through the machine, the machine is left in a non-accepting state (i.e., one represented by a single circle, such as state 1 in Figure 2.1), the string is not a sentence. Formally, a FSA *decides* a language.

2.3.3 Regular Languages

A *regular language* is a formal language that can be denoted by a regular expression and recognized by a finite-state automaton. A regular language is the most restrictive type of formal language. A regular expression is a *denotational* construct for a regular language. In other words, a regular expression denotes sentences from the language it represents. For example, the regular expression opus^* denotes the regular language $\{\text{opu}, \text{opus}, \text{opuss}, \text{opusss}, \dots\}$.

If a language is finite, it can be denoted by a regular expression. This regular expression is constructed by enumerating each element of the finite set of sentences in the language with intervening $+$ metacharacters. For example, the finite language $\{a, b, c\}$ is denoted by the regular expression $a + b + c$. Thus, all finite languages are regular, but the reverse is not true.

In summary, a regular language (which is the most restrictive type of formal language) is denoted by a regular expression and is recognized by a finite-state automaton (which is the simplest model of computation).

Conceptual Exercises for Section 2.3

Exercise 2.3.1 Give a *regular expression* that defines a language whose sentences are the set of all strings of alphabetic (in any case) and numeric characters that are permissible as login IDs for a computer account, where the first character must be a letter and the string must contain at least one character, but no more than eight.

Exercise 2.3.2 Give a *regular expression* that denotes the language of five-digit zip codes (e.g., 45469) with an optional four-digit extension (e.g., 45469-0280).

Exercise 2.3.3 Give a *regular expression* to denote the language of phrases of exactly three words separated by whitespace, where a word is any string of non-whitespace characters and whitespace is any string of spaces or tabs. In your expression, represent a single space character as $\square$ and a single tab character as $\rightarrow$. Among the set of sentences that your regular expression denotes are the three underlined substrings in the following string: A room with a view.

Exercise 2.3.4 Give a *regular expression* that denotes the language of decimals representing ASCII characters (i.e., integers between 0–127, without leading 0s for any integer except 0 itself). Thus, the strings 0, 2, 25, and 127 are in the language, but 00, 02, 000, 025, and 255 are not.

Exercise 2.3.5 Give a *regular expression* for the language of zero or more nested, matched parentheses, where every opening and closing parenthesis has a match of the other type, with the matching opening parentheses appearing before the matching closing parentheses in the sentence, but where the parentheses are never nested more than three levels deep (i.e., no character in the string is ever within more than three levels of nesting). To avoid confusion between parentheses in the string and parentheses used for grouping in the regular expression, use the “l” and “r” characters to denote left (i.e., opening) and right (i.e., closing) parentheses in the string, respectively.

Exercise 2.3.6 Since all finite languages are regular, we can construct an FSA for any finite language. Describe how an FSA for a finite language can be constructed.

2.4 Grammars and Backus–Naur Form

Grammars are yet another way to define languages. A *formal grammar* is used to define a formal language. The following is a formal grammar defined for the language denoted by the a^* regular expression:

$$\begin{array}{lcl} S & \rightarrow & aS \\ S & \rightarrow & \epsilon \end{array}$$

The formal definition of a *grammar* is $G = (V, \Sigma, P, S)$, where

- V is a set of *non-terminal* symbols (e.g., $\{S\}$ in the grammar shown here).
- Σ is an alphabet (e.g., $\Sigma = \{a\}$).
- P is a finite set of *production rules*, each of the form $x \rightarrow y$, where x and y are strings over $\Sigma \cup V$ and $x \neq \epsilon$ (or, alternatively, P is a finite relation $P : V \rightarrow (V \cup \Sigma)^*$ (e.g., each line in the example grammar is a production rule).
- S is the *start symbol* and $S \in V$ (e.g., S).

V is called the *non-terminal* alphabet, while Σ is the *terminal* alphabet, and $V \cap \Sigma = \emptyset$. In other words, strings of symbols from Σ are called *terminals*. Formally, for each terminal t , $t \in \Sigma^*$ (e.g., “a” in the example grammar is the only terminal). We can think of terminals as the atomic lexical units of a program, called *lexemes*. The example grammar is defined formally as $G = (\{S\}, \{a\}, S, \{S \rightarrow aS, S \rightarrow \epsilon\})$.

Notice that a grammar is a metalanguage, or a language that describes a language. Moreover, like regular expressions, grammars have their own syntax—again, not to be confused with the syntax of the languages they are used to define. Thus, grammars themselves are defined using a metalanguage—a language for defining a language, which, in this case, could itself be called a metalanguage—a language for defining a language defines a language! A metalanguage for defining grammars is called *Backus–Naur Form* (BNF). BNF takes its name from the last names of John Backus, who developed the notation and used it to define the syntax of ALGOL 58 at IBM, and Peter Naur, who later extended the notation and used it for ALGOL 60 (Section 2.10). The example grammar G is in BNF.

By applying the production rules, beginning with the start symbol, a grammar can be used to *generate* a sentence from the language it defines. For instance, the following is a *derivation* of the sentence aaaa:

$$S \xRightarrow{r_1} aS \xRightarrow{r_1} aaS \xRightarrow{r_1} aaaS \xRightarrow{r_1} aaaaS \xRightarrow{r_2} aaaa$$

Note that every application of a production rule involves replacing the non-terminal on the left-hand side of the rule with the entire right-hand side of the rule. The semantics of the symbol $\Rightarrow$ is “derives” and the symbol indicates a one-step derivation relation. The r_n annotation over each $\Rightarrow$ symbol indicates which production rule is used in the substitution. The $\Rightarrow^*$ symbol indicates a zero-or-more-step derivation relation. Thus, $S \Rightarrow^* aaaa$.

A formal grammar is a *generative* construct for a formal language. In other words, a grammar generates sentences from the language it defines. Formally, if $G = (V, \Sigma, S, P)$, then the language generated by G is $L(G) = \{x \mid x \in \Sigma^* \text{ and } S \Rightarrow^* x\}$. A grammar for the language denoted by the regular expression `opus*` is $(\{S, W\}, \{o, p, u, s\}, \{S \rightarrow \text{opu}W, W \rightarrow \text{s}W, W \rightarrow \epsilon\})$, which generates the language $\{\text{opu}, \text{opus}, \text{opuss}, \dots\}$.

2.4.1 Regular Grammars

Linguist Noam Chomsky formalized a set of grammars in the late 1950s—unintentionally making a seminal contribution to computer science. Chomsky’s work resulted in the *Chomsky hierarchy*, which is a progressive classification of formal grammars used to describe the syntax of languages.

Level 1 of the hierarchy defines a type of formal grammar, called a *regular grammar*, which is most appropriate for describing the lexemes of programming languages (e.g., keywords in C such as `int` and `float`). The complete set of lexemes of a language is referred to as a *lexicon* (or *lexis*). A grammar is a *regular grammar* if and only if every production rule is in one of the following two forms:

$$\begin{aligned} X &\rightarrow zY \\ X &\rightarrow z \end{aligned}$$

where $X \in V$, $Y \in V$, and $z \in \Sigma^*$. A grammar whose production rules conform to these patterns is called a *right-linear grammar*. Grammars whose production rules conform to the following pattern are called *left-linear grammars*:

$$\begin{aligned} X &\rightarrow Yz \\ X &\rightarrow z \end{aligned}$$

Left-linear grammars also generate regular languages. Notice the one-for-one replacement of a non-terminal for a non-terminal in V in the rules of a right- or left-linear grammar. Thus, a regular grammar is also referred to as a *linear grammar*. Regular grammars define a class of languages known as *regular languages*.

A regular grammar is a *generative* device for a regular language. In other words, it generates sentences from the regular language it defines. However, a grammar does not have to be regular to generate a regular language. We leave it as an

Regular expressions	<i>denote</i>	regular languages.
Regular grammars	<i>generate</i>	regular languages.
Finite-state automata	<i>recognize</i>	regular languages.
All three	<i>define</i>	regular languages.

Table 2.5 Relationship of Regular Expressions, Regular Grammars, and Finite-State Automata to Regular Languages

exercise to define a non-regular grammar that defines a regular language (i.e., one that can be denoted by a regular expression; Conceptual Exercise 2.10.7).

In summary, a regular language (which is the most restrictive type of formal language) is:

- *denoted* by a regular expression,
- *recognized* by a finite-state automaton (which is the simplest model of computation), and
- *generated* by a regular grammar.

See Table 2.5.

Regular expressions, regular grammars, and finite-state automata are equivalent in their power to denote, generate, and recognize regular languages. In other words, there does not exist a regular language that could be *denoted* with a regular expression that could not be *decided* by a FSA or *generated* by a regular grammar. Mechanical techniques can be used to convert from one of these three models of a regular language to any of the other two.

An enumeration of the elements of a set of sentences defines a regular language *extensionally*, while a regular expression, finite-state automata, and regular grammar each define a regular language *intensionally*.

Some formal languages are not regular. Moreover, grammars, in addition to being language-generation devices, can be used (like an FSA) as language-recognition devices. We return to this theme of the dual nature of grammars while discussing context-free grammars in the next section.

2.5 Context-Free Languages and Grammars

There is a limit on the expressivity of regular expressions and regular grammars. In other words, some languages cannot be defined by a regular expression or a regular grammar. As a result, there are also computational limits on the sentence-recognition capabilities of finite-state automata. Consider the language L of balanced parentheses, whose sentences are strings of nested parentheses with the same number of opening parentheses in the first half of the string as closing parentheses in the second half of the string: $L = \{ (^n)^n \mid n \geq 0 \text{ and } \Sigma = \{ (,) \} \}$. The strings $()$ and $((((()))$ are balanced and, therefore, sentences in this language; conversely, the strings $(, (())$, and $((())$ are unbalanced and not in the language. In formal language theory, a language of strings of balanced parentheses is called

a *Dyck language*. A Dyck language cannot be defined by a regular expression. Alternatively, consider the language L of binary *palindromes*—binary numbers that read the same forward as backward: $L = \{xx^r \mid x \in \{0, 1\}^*\}$, where x^r means “a reversed copy of x .” The strings 00, 11, 101, 010, 1111, and 001100 are in the language, but 01, 10, 1000, and 1101 are not. We cannot construct either a regular expression or a regular grammar to define these languages. In other words, neither a regular expression nor a regular grammar has the expressive capability to model these languages.

What capability is absent from regular expressions or regular grammars that renders them unusable for defining these languages? Consider how we might implement a computer program to recognize strings of balanced parentheses. We could use a stack data structure to match each opening parenthesis with a closing parenthesis. Whenever we encounter an open parenthesis, we push it onto the stack; whenever we see a closing parenthesis, we pop from the stack. If the stack is empty when all the characters in the string are consumed, then the parentheses in the string are balanced and the string is a sentence; otherwise, it is not. The utility of a stack (formally, a *pushdown automata*) for this purpose implies that we need some form of *unbounded* memory to the match parentheses in the candidate string (i.e., to keep track of the number of unclosed open parentheses unknown a priori). Recall that the F in FSA stands for *finite*.

While regular expressions can denote the lexemes (e.g., identifiers) of programming languages, they cannot model syntactic structures nested arbitrarily deep that involve *balanced pairs* of lexemes (e.g., matched curly braces or *begin/end* keyword pairs identifying blocks of code; or parentheses in mathematical expressions), which are ubiquitous in programming languages. In other words, a sequence of lexemes in a program must be arranged in a particular order, and that order cannot be captured by a regular expression or a regular grammar. Regular expressions are expressive enough to denote the lexemes of programming languages, but not the higher-order syntactic structures (e.g., expressions and statements) of programming languages. Therefore, we must turn our attention to formal grammars with greater expressive capabilities than regular grammars if we need to define more sophisticated formal languages, including, in particular, programming languages.

Level 2 of the Chomsky hierarchy defines a type of formal grammar, called a *context-free grammar*, which is most appropriate for defining (and, as we see later, implementing) programming languages. Like the production rules of a regular grammar, the productions of a context-free grammar must conform to a particular pattern, but that pattern is less restrictive than the pattern to which regular grammars must adhere. The productions of a context-free grammar may have only one non-terminal on the left-hand side. Formally, a grammar is a *context-free grammar* if and only if every production rule is in the following form:

$$X \rightarrow \gamma$$

where $X \in V$ and $\gamma \in (\Sigma \cup V)^*$, there is only one non-terminal on the left-hand side of any rule, and X can be replaced with γ anywhere. Notice that since this

definition is less restrictive than that of a regular grammar, every regular grammar is also a context-free grammar, but the reverse is not true.

Context-free grammars define a class of formal languages called *context-free languages*. The concept of balanced pairs of syntactic entities—the essence of a Dyck language—is at the heart of context-free languages. This single syntactic feature (and its variations) distinguishes regular languages from context-free languages, and the capability of expressing balanced pairs is the essence of a context-free grammars.

2.6 Language Generation: Sentence Derivations

Consider the following a context-free grammar defined in BNF for simple English sentences:

(r_1)	$\langle \text{sentence} \rangle$	$\rightarrow$	$\langle \text{article} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle \langle \text{adverb} \rangle .$
(r_2)	$\langle \text{article} \rangle$	$\rightarrow$	a
(r_3)	$\langle \text{article} \rangle$	$\rightarrow$	an
(r_4)	$\langle \text{article} \rangle$	$\rightarrow$	the
(r_5)	$\langle \text{noun} \rangle$	$\rightarrow$	apple
(r_6)	$\langle \text{noun} \rangle$	$\rightarrow$	rose
(r_7)	$\langle \text{noun} \rangle$	$\rightarrow$	umbrella
(r_8)	$\langle \text{verb} \rangle$	$\rightarrow$	is
(r_9)	$\langle \text{verb} \rangle$	$\rightarrow$	appears
(r_{10})	$\langle \text{adverb} \rangle$	$\rightarrow$	here
(r_{11})	$\langle \text{adverb} \rangle$	$\rightarrow$	there

As briefly shown here, grammars are used to generate sentences from the language they define. Beginning with the start symbol and repeatedly applying the production rules until the string contains no non-terminals results in a *derivation*—a sequence of applications of the production rules of a grammar beginning with the start symbol and ending with a sentence (i.e., a string of all terminals arranged according to the rules of the grammar). For example, consider deriving the sentence “the apple is there.” from the preceding grammar. The r_n parenthesized annotation on the right-hand side of each application indicates which production rule was used in the substitution:

$\langle \text{sentence} \rangle$	$\Rightarrow$	$\langle \text{article} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle \langle \text{adverb} \rangle .$	(r_1)
	$\Rightarrow$	$\langle \text{article} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle$ there.	(r_{11})
	$\Rightarrow$	$\langle \text{article} \rangle \langle \text{noun} \rangle$ is there.	(r_8)
	$\Rightarrow$	$\langle \text{article} \rangle$ apple is there.	(r_5)
	$\Rightarrow$	the apple is there.	(r_4)

The result (on the right-hand side of the $\Rightarrow$ symbol) of each step is a string containing terminals and non-terminals that is called a *sentential form*. A *sentence* is a sentential form containing only terminals.

Peter Naur extended BNF for ALGOL 60 to make the definition of the production rules in a grammar more concise. While we discuss the details of

the extension, called *Extended Backus–Naur Form* (EBNF), later (in Section 2.10), we cover one element of the extension, alternation, here since we use it in the following examples. Alternation allows us to consolidate various production rules whose left-hand sides match into a single rule whose right-hand side consists of the right-hand sides of each of the individual rules separated by the $|$ symbol. Therefore, alternation is syntactic sugar, in that any grammar using it can be rewritten without it. *Syntactic sugar* is a term coined by Peter Landin that refers to special, typically terse syntax in a language that serves only as a convenient method for expressing syntactic structures that are traditionally represented in the language through uniform and often long-winded syntax. With alternation, we can define the preceding grammar, which contains 11 production rules with only 5 rules:

- (r_1) $\langle \text{sentence} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle \langle \text{adverb} \rangle.$
- (r_2) $\langle \text{article} \rangle \rightarrow a \mid \text{an} \mid \text{the}$
- (r_3) $\langle \text{noun} \rangle \rightarrow \text{apple} \mid \text{rose} \mid \text{umbrella}$
- (r_4) $\langle \text{verb} \rangle \rightarrow \text{is} \mid \text{appears}$
- (r_5) $\langle \text{adverb} \rangle \rightarrow \text{here} \mid \text{there}$

To differentiate non-terminals from terminals, especially when using grammars to describe programming languages, we place non-terminal symbols within the symbols $\langle \rangle$ by convention.⁴

Consider the following context-free grammar for arithmetic expressions for a simple four-function calculator with three available identifiers:

- (r_1) $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$
- (r_2) $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle - \langle \text{expr} \rangle$
- (r_3) $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle * \langle \text{expr} \rangle$
- (r_4) $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle / \langle \text{expr} \rangle$
- (r_5) $\langle \text{expr} \rangle ::= \langle \text{id} \rangle$
- (r_6) $\langle \text{id} \rangle ::= x \mid y \mid z$
- (r_7) $\langle \text{expr} \rangle ::= (\langle \text{expr} \rangle)$
- (r_8) $\langle \text{expr} \rangle ::= \langle \text{number} \rangle$
- (r_9) $\langle \text{number} \rangle ::= \langle \text{number} \rangle \langle \text{digit} \rangle$
- (r_{10}) $\langle \text{number} \rangle ::= \langle \text{digit} \rangle$
- (r_{11}) $\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

A derivation is called *leftmost* if the leftmost non-terminal is always replaced first in each step. The following is a leftmost derivation of 132:

- $\langle \text{expr} \rangle \Rightarrow \langle \text{number} \rangle \quad (r_8)$
- $\Rightarrow \langle \text{number} \rangle \langle \text{digit} \rangle \quad (r_9)$
- $\Rightarrow \langle \text{number} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle \quad (r_9)$

4. Interestingly, Chomsky and Backus/Naur developed their notion for defining grammars independently. Thus, the two notions have some minor differences: Chomsky used uppercase letters for non-terminals, the $\rightarrow$ symbol in production rules, and ϵ as the empty string; Backus/Naur used words in any case enclosed in $\langle \rangle$ symbols, $::=$, and $\langle \text{empty} \rangle$, respectively.

$$\begin{aligned}
\Rightarrow & \langle \text{digit} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle & (r_{10}) \\
\Rightarrow & 1 \langle \text{digit} \rangle \langle \text{digit} \rangle & (r_{11}) \\
\Rightarrow & 13 \langle \text{digit} \rangle & (r_{11}) \\
\Rightarrow & 132 & (r_{11})
\end{aligned}$$

A derivation is called *rightmost* if the rightmost non-terminal is always replaced first in each step. The following is a rightmost derivation of 132:

$$\begin{aligned}
\langle \text{expr} \rangle & \Rightarrow \langle \text{number} \rangle & (r_8) \\
& \Rightarrow \langle \text{number} \rangle \langle \text{digit} \rangle & (r_9) \\
& \Rightarrow \langle \text{number} \rangle 2 & (r_{11}) \\
& \Rightarrow \langle \text{number} \rangle \langle \text{digit} \rangle 2 & (r_9) \\
& \Rightarrow \langle \text{number} \rangle 32 & (r_{11}) \\
& \Rightarrow \langle \text{digit} \rangle 32 & (r_{10}) \\
& \Rightarrow 132 & (r_{11})
\end{aligned}$$

Some derivations, such as the next two derivations, are neither leftmost nor rightmost:

$$\begin{aligned}
\langle \text{expr} \rangle & \Rightarrow \langle \text{number} \rangle & (r_8) \\
& \Rightarrow \langle \text{number} \rangle \langle \text{digit} \rangle & (r_9) \\
& \Rightarrow \langle \text{number} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle & (r_9) \\
& \Rightarrow \langle \text{number} \rangle \langle \text{digit} \rangle 2 & (r_{11}) \\
& \Rightarrow \langle \text{number} \rangle 32 & (r_{11}) \\
& \Rightarrow \langle \text{digit} \rangle 32 & (r_{10}) \\
& \Rightarrow 132 & (r_{11})
\end{aligned}$$

$$\begin{aligned}
\langle \text{expr} \rangle & \Rightarrow \langle \text{number} \rangle & (r_8) \\
& \Rightarrow \langle \text{number} \rangle \langle \text{digit} \rangle & (r_9) \\
& \Rightarrow \langle \text{number} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle & (r_9) \\
& \Rightarrow \langle \text{number} \rangle 3 \langle \text{digit} \rangle & (r_{11}) \\
& \Rightarrow \langle \text{digit} \rangle 3 \langle \text{digit} \rangle & (r_{10}) \\
& \Rightarrow 13 \langle \text{digit} \rangle & (r_{11}) \\
& \Rightarrow 132 & (r_{11})
\end{aligned}$$

The following is a rightmost derivation of $x + y * z$:

$$\begin{aligned}
\langle \text{expr} \rangle & \Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle & (r_1) \\
& \Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle * \langle \text{expr} \rangle & (r_3) \\
& \Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle * \langle \text{id} \rangle & (r_5) \\
& \Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle * z & (r_6) \\
& \Rightarrow \langle \text{expr} \rangle + \langle \text{id} \rangle * z & (r_5) \\
& \Rightarrow \langle \text{expr} \rangle + y * z & (r_6) \\
& \Rightarrow \langle \text{id} \rangle + y * z & (r_5) \\
& \Rightarrow x + y * z & (r_6)
\end{aligned}$$

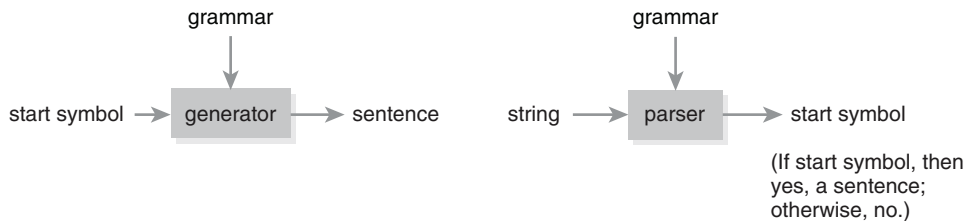


Figure 2.2 The dual nature of grammars as generative and recognition devices. (left) A language generator that accepts a grammar and a start symbol and generates a sentence from the language defined by the grammar. (right) A language parser that accepts a grammar and a string and determines if the string is in the language.

2.7 Language Recognition: Parsing

In the prior subsection we used context-free grammars as language generation devices to construct derivations. We can also implement a computer program to construct derivations; that is, to randomly choose the rules used to substitute non-terminals. That sentence-generator program takes a grammar as input and outputs a random sentence in the language defined by that grammar (see the left side of Figure 2.2). One of the seminal discoveries in computer science is that grammars can (like finite-state automata) also be used for language *recognition*—the reverse of generation. Thus, we can implement a computer program to accept a candidate string as input and construct a rightmost derivation in reverse to determine whether the input string is a sentence in the language defined by the grammar (see the right side of Figure 2.2). That computer program is called a *parser* and the process of constructing the derivation is called *parsing*—the topic of Chapter 3. If in constructing the rightmost derivation in reverse we return to the start symbol when the input string is expired, then the string is a sentence; otherwise, it is not.

Language *generation*: start symbol $\longrightarrow$ sentence
 Language *recognition*: sentence $\longrightarrow$ start symbol

A *generator* applies the production rules of a grammar *forward*. A *parser* applies the rules *backward*.⁵

Consider parsing the string $x + y * z$. In the following parse, $.$ denotes “top of the stack”:

1	$. x + y * z$	(shift)
2	$x . + y * z$	(reduce r_6)
3	$\langle id \rangle . + y * z$	(reduce r_5)
4	$\langle expr \rangle . + y * z$	(shift)
5	$\langle expr \rangle + . y * z$	(shift)
6	$\langle expr \rangle + y . * z$	(reduce r_6)

5. Another class of parsers applies production rules in a top-down fashion (Section 3.4).

7	$\langle \text{expr} \rangle + \langle \text{id} \rangle . * z$	(reduce r_5)
8	$\langle \text{expr} \rangle + \langle \text{expr} \rangle . * z$	(shift; why not reduce r_1 here instead?)
9	$\langle \text{expr} \rangle + \langle \text{expr} \rangle * . z$	(shift)
10	$\langle \text{expr} \rangle + \langle \text{expr} \rangle * z .$	(reduce r_6)
11	$\langle \text{expr} \rangle + \langle \text{expr} \rangle * \langle \text{id} \rangle .$	(reduce r_5)
12	$\langle \text{expr} \rangle + \langle \text{expr} \rangle * \langle \text{expr} \rangle .$	(reduce r_3 ; emit multiplication)
13	$\langle \text{expr} \rangle + \langle \text{expr} \rangle .$	(reduce r_1 ; emit addition)
14	$\langle \text{expr} \rangle .$	(start symbol; this is a sentence)

The left-hand side of the $.$ represents a stack and the right-hand side of the $.$ (i.e., the top of the stack) represents the remainder of the string to be parsed, called the *handle*. At each step, either shift or reduce. To determine which to do, examine the stack. If the items at the top of the stack match the right-hand side of any production rule, replace those items with the non-terminal on the left-hand side of that rule. This is known as *reducing*. If the items at the top of the stack do not match the right-hand side of any production rule, shift the next lexeme on the right-hand side of the $.$ to the stack. If the stack contains only the start symbol when the input string is entirely consumed (i.e., shifted), then the string is a sentence; otherwise, it is not.

This process is called *shift-reduce* or *bottom-up* parsing because it starts with the string or, in other words, the terminals, and works back through the non-terminals to the start symbol. A bottom-up parse of an input string constructs a rightmost derivation of the string in reverse (i.e., bottom-up). For instance, notice that reading the lines of the rightmost derivation in Section 2.6 in reverse (i.e., from the bottom line up to the top line) corresponds to the shift-reduce parsing method discussed here. In particular, the production rules in the preceding shift-reduce parse of the string $x + y * z$ are applied in reverse order as those in the rightmost derivation of the same string in Section 2.6. Later, in Chapter 3, we contrast this method of parsing with *top-down* or *recursive-descent* parsing. The preceding parse proves that $x + y * z$ is a sentence.

2.8 Syntactic Ambiguity

The following parse, although different from that in Section 2.7, proves precisely the same result—that the string is a sentence.

1	$. x + y * z$	(shift)
2	$x . + y * z$	(reduce r_6)
3	$\langle \text{id} \rangle . + y * z$	(reduce r_5)
4	$\langle \text{expr} \rangle . + y * z$	(shift)
5	$\langle \text{expr} \rangle + . y * z$	(shift)
6	$\langle \text{expr} \rangle + y . * z$	(reduce r_6)
7	$\langle \text{expr} \rangle + \langle \text{id} \rangle . * z$	(reduce r_5)
8	$\langle \text{expr} \rangle + \langle \text{expr} \rangle . * z$	(reduce r_1 ; emit addition; why not shift here instead?)

A formal grammar defines only the *syntax* of a formal language.
 A BNF grammar defines the *syntax* of a programming language,
 and some of its *semantics* as well.

Table 2.6 Formal Grammars Vis-à-Vis BNF Grammars

9	<code><expr> . * z</code>	(shift)
10	<code><expr> * . z</code>	(shift)
11	<code><expr> * z .</code>	(reduce r_6)
12	<code><expr> * <id> .</code>	(reduce r_5)
13	<code><expr> * <expr> .</code>	(reduce r_3 ; emit multiplication)
14	<code><expr> .</code>	(start symbol; this is a sentence)

Which of these two parses is preferred? How can we evaluate which is preferred? On what criteria should we evaluate them? The short answer to these questions is: It does not matter. The objective of language recognition and parsing is to determine if the input string is a sentence (i.e., does its structure conform to the grammar). Both of these parses meet that objective; thus, with respect to syntax, they both equally meet the objective. Here, we are only concerned with the syntactic validity of the string, not whether it makes sense (i.e., semantic validity). Parsing deals with syntax rather than semantics.

However, parsers often address issues of semantics with techniques originally intended only for addressing syntactic validity. One reason for this is that, unfortunately, unlike for syntax, we do not have formal models of semantics that are easily implemented in a computer system. Another reason is that addressing semantics while parsing can obviate the need to make multiple passes through the input string. While formal systems help us reason about concepts such as syntax and semantics, programming language systems implemented based on these formalisms must address practical issues such as efficiency. (Certain types of parsers require the production rules of the grammar of the language of the sentences they parse to be in a particular form, even though the same language can be defined using production rules in multiple forms. We discuss this concept in Chapter 3.) Therefore, although this approach is considered impure from a formal perspective, sometimes we address syntax and semantics at the same time (Table 2.6).

2.8.1 Modeling Some Semantics in Syntax

One way to gently introduce semantics into syntax is to think of syntax implying semantics as a desideratum. In other words, the form of an expression or command (i.e., its syntax) should provide some clue as to its meaning (i.e., semantics). A complaint against UNIX systems vis-à-vis systems with graphical user interfaces is that the form (i.e., syntax) of a UNIX command does not imply the meaning (i.e., semantics) of the command (e.g., `ls`, `ps`, and `grep` vis-à-vis `date` and `whoami`). The idea of integrating semantics into syntax may not seem so foreign a concept. For instance, we are taught in introductory computer programming courses to use

identifier names that imply the meaning of the variable to which they refer (e.g., `rate` and `index` vis-à-vis `x` and `y`).

Here we would like to infuse semantics into parsing in an identifiable way. Specifically, we would like to evaluate the expression while parsing it. This helps us avoid making unnecessary passes over the string if it is a sentence. Again, it is important to realize we are shifting from the realm of syntactic validity into interpretation. The two should not be confused, as they serve different purposes. Determining if a string is a sentence is completely independent of evaluating it for a return value. We often subconsciously impart semantics onto an expression such as $x + y * z$ because without any mention of meaning we presume it is a mathematical expression. However, it is simply a string conforming to a syntax (i.e., form) and can have any interpretation or meaning we impart to it. Indeed, the meaning of the expression $x + y * z$ could be a list of five elements.

Thus, in evaluating an expression while parsing it, we are imparting knowledge of how to interpret the expression (i.e., semantics). Here, we interpret these sentences as standard mathematical expressions. However, to evaluate these mathematical expressions, we must adopt even more semantics beyond the simple interpretation of them as mathematical expressions. If they are mathematical expressions, to evaluate them we must determine which operators have *precedence* over each other [i.e., is $x + y * z$ interpreted as $(x + y) * z$ or $x + (y * z)$] as well as the order in which each operator *associates* [i.e., is $6 - 3 - 2$ interpreted as $(6 - 3) - 2$ or $6 - (3 - 2)$?]. *Precedence* deals with the order of distinct operators (e.g., $*$ computes before $+$), while *associativity* deals with the order of operators with the same precedence (e.g., $-$ associates left-to-right).

Formally, a binary operator $\oplus$ on a set S is *associative* if $(a \oplus b) \oplus c = a \oplus (b \oplus c) \forall a, b, c \in S$. Intuitively, *associativity* means that the value of an expression containing more than one instance of a single binary associative operator is independent of evaluation order as long as the sequence of the operands is unchanged. In other words, parentheses are unnecessary and rearranging the parentheses in such an expression does not change its value.

Notice that both parses of the expression $x + y * z$ are the same until line 8, where a decision must be made to shift or reduce. The first parse shifts while the second reduces. Both lead to successful parses. However, if we evaluate the expression while parsing it, each parse leads to different results. One way to evaluate a mathematical expression while parsing it is to emit the mathematical operation when reducing. For instance, in step 12 of the first parse, when we reduce $\langle \text{expr} \rangle * \langle \text{expr} \rangle$ to $\langle \text{expr} \rangle$, we can compute $y * z$. Similarly, in step 13 of that same parse, when we reduce $\langle \text{expr} \rangle + \langle \text{expr} \rangle$ to $\langle \text{expr} \rangle$, we can compute $x + \langle \text{the result computed in step 12} \rangle$. This interpretation [i.e., $x + (y * z)$] is desired because in mathematics multiplication has higher precedence than addition. Now consider the second parse. In step 8 of that parse, when we (prematurely) reduce $\langle \text{expr} \rangle + \langle \text{expr} \rangle$ to $\langle \text{expr} \rangle$, we compute $x + y$. Then in step 13, when we reduce $\langle \text{expr} \rangle * \langle \text{expr} \rangle$ to $\langle \text{expr} \rangle$, we compute $\langle \text{the result computed in step 8} \rangle * z$. This interpretation [i.e., $(x + y) * z$] is obviously not desired. If we shift at step 8, multiplication has higher precedence

than addition (desired). If we reduce at step 8, addition has higher precedence than multiplication (undesired). Therefore, we prefer the first parse. These two parses exhibit a *shift-reduce conflict*. If we shift at step 8, then multiplication has higher precedence than addition (which is the desired semantics). If we reduce at step 8, then addition has higher precedence (which is the undesired semantics).

The possibility of a *reduce-reduce* conflict also exists. Consider the following grammar:

$$\begin{aligned} (r_1) \quad & \langle \text{expr} \rangle ::= \langle \text{term} \rangle \\ (r_2) \quad & \langle \text{expr} \rangle ::= \langle \text{id} \rangle \\ (r_3) \quad & \langle \text{term} \rangle ::= \langle \text{id} \rangle \\ (r_4) \quad & \langle \text{id} \rangle ::= x \mid y \mid z \end{aligned}$$

and a bottom-up parse of the expression x :

$$\begin{aligned} . x & \quad (\text{shift}) \\ x . & \quad (\text{reduce } r_4) \\ \langle \text{id} \rangle . & \quad (\text{reduce } r_2 \text{ or } r_3 \text{ here?}) \end{aligned}$$

2.8.2 Parse Trees

The underlying source of shift-reduce and reduce-reduce conflicts is an *ambiguous grammar*. A grammar is *ambiguous* if there exists a sentence that can be parsed in more than one way. A parse of a sentence can be graphically represented using a parse tree. A *parse tree* is a tree whose root is the start symbol of the grammar, non-leaf vertices are non-terminals, and leaves are terminals, where the structure of the tree represents the conformity of the sentence to the grammar. A parse tree is fully expanded. Specifically, it has no leaves that are non-terminals and all of its leaves are terminals that, when collected from left to right, constitute the expression whose parse it represents. Thus, a grammar is *ambiguous* if we can construct more than one parse tree for the same sentence from the language defined by the grammar. Figure 2.3 gives parse trees for the expression $x + y * z$ derived from the four-function calculator grammar in Section 2.6. The left tree represents the first parse and the right tree represents the second parse. The existence of these trees proves that the grammar is ambiguous. The last grammar in Section 2.8.1 is also

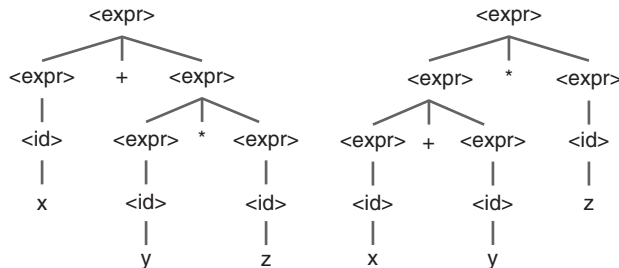


Figure 2.3 Two parse trees for the expression $x + y * z$.

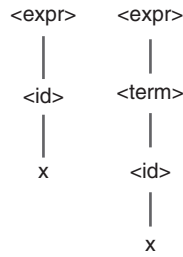


Figure 2.4 Parse trees for the expression x .

ambiguous; a proof of ambiguity exists in Figure 2.4, which contains two parse trees for the expression x .

Ambiguity is a term used to describe a grammar, whereas a shift-reduce conflict and a reduce-reduce conflict are phrases used to describe a particular parse. However, each concept is a different side of the same coin. If a grammar is ambiguous, a bottom-up parse of a sentence in the language the grammar defines will exhibit either a shift-reduce or reduce-reduce conflict, and vice versa.

Thus, proving a grammar is ambiguous is a straightforward process. All we need to do is build two parse trees for the same expression. Much more difficult, by comparison, is proving that a grammar is unambiguous.

It is important to note that a parse tree is not a derivation, or vice versa. A derivation illustrates how to *generate* a sentence. A parse tree illustrates the opposite—how to *recognize* a sentence. However, both prove a sentence is in a language (Table 2.7). Moreover, while multiple derivations of a sentence (as illustrated in Section 2.6) are not a problem, having multiple parse trees for a sentence is a problem—not from a recognition standpoint, but rather from an interpretation (i.e., meaning) perspective. Consider Table 2.8, which contains four sentences from the four-function calculator grammar in Section 2.6. While the

A derivation	<i>generates</i>	a sentence in a formal language.
A parse tree	<i>recognizes</i>	a sentence in a formal language.
Both	<i>prove</i>	a sentence is in a formal language.

Table 2.7 The Dual Use of Grammars: For *Generation* (Constructing a Derivation) and *Recognition* (Constructing a Parse Tree)

Sentence	Derivation(s)	Parse Tree(s)	Semantics
132	multiple	one	one: 132
$1 + 3 + 2$	multiple	multiple	one: 6
$1 + 3 * 2$	multiple	multiple	multiple: 7 or 8
$6 - 3 - 2$	multiple	multiple	multiple: 1 or 5

Table 2.8 Effect of Ambiguity on Semantics

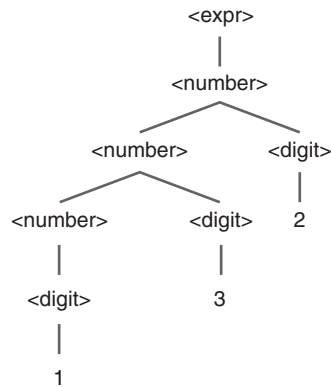


Figure 2.5 Parse tree for the expression 132.

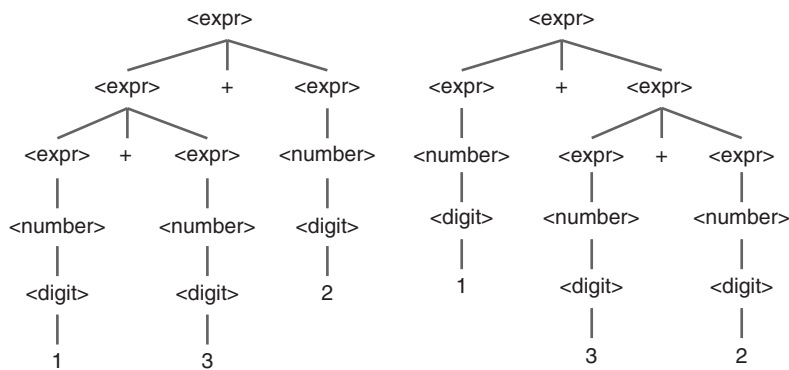


Figure 2.6 Parse trees for the expression $1 + 3 + 2$.

first sentence 132 has multiple derivations, it has only one parse tree (Figure 2.5) and, therefore, only one meaning. The second expression, $1 + 3 + 2$, in contrast, has multiple derivations and multiple parse trees. However, those parse trees (Figure 2.6) all convey the same meaning (i.e., 6). The third expression, $1 + 3 * 2$, also has multiple derivations and parse trees (Figure 2.7). However, its parse trees each convey a different meaning (i.e., 7 or 8). Similarly, the fourth expression, $6 - 3 - 2$, has multiple derivations and parse trees (Figure 2.8), and those parse trees each have different interpretations (i.e., 1 or 5). The last three rows of Table 2.8 show the grammar to be ambiguous even though the ambiguity manifested in the expression $1 + 3 + 2$ is of no consequence to interpretation. The third expression demonstrates the need for rules establishing precedence among operators, and the fourth expression illustrates the need for rules establishing how each operator associates (left-to-right or right-to-left).

Bear in mind, that we are addressing semantics using a formalism intended for syntax. We are addressing semantics using formalisms and techniques reserved for syntax primarily because we do not have easily implementable methods

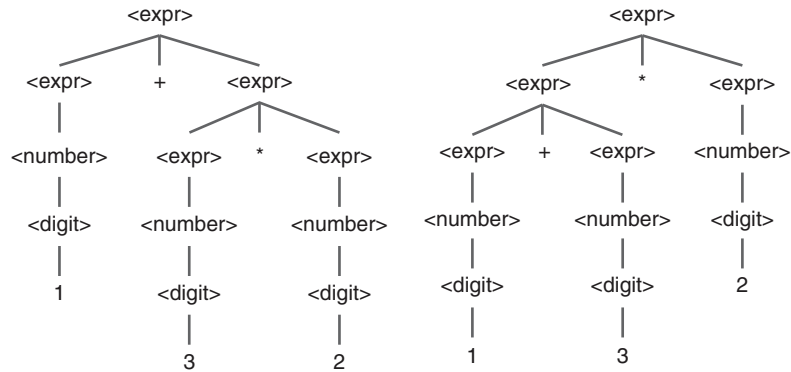


Figure 2.7 Parse trees for the expression $1 + 3 * 2$.

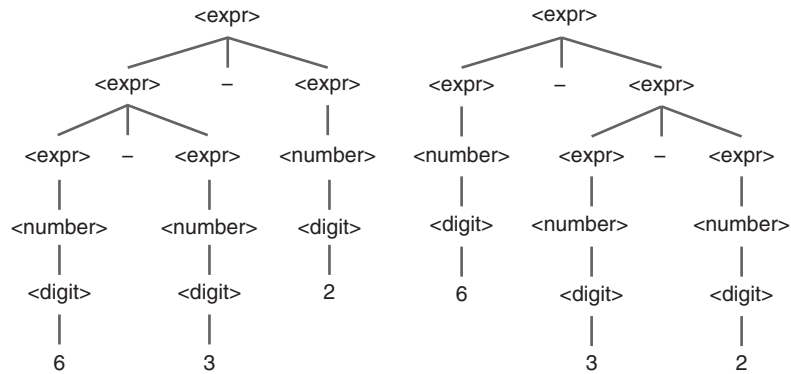


Figure 2.8 Parse trees for the expression $6 - 3 - 2$.

for dealing with context, which is necessary to effectively address semantics, in computer systems. By definition, *context-free* grammars are not intended to model *context*. However, the semantics we address through syntactic means—namely, precedence and associativity—are not dependent on context. In other words, multiplication does not have higher precedence than addition in some contexts and vice versa in others (though it could, since we are defining the language⁶). Similarly, subtraction does not associate left-to-right in some contexts and right-to-left in others. Therefore, all we need to do is make a decision for each and implement the decision.

Typically semantic rules such as precedence and associativity are specified in English (in the absence of formalisms to encode semantics easily and succinctly) in the programming manual of a particular programming language (e.g., $*$ has higher precedence than $+$ and $-$ associates left-to-right). Thus, English is one way to specify semantic rules. However, English itself is ambiguous. Therefore, when the ambiguity—in the formal language, not English—is not dependent on context, as

6. In the programming language APL, addition has higher precedence than multiplication.

in the case here with precedence and associativity, we can modify the grammar so that the ambiguity is removed, making the meaning (or semantics) determinable from the grammar (syntax). When ambiguity is dependent on context, grammar disambiguation to force one interpretation is not possible because you actually want more than one interpretation, though only one per context. For instance, the English sentence “Time flies like an arrow” can be parsed multiple ways. It can be parsed to indicate that there are creatures called “time flies,” which really like arrows (i.e., *<adjective> <noun> <verb> <article> <noun>*), or metaphorically (i.e., *<noun> <verb> <preposition> <article> <noun>*). English is a language with an ambiguous grammar. How can we determine intended meaning? We need the surrounding context provided by the sentences before and after this sentence. Consider parsing the sentence “Mary saw the man on the mountain with a telescope.”, which also has multiple interpretations corresponding to the different parses of it. This sentence has *syntactic ambiguity*, meaning that the same sentence can be diagrammed (or parsed) in multiple ways (i.e., it has multiple syntactic structures). “They are moving pictures.” and “The duke yet lives that Henry shall depose.”⁷ are other examples of sentences with multiple interpretations.

English sentences can also exhibit *semantic ambiguity*, where there is only one syntactic structure (i.e., parse), but the individual words can be interpreted differently. An underlying source of these ambiguities is the presence of *polysemes*—a word with one spelling and pronunciation, but different meanings (e.g., book, flies, or rush). Polysemes are the opposite of *synonyms*—different words with one meaning (e.g., peaceful and serene). Polysemes that are different parts of speech (e.g., book, flies, or rush) can cause syntactic ambiguity, whereas polysemes that are the same part of speech (e.g., mouse) can cause semantic ambiguity. Note that not all sentences with syntactic ambiguity contain a polyseme (e.g., “They are moving pictures.”). For summaries of these concepts, see Tables 2.9 and 2.10.

Similarly, in programming languages, the source of a semantic ambiguity is not always a syntactic ambiguity. For instance, consider the expression `(Integer) -a` on line 5 of the following Java program:

```

1  class SemanticAmbiguity {
2      public static void main(String args[]) {
3          int a = 1;
4          int Integer = 5;
5          int b = (Integer)-a;
6          System.out.println(b); // prints 4, not -1
7          b = (Integer)(-a);
8          System.out.println(b); // prints -1, not 4
9      }
10 }
```

The expression `(Integer) -a` (line 5) has only one parse tree given the grammar of a four-function calculator presented this section (assuming `Integer` is an *<id>*) and, therefore, is syntactically unambiguous. However, that expression has multiple interpretations in Java: (1) as a *subtraction*—the variable `Integer`

7. *Henry VI* by William Shakespeare.

Concept	Syntactic Structure(s)	Meaning	Example
<i>Syntactic ambiguity</i>	multiple	multiple	They are moving pictures.
<i>Semantic ambiguity</i>	one	multiple	The mouse was right on my computer.

Table 2.9 Syntactic Ambiguity Vis-à-Vis Semantic Ambiguity

Term	Spelling	Pronunciation	Meaning	Example(s)
<i>Polysemes</i>	same	same	different	book, flies, or rush
<i>Homonyms</i>				
<i>Homophones</i>	different	same	different	knight/night
<i>Homographs</i>	same	different	different	close or wind
<i>Synonyms</i>	different	different	same	peaceful/serene

Table 2.10 Polysemes, Homonyms, and Synonyms

minus the variable a , which is 4, and (2) as a *type cast*—type casting the value $-a$ (or -1) to a value of type `Integer`, which is -1 . Table 2.11 contains sentences from both natural and programming languages with various types of ambiguity, and demonstrates the interplay between those types. For example, a sentence without syntactic ambiguity can have semantic ambiguity; and a sentence without semantic ambiguity can have syntactic ambiguity.

We have two options for dealing with an ambiguous grammar, but both have disadvantages. First, we can state disambiguation rules in English (i.e., attach notes to the grammar), which means we do not have to alter (i.e., lengthen) the grammar, but this comes at the expense of being less formal (by the use of English). Alternatively, we can disambiguate the grammar by revising it, which is a more formal approach than the use of English, but this inflates the number of production rules in the grammar. Disambiguating a grammar is not always possible. The existence of context-free languages for which no unambiguous context-free grammar exists has been proven (in 1961 with *Parikh's theorem*). These languages are called *inherently ambiguous languages*.

Sentence	Ambiguity		
	Lexical	Syntactic	Semantic
flies	✓	✓	✓
Time flies like an arrow.	✓	✓	✓
They are moving pictures.	×	×	✓
*	✓	✓	✓
$1+3+2$	×	✓	×
$1+3*2$	✓	✓	✓
<code>(Integer) - a</code>	×	×	✓

Table 2.11 Interplay Between and Interdependence of Types of Ambiguity

2.9 Grammar Disambiguation

Here, “having higher precedence” means “occurring lower in the parse tree” because expressions are evaluated bottom-up. In general, grammar disambiguation involves introducing additional non-terminals to prevent a sentence from being parsed multiple ways. To remove the ambiguity caused by (the lack of) operator precedence, we introduce new steps (i.e., non-terminals) in the non-terminal cascade so that multiplications are always lower than additions in the parse tree. Recall that we desire part of the meaning (or semantics) to be determined from the grammar (or syntax).

2.9.1 Operator Precedence

Consider the following updated grammar, which addresses precedence:

$$\begin{aligned}
 \langle \text{expr} \rangle &::= \langle \text{expr} \rangle + \langle \text{expr} \rangle \\
 \langle \text{expr} \rangle &::= \langle \text{expr} \rangle - \langle \text{expr} \rangle \\
 \langle \text{expr} \rangle &::= \langle \text{term} \rangle \\
 \langle \text{term} \rangle &::= \langle \text{term} \rangle * \langle \text{term} \rangle \\
 \langle \text{term} \rangle &::= \langle \text{term} \rangle / \langle \text{term} \rangle \\
 \langle \text{term} \rangle &::= (\langle \text{expr} \rangle) \\
 \langle \text{term} \rangle &::= (\langle \text{id} \rangle) \\
 \langle \text{term} \rangle &::= \langle \text{number} \rangle
 \end{aligned}$$

With this grammar it is no longer possible to construct two parse trees for the expression $x + y * z$. The expression $x + y * z$, by virtue of being parsed using this revised grammar, will always be interpreted as $x + (y * z)$. However, while the example grammar addresses the issue of precedence, it remains ambiguous because it is still possible to use it to construct two parse trees for the expression $6 - 3 - 2$ since it does not address associativity (Figure 2.8). Recall that associativity comes into play when dealing with operators with the same precedence. Subtraction is left-associative [e.g., $6 - 3 - 2 = (6 - 3) - 2 = 1$], while unary minus is right-associative [e.g., $- - - 6 = -(-(-6))$]. Associativity is mute with certain operators, including addition [e.g., $1 + 3 + 2 = (1 + 3) + 2 = 1 + (3 + 2) = 6$], but significant with others, including subtraction and unary minus. Theoretically, addition associates either left or right with the same result. However, when addition over floating-point numbers is implemented in a computer system, associativity is significant because left- and right-associativity can lead to different results. Thus, the grammar is still ambiguous for the sentences $1 + 3 + 2$ and $6 - 3 - 2$, although the former does not cause problems because both parses result in the same interpretation.

2.9.2 Associativity of Operators

Consider the following updated grammar, which addresses precedence *and* associativity:

$$\begin{aligned}
\langle \text{expr} \rangle &::= \langle \text{expr} \rangle + \langle \text{term} \rangle \\
\langle \text{expr} \rangle &::= \langle \text{expr} \rangle - \langle \text{term} \rangle \\
\langle \text{expr} \rangle &::= \langle \text{term} \rangle \\
\langle \text{term} \rangle &::= \langle \text{term} \rangle * \langle \text{factor} \rangle \\
\langle \text{term} \rangle &::= \langle \text{term} \rangle / \langle \text{factor} \rangle \\
\langle \text{term} \rangle &::= \langle \text{factor} \rangle \\
\langle \text{factor} \rangle &::= (\langle \text{expr} \rangle) \\
\langle \text{factor} \rangle &::= (\text{id}) \\
\langle \text{factor} \rangle &::= \langle \text{number} \rangle
\end{aligned}$$

In disambiguating the grammar for associativity, we follow the same thematic process as we used earlier: Obviate multiple parse trees by adding another level of indirection through the introduction of a new non-terminal. If we want an operator to be left-associative, then we write the production rule for that operator in a left-recursive manner because left-recursion leads to left-associativity. Similarly, if we want an operator to be right-associative, then we write the production rule for that operator in a right-recursive manner because right-recursion results in right-associativity. Since subtraction is a left-associative operator, we write the production rule as $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle - \langle \text{term} \rangle$ (i.e., left-recursive) rather than $\langle \text{expr} \rangle ::= \langle \text{term} \rangle - \langle \text{expr} \rangle$ (i.e., right-recursive). The same holds for division. Since addition and multiplication are non-associative operators, we write the production rules dealing with those operators in a left-recursive manner for consistency. Therefore, the final non-ambiguous grammar is that shown previously.

2.9.3 The Classical Dangling `else` Problem

The dangling `else` problem is a classical example of grammar ambiguity in programming languages: In the absence of curly braces for disambiguation, when we have an `if-else` statement such as `if <expr1> if <expr2> <stmt1> else <stmt2>`, the `if` to which the `else` is associated is ambiguous. In other words, without a semantic rule, the statement can be interpreted in the following two ways:

```

if expr1
  if expr2
    stmt1
  else
    stmt2

```

```

if expr1
  if expr2
    stmt1
else
  stmt2

```

Indentation is used to indicate to which `if` the `else` is *intended* to be associated. Of course, in free-form languages, indentation has no bearing on program semantics.

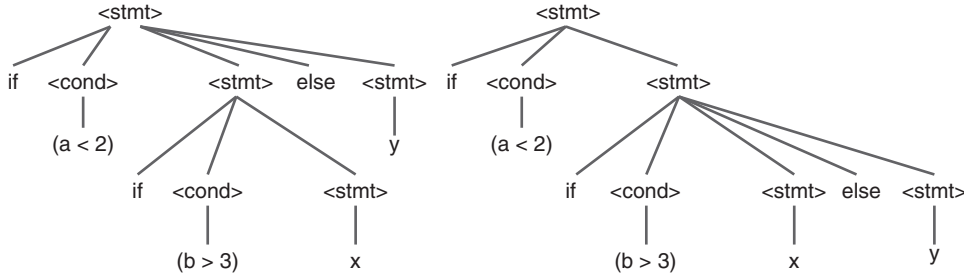


Figure 2.9 Parse trees for the sentence `if (a < 2) if (b > 3) x else y`. (left) Parse tree for an `if-(if)-else` construction. (right) Parse tree for an `if-(if-else)` construction.

In C, the semantic rule is that an `else` associates with the closest unmatched `if` and, therefore, the first interpretation is used.

Consider the following grammar for generating `if-else` statements:

```

<stmt> ::= if <cond> <stmt>
<stmt> ::= if <cond> <stmt> else <stmt>

```

Using this grammar, we can generate the following statement (save for the comment):

```

if (a < 2)
  if (b > 3)
    x = 4;
  else /* associates with which if above ? */
    y = 5;

```

for which we can construct two parse trees (Figure 2.9) proving that the grammar is ambiguous. Again, since formal methods for modeling semantics are not easily implementable, we need to revise the grammar (i.e., syntax) to imply the desired meaning (i.e., semantics). We can do that by disambiguating this grammar so that it is capable of generating `if` sentences that can only be parsed to imply that any `else` associates with the nearest unmatched `if` (i.e., parse trees of the form shown on the right side of Figure 2.9). We leave it as an exercise to develop an unambiguous grammar to solve the dangling `else` problem (Conceptual Exercise 2.10.25).

Notice that while semantics (e.g., precedence and associativity) can sometimes be reasonably modeled using context-free grammars, which are devices for modeling the *syntactic structure* of language, context-free grammars can always be used to model the *lexical structure* (or *lexics*) of language, since any regular language can be modeled by a context-free grammar. For instance, embedded into the first grammar of a four-function calculator presented in this section is the lexics of the numbers:

$$\begin{aligned}
\langle \text{number} \rangle &::= \langle \text{number} \rangle \langle \text{digit} \rangle \\
\langle \text{number} \rangle &::= \langle \text{digit} \rangle \\
\langle \text{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

Thus, in the four-function calculator grammar containing these productions, the token structure (of numbers) and the syntactic structure of the expressions are inseparable. Alternatively, we could have used the regular expression $(0+1+\dots+8+9)(0+1+\dots+8+9)^*$ to define the lexics and used a simpler rule in the context-free grammar:

$$\langle \text{number} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid \dots \mid 2^{31}-2 \mid 2^{31}-1$$

2.10 Extended Backus–Naur Form

Extended Backus–Naur Form (EBNF) includes the following syntactic extensions to BNF.

- $|$ means “alternation.”
- $[X]$ means “ X is optional.”
- $\{X\}^*$ means “zero or more of X .”
- $\{X\}^+$ means “one or more of X .”
- $\{X\}^{*(c)}$ means “zero or more of X separated by C s.”
- $\{X\}^{+(c)}$ means “one or more of X separated by C s.”

Note that we have already encountered the extension to BNF for alternation (using $|$). Consider the following context-free grammar defined in BNF:

$$\begin{aligned}
\langle \text{symbol-expr} \rangle &::= x \\
\langle \text{symbol-expr} \rangle &::= y \\
\langle \text{symbol-expr} \rangle &::= z \\
\langle \text{symbol-expr} \rangle &::= (\langle s\text{-list} \rangle) \\
\langle s\text{-list} \rangle &::= \langle s\text{-list} \rangle, \langle \text{symbol-expr} \rangle \\
\langle s\text{-list} \rangle &::= \langle \text{symbol-expr} \rangle
\end{aligned}$$

which can be used to derive the following sentences: x , (x, y, z) , $((x))$, and $((x)), ((y), (z)))$. We can reexpress this grammar in EBNF using alternation as follows:

$$\begin{aligned}
\langle \text{symbol-expr} \rangle &::= x \mid y \mid z \mid (\langle s\text{-list} \rangle) \\
\langle s\text{-list} \rangle &::= \langle s\text{-list} \rangle, \langle \text{symbol-expr} \rangle \mid \langle \text{symbol-expr} \rangle
\end{aligned}$$

We can express r_2 more concisely using the extension for an optional item:

$$\begin{aligned}
\langle \text{symbol-expr} \rangle &::= x \mid y \mid z \mid (\langle s\text{-list} \rangle) \\
\langle s\text{-list} \rangle &::= [\langle s\text{-list} \rangle,] \langle \text{symbol-expr} \rangle
\end{aligned}$$

As another example, consider the following grammar defined in BNF:

$$\begin{aligned}
\langle \text{arglist} \rangle &::= \langle \text{arg} \rangle, \langle \text{arg} \rangle \\
\langle \text{arg} \rangle &::= \langle \text{arglist} \rangle
\end{aligned}$$

It can be rewritten in EBNF as a single rule:

$$\langle arglist \rangle ::= \langle arg \rangle, \langle arg \rangle \{, \langle arg \rangle \}^*$$

and can be simplified further as

$$\langle arglist \rangle ::= \langle arg \rangle, \langle arg \rangle \{ \langle arg \rangle \}^*(,)$$

or expressed alternatively as

$$\langle arglist \rangle ::= \langle arg \rangle, \{ \langle arg \rangle \}^+(,)$$

These extensions are intended for ease of grammar definition. Any grammar defined in EBNF can be expressed in BNF. Thus, these shortcuts are simply syntactic sugar. In summary, a context-free language (which is a type of formal language) is generated by a context-free grammar (which is a type of formal grammar) and recognized by a pushdown automaton (which is a model of computation).

Conceptual Exercises for Sections 2.4–2.10

Exercise 2.10.1 Define a *regular grammar* in BNF for the language of Conceptual Exercise 2.3.1.

Exercise 2.10.2 Define a *regular grammar* in EBNF for the language of Conceptual Exercise 2.3.1.

Exercise 2.10.3 Define a *regular grammar* in BNF for the language of Conceptual Exercise 2.3.3.

Exercise 2.10.4 Define a *regular grammar* in EBNF for the language of Conceptual Exercise 2.3.3.

Exercise 2.10.5 Define a *regular grammar* in BNF for the language of Conceptual Exercise 2.3.4.

Exercise 2.10.6 Define a *regular grammar* in EBNF for the language of Conceptual Exercise 2.3.4.

Exercise 2.10.7 Define a grammar G , where G is not regular but defines a *regular language* (i.e., one that can be denoted by a *regular expression*).

Exercise 2.10.8 Express the *regular expression* $hw(1+2+\dots+8+9)(0+1+2+\dots+8+9)^*$ as a *regular grammar*.

Exercise 2.10.9 Express the *regular expression* $hw(1+2+\dots+8+9)(0+1+2+\dots+8+9)^*$ as a *context-free grammar*.

Exercise 2.10.10 Notice that the grammar of a four-function calculator presented in Section 2.6 is capable of generating numbers containing one or more leading

0s (e.g., 001 and 0001931), which four-function calculators are typically unable to produce. Revise this grammar so that it is unable to generate numbers with leading zeros, save for 0 itself.

Exercise 2.10.11 Reduce the number of production rules in the grammar of a four-function calculator presented in Section 2.6. In particular, consolidate rules r_1 – r_4 into two rules by adding a new non-terminal $\langle operator \rangle$.

Exercise 2.10.12 Describe in English, as precisely as possible, the language defined by the following grammar:

$$\begin{aligned} T &\rightarrow ab \mid ba \\ T &\rightarrow abT \mid baT \\ T &\rightarrow aTb \mid bTa \\ T &\rightarrow aTbT \mid bTaT \end{aligned}$$

where T is a non-terminal and a and b are terminals.

Exercise 2.10.13 Prove that the grammar in Conceptual Exercise 2.10.12 is *ambiguous*.

Exercise 2.10.14 Consider the following grammar in EBNF:

$$\begin{aligned} \langle expr \rangle &::= \langle expr \rangle + \langle expr \rangle \mid \langle term \rangle \\ \langle term \rangle &::= \langle term \rangle * \langle term \rangle \mid \langle expr \rangle \mid id \end{aligned}$$

where $\langle expr \rangle$ and $\langle term \rangle$ are non-terminals and $+$, $*$, and id are terminals.

(a) Prove that this grammar is *ambiguous*.

(b) Modify this grammar so that it is *unambiguous*.

(c) Define an *unambiguous* version of this grammar containing *only two non-terminals*.

Exercise 2.10.15 Prove that the following grammar defined in EBNF is *ambiguous*:

$$\begin{aligned} (r_1) \quad \langle symbol\text{-}expr \rangle &::= x \mid y \mid z \mid (\langle s\text{-}list \rangle) \\ (r_2) \quad \langle s\text{-}list \rangle &::= [\langle s\text{-}list \rangle,] \langle symbol\text{-}expr \rangle \\ (r_3) \quad \langle s\text{-}list \rangle &::= [\langle symbol\text{-}expr \rangle,] \langle symbol\text{-}expr \rangle \end{aligned}$$

where $\langle symbol\text{-}expr \rangle$ and $\langle s\text{-}list \rangle$ are non-terminals; x , y , z , $($, and $)$ are terminals; and $\langle symbol\text{-}expr \rangle$ is the start symbol.

Exercise 2.10.16 Does removing rule r_3 from the grammar in Conceptual Exercise 2.10.15 eliminate the ambiguity from the grammar? If not, prove that the grammar with r_3 removed is still ambiguous.

Exercise 2.10.17 Define a grammar for a language L consisting of strings that have n copies of the letter a followed by the same number of copies of the letter b , where $n > 0$. Formally, $L = \{a^n b^n \mid n > 0 \text{ and } \Sigma = \{a, b\}\}$, where x^n means “ n copies of

x .” For instance, the strings `ab`, `aaaabbbb`, and `aaaaaaaaabbbbbbbb` are sentences in the language, but the strings `a`, `abb`, `ba`, and `aaabb` are not. Is this language *regular*? Explain.

Exercise 2.10.18 Define an *unambiguous, context-free grammar* for a language L of *palindromes* of binary numbers. A *palindrome* is a string that reads the same forward as backward. For example, the strings `0`, `1`, `00`, `11`, `101`, and `100101001` are palindromes, while the strings `10`, `01`, and `10101010` are not. The empty string ϵ is not in this language. Formally, $L = \{xx^r \mid x \in \{0, 1\}^*\}$, where x^r means “a reversed copy of x .”

Exercise 2.10.19 Matching syntactic entities (e.g., parentheses, brackets, or braces) is an important aspect of many programming languages. Define a *context-free grammar* capable of generating only balanced strings of (nested or flat) matched parentheses. The empty string ϵ is not in this language. For instance, the strings `()`, `()()`, `(( ))`, `(( ))()`, and `(( ))()` are sentences in this language, while the strings `)(`, `)()`, `)()()`, `(( ))()`, `(( ))()`, and `(( ))()` are not. Note that *not* all strings with the same number of open and close parentheses are in this language. For example, the strings `)()` and `)()()` are not sentences in this language. State whether your grammar is ambiguous and, if it is ambiguous, prove it.

Exercise 2.10.20 Define an *unambiguous, context-free grammar* for the language of Exercise 2.10.19.

Exercise 2.10.21 Define a *context-free grammar* for a language L of binary numbers that contain the same number of 0s and 1s. Formally, $L = \{x \mid x \in \{0, 1\}^* \text{ and the number of 0s in } x \text{ equals the number of 1s in } x\}$. For instance, the strings `01`, `10`, `0110`, `1010`, `011000100111`, and `000001111011` are sentences in the language, while the strings `0`, `1`, `00`, `11`, `1111000`, `01100010011`, and `000001111011` are not. The empty string ϵ is not in this language. Indicate whether your grammar is ambiguous and, if it is ambiguous, prove it.

Exercise 2.10.22 Solve Exercise 2.10.21 with an *unambiguous grammar*.

Exercise 2.10.23 Rewrite the grammar in Section 2.9.3 in EBNF.

Exercise 2.10.24 The following grammar for `if–else` statements has been proposed to eliminate the dangling `else` ambiguity (Aho, Sethi, and Ullman 1999, Exercise 4.5, p. 268):

```

<stmt> ::= if <expr> <stmt> | <matched_stmt>
<matched_stmt> ::= if <expr> <matched_stmt> else <stmt>
<matched_stmt> ::= <other>

```

where the non-terminal `<other>` generates some non-`if` statement such as a `print` statement. Prove that this grammar is still *ambiguous*.

Exercise 2.10.25 Define an *unambiguous grammar* to remedy the dangling `else` problem (Section 2.9.3).

Exercise 2.10.26 Surprisingly enough, the abilities of programmers have historically had little influence on programming language design and implementation, despite programmers being the primary users of programming languages! For instance, the ability to nest comments is helpful when a programmer desires to comment out a section of code that may already contain a comment. However, the designers of C decided to forbid nesting comments. That is, comments *cannot* nest in C. As a consequence, the following code is not syntactically valid in C:

```

1  /* the following function contains a bug;
2     I'll just comment it out for now.
3  void f() {
4     /* an integer x */
5     int x;
6     ...
7  }
8  */

```

Why did the designers of C decide to forbid nesting comments?

Exercise 2.10.27 Give a specific example of *semantics* in programming languages not mentioned in this chapter.

Exercise 2.10.28 Can a language whose sentences are all sets from an *infinite* universe of items be defined with a *context-free grammar*? Explain.

Exercise 2.10.29 Can a language whose sentences are all sets from a *finite* universe of items be defined with a *context-free grammar*? Explain.

Exercise 2.10.30 Consider the language L of binary strings where the first half of the string is identical to the second half (i.e., all sentences have even length). For instance, the strings 11, 0000, 0101, 1010, 010010, 101101, and 11111111, are sentences in the language, but the strings 0110 and 1100 are not. Formally, $L = \{xx \mid x \in \{0, 1\}^*\}$. Is this language *context-free*? If so, give a *context-free grammar* for it. If not, state why not.

2.11 Context-Sensitivity and Semantics

Context-free grammars, by definition, cannot represent context in language. A classical example of context-sensitivity in English is “the first letter of a sentence must be capitalized.” A *context-sensitive grammar*⁸ for this property of English sentences is:

$\langle \text{sentence} \rangle$	$\rightarrow$	$\langle \text{start} \rangle \langle \text{article} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle \langle \text{adverb} \rangle$.
$\langle \text{start} \rangle \langle \text{article} \rangle$	$\rightarrow$	A An The
$\langle \text{article} \rangle$	$\rightarrow$	a an the

8. Note that the use of the words *-free* and *-sensitive* in the names of formal grammars is inconsistent. The *-free* in context-free grammar indicates what such a grammar is *unable* to model—namely, context. In contrast, the *-sensitive* in context-sensitive grammar indicates what such a grammar *can* model.

In a context-sensitive grammar, the left-hand side of a production rule is not limited to one non-terminal, as is the case in context-free grammars. In this example, the production rule “ $\langle \text{article} \rangle \rightarrow A \mid An \mid The$ ” only applies in the context of $\langle \text{start} \rangle$ to the left of $\langle \text{article} \rangle$; that is, the non-terminal $\langle \text{start} \rangle$ provides the context for the application of the rule.

The pattern to which the production rules of a context-sensitive grammar must adhere are less restrictive than that of a context-free grammar. The productions of a context-sensitive grammar may have more than one non-terminal on the left-hand side. Formally, a grammar is a *context-sensitive grammar* if and only if every production rule is in the form:

$$\alpha X \beta \rightarrow \alpha \gamma \beta$$

where $X \in V$ and $\alpha, \beta, \gamma \in (\Sigma \cup V)^*$, and X can be replaced with γ only in the context of α to its left and β to its right. The strings α and β may be empty in the productions of a context-sensitive grammar, but $\gamma \neq \epsilon$. However, the rule $S \rightarrow \epsilon$ is permitted as long as S does not appear on the right-hand side of any production.

Context and semantics are often confused. Recall that semantics deals with the meaning of a sentence. Context can be used to *validate* or *discern* the meaning of a sentence. Context can be used in two ways:

- **Determine semantic validity.** A classical example of context-sensitivity in programming languages is “a variable must be declared before it is used.” For instance, while the following C program is syntactically valid, context reveals that it is not semantically valid because the variable y is referenced, but never declared:

```
int main() {
    int x;
    y = 1;
}
```

Even if all referenced variables are declared, context may still be necessary to identify type mismatches. For instance, consider the following C++ program:

```
1 int main() {
2     int x;
3     bool y;
4
5     x = 1;
6     y = false;
7     x = y;
8 }
```

Again, while this program is syntactically correct, it is not semantically valid because of the assignment of the value of a variable of one type to a variable of a different type (line 6). We need methods of *static semantics* (i.e., before run-time) to address this problem. We can generate semantically invalid programs from a context-free grammar because the production rules of a context-free grammar always apply, regardless of the context in which

a non-terminal on the left-hand side appears; hence, the rules are called context-free.

- **Disambiguate semantic validity.** Another example of context-sensitivity in programming languages is the `*` operator in C. Its meaning is dependent upon the context in which it is used. It can be used (1) as the multiplication operator (e.g., `x*3`); (2) as the pointer dereferencing operator (e.g., `*ptr`); and (3) in the declaration of pointer types (e.g., `int* ptr`). Without context, the semantics of the expression `x* y` are ambiguous. If we see the declarations `int x=1, y=2;` immediately preceding this expression, the meaning of the `*` is multiplication. However, if the statement `typedef int x;` precedes the expression `x* y`, it declares a pointer to an `int`.

Formalisms, including context-sensitive grammars, for dealing with these and other issues of semantics in programming languages are not easily implementable. Context-free grammars lend themselves naturally to the implementation of parsers (as we see in Chapter 3); context-sensitive grammars do not and, therefore, are not helpful in parser implementation. Thus, while C, Python, and Scheme are context-sensitive languages, the parser for them is implemented using a context-free grammar.

A practical approach to modeling context in programming languages is to infuse context, where practically possible, into a context-free grammar—that is, to include additional production rules to help (brute-)force the syntax to imply the semantics.⁹ This approach involves designing the context-free production rules in such a way that they cannot generate a semantically invalid program. We used this approach previously to enforce proper operator precedence and associativity.

Applying this approach to capture more sophisticated semantic rules, including the requirement that variables must be declared prior to use, leads to an inordinately large number of production rules; consequently it is often unreasonable and impractical. For instance, consider the determination of whether a collection of items is a set (i.e., an unordered collection without duplicates). That determination requires context. In particular, to determine if an element disqualifies the collection from being a set, we must examine the other items in the collection (i.e., the context). If the universe from which the items in the collection are drawn is finite, we can simply enumerate all possible sets from that universe. Such an enumeration results in not only a context-free grammar, but also a regular grammar. However, that approach can involve a large number of production rules. A device called an *attribute grammar* is an extension to a context-free grammar that helps bridge the gap between content-free and context-sensitive grammars, while being practical for use in language implementation (Section 2.14).

While we encounter semantics of programming languages throughout this text, we briefly comment on formal semantics here. There are two types of

9. Both approaches—use of context-sensitive grammar and use of a context-free grammar with many rules modeling the context—model context in a purely *syntactic* way (i.e., without ascribing meaning to the language). For instance, with a context-sensitive grammar or a context-free grammar with many rules to enforce semantic rules for C, it is impossible to generate a program referencing an undeclared variable, and a program referencing an undeclared variable would be *syntactically* invalid.

semantics: static and dynamic. In general, in computing, these terms mean before and during run-time, respectively. An example of *static semantics* is the detection of the use of an undeclared variable or a type incompatibility (e.g., `int x = "this is not an int";`). Attribute grammars can be used for static semantics.

There are three approaches to *dynamic semantics*: operational, denotational, and axiomatic. Operational semantics involves discerning the meaning of a programming construct by exploring the effects of running a program using it. Since an interpreter for a programming language, through its implementation, implicitly specifies the semantics of the language it interprets, running a program through an interpreter is an avenue to explore the operational semantics of the expressions and statements within the program. (Building interpreters for programming languages with a variety of constructs and features is the primary focus of Chapters 10–12.) Consider the English sentence “I chose wisely” which is in the past tense. If we replace the word “chose” with “chos,” the sentence has a lexics error because the substring “chos” is not lexically valid. However, if we replace the word “chose” with “choose,” the sentence is lexically, syntactically, and semantically valid, but in the present tense. Thus, the semantics of the sentence are valid, but unintended. Such a semantic error, like a run-time error in a program, is difficult to detect.

Conceptual Exercises for Section 2.11

Exercise 2.11.1 Give an example of a property in programming languages (other than any of those given in the text) that is *context-sensitive* or, in other words, an example property that is *not* context-free.

Exercise 2.11.2 A *context-sensitive grammar* can express context that a *context-free grammar* cannot model. State what a *context-free grammar* can express that a *regular grammar* cannot model.

Exercise 2.11.3 We stated in this section that sometimes we can infuse context into a *context-free grammar* (often by adding more production rules) even though a context-free grammar has no provisions for representing context. Express the *context-sensitive grammar* given in Section 2.11 enforcing the capitalization of the first character of an English sentence using a *context-free grammar*.

Exercise 2.11.4 Define a *context-free grammar* for the language whose sentences correspond to sets of the elements *a*, *b*, and *c*. For instance, the sentences $\{a\}$, $\{a, b\}$, $\{a, b, c\}$ are in the language, but the sentences $\{a, a\}$, $\{b, a, b\}$, and $\{a, b, c, a\}$ are not.

2.12 Thematic Takeaways

- The identifiers and numbers in programming languages can be described by a *regular grammar*.

- The nested expressions and blocks in programming languages can be described by a *context-free grammar*.
- Neither a regular nor a context-free grammar can describe the rule that a variable must be declared before it is used.
- Grammars are language recognition devices as well as language generative devices.
- An *ambiguous* grammar poses a problem for language recognition.
- Two parse trees for the same sentence from a language are sufficient to prove that the grammar for the language is *ambiguous*.
- Semantic properties, including precedence and associativity, can be modeled in a context-free grammar.

2.13 Chapter Summary

This chapter addresses *constructs* (e.g., regular expressions, grammars, automata) for defining (i.e., denoting, generating, and recognizing, respectively) languages and the *capabilities* (or limitations) of those constructs in relation to programming languages (Table 2.12). A *regular expression* denotes a set of strings—that is, the sentences of the language that the regular expression denotes. *Regular expressions* and *regular grammars* can capture the rules for a valid identifier in a programming language. More generally, regular expressions can model the *lexics* (i.e., lexical structure) of a programming language. *Context-free grammars* can capture the concept of balanced entities nested arbitrarily deep (e.g., parentheses, brackets, curly braces) whose use is pervasive in the syntactic structures (e.g., mathematical expression, `if-else` blocks) of programming languages. More generally, context-free grammars can model the *syntax* (i.e., syntactic structure) of a programming language. (Formally, context-free grammars are expressive enough to define formal languages that require an unbounded amount of memory used in a restricted way [i.e., LIFO] to recognize sentences in those languages.) If a sentence from a language has more than one parse tree, then the grammar for the language is *ambiguous*. Neither regular grammars nor context-free grammars can capture

Formal Language/ Grammar	Modeling Capability	Example Language	PL Analog	PL Code Example
Regular	lexemes	$L(a^*b^*)$	tokens (ids, #s)	<code>index1;17.76</code>
Context-free	balanced pairs	$\{a^n b^n \mid n \geq 0\}$	nested expressions/ blocks	<code>(a*(b+c));if/else</code>
Context-free	palindromes	$\{xx^r \mid x \in \{a,b\}^*\}$	—	—
Context-sensitive	one-to-one mapping	$\{xx \mid x \in \{a,b\}^*\}$	variable declarations and references	<code>int a; a=1;</code>
Context-sensitive	context	$\{a^n b^n c^n \mid n \geq 0\}$	—	—

Table 2.12 Formal Grammar Capabilities Vis-à-Vis Programming Language Constructs (Key: PL = programming language.)

Type	Formal Language	(defined/generated by) Formal Grammar	(recognized by) Automaton (model of computation)	(constraints on) Production Rules
Type-3	regular language	regular grammar	deterministic finite automaton	$X \rightarrow ZY \mid Z$ or $X \rightarrow YZ \mid Z$
Type-2	context-free language	context-free grammar	pushdown automaton	$X \rightarrow \gamma$
Type-1	context-sensitive language	context-sensitive grammar	linear-bounded automaton	$\alpha X \beta \rightarrow \alpha \gamma \beta$
Type-0	recursively enumerable language	unrestricted grammar	Turing machine	$\alpha \rightarrow \beta$

Table 2.13 Summary of Formal Languages and Grammars, and Models of Computation

the rule that a variable must be declared before it is used. However, we can model some semantic properties, including operator precedence and associativity, with a context-free grammar. Thus, not all formal grammars have the same expressive power; likewise, not all automata have the same power to decide if a string is a sentence in a language. (The corollary is that there are limits to computation.) While most programming languages are context-sensitive (because variables often must be declared before they are used), context-free grammars are the theoretical basis for the syntax of programming languages (in both language definition and implementation, as we see in Chapters 3 and 4).

Table 2.13 summarizes each of the progressive four types of formal grammars in the Chomsky Hierarchy; the class of formal language each grammar generates; the type of automaton that recognizes each member of each class of those formal languages; and the constraints on the production rules of the grammars. Regular and context-free grammars are fundamental topics in the study of the formal languages. In our course of study, they are useful for both describing the syntax of and parsing programming languages. In particular, regular and context-free grammars are essential ingredients in scanners and parsers, respectively, which are discussed in Chapter 3.

2.14 Notes and Further Reading

We refer readers to Webber (2008) for a practical, more detailed discussion of formal languages, grammars, and automata theory.

John Backus and Peter Naur are the recipients of the 1977 and 2005 ACM A. M. Turing Awards, respectively, in part, for their contributions to language design (through Fortran and ALGOL 60, respectively) and their contributions of formal methods for the specification of programming languages.

Attribute grammars are a formalism contributed by Donald Knuth, which can be used to capture semantics in a practical way; these grammars are context-free grammars annotated with semantics rules and checks. Knuth is the recipient of the 1974 ACM A. M. Turing Award for contributions to programming language design, including attribute grammars, and to “the art of computer programming”—communicated through his monograph titled *The Art of Computer Programming*.

Chapter 3

Scanning and Parsing

Although mathematical notation undoubtedly possesses parsing rules, they are rather loose, sometimes contradictory, and seldom clearly stated. . . . The proliferation of programming languages shows no more uniformity than mathematics. Nevertheless, programming languages do bring a different perspective. . . . Because of their application to a broad range of topics, their strict grammar, and their strict interpretation, programming languages can provide new insights into mathematical notation.

— Kenneth E. Iverson

ANY implementation of a programming language involves scanning and parsing the source program into a representation that can be subsequently processed (i.e., interpreted or compiled or a combination of both). *Scanning* involves analyzing a program represented as a string to determine whether the atomic lexical units of that string are valid. If so, the process of *parsing* determines whether those lexical units are arranged in a valid order with respect to the grammar of the language and, if so, converts the program into a more easily processable representation.

3.1 Chapter Objectives

- Establish an understanding of *scanning*.
- Establish an understanding of *parsing*.
- Introduce *top-down parsing*.
- Differentiate between *table-driven* and *recursive-descent* top-down parsers.
- Illustrate the natural relationship between a *context-free grammar* and a *recursive-descent parser*.
- Introduce *bottom-up, shift-reduce parsing*.
- Introduce parser-generation tools (e.g., `lex/yacc` and `PLY`).

3.2 Scanning

For purposes of scanning, the valid lexical units of a program are called *lexemes* (e.g., `+`, `main`, `int`, `x`, `h`, `hw`, `hww`). The first step of *scanning* (also referred to as *lexical analysis*) is to parcel the characters (from the alphabet Σ) of the string representing the line of code into lexemes. Lexemes can be formally described by *regular expressions* and *regular grammars*. *Lexical analysis* is the process of determining if a string (typically of a programming language) is lexically valid—that is, if all of the lexical units of the string are lexemes.

Programming languages must specify how the lexical units of a program are delimited. There are a variety of methods that languages use to determine where lexical units begin and end. Most programming languages delimit lexical units using whitespace (i.e., spaces and tabs) and other characters. In C, lexical units are delimited by whitespace and other characters, including arithmetic operators. As an example, consider parceling the characters from the line `int i = 20 ;` of C code into lexemes (Table 3.1). The lexemes are `int`, `i`, `=`, `20`, and `;`. The lines of code `int i=20;`, `int i = 20;`, and `int i = 20 ;` have this same set of lexemes.

Free-format languages are languages where formatting has no effect on program structure—of course, other than use of some delimiter to determine where lexical units begin and end. Most languages, including C, C++, and Java, are free-format languages. However, some languages impose restrictions on formatting. Languages where formatting has an effect on program structure, and where lexemes must occur in predetermined areas, are called *fixed-format languages*. Early versions of Fortran were fixed-format. Other languages, including Python, Haskell, Miranda, and occam, use *layout-based* syntactic grouping (i.e., indentation).

Once we have a list of lexical units, we must determine whether each is a lexeme (i.e., lexically valid). This can be done by checking them against the lexemes of the language (i.e., a *lexicon*), or by running each through a finite-state automaton that can recognize the lexemes of the language. Most programming languages have *reserved words* that cannot be used as an identifier (e.g., `int` in C). Reserved words are not the same as *keywords*, which are only special in certain contexts (e.g., `main` in C).

Lexeme	Token
<code>int</code>	reserved word
<code>i</code>	identifier
<code>=</code>	special symbol
<code>20</code>	constant
<code>;</code>	special symbol

Table 3.1 Parceling Lexemes into Tokens in the Sentence `int i = 20 ;`

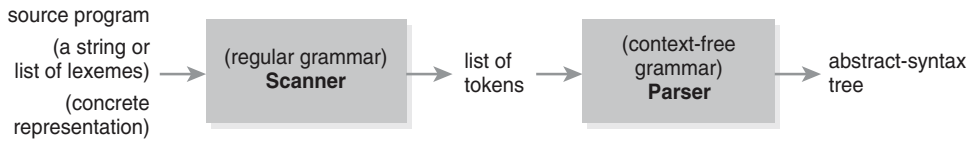


Figure 3.1 Simplified view of scanning and parsing: the front end.

As each lexical unit is determined to be valid, each is abstracted into a token because the individual lexemes are superfluous for the next phase—syntactic analysis. Lexemes are partitioned into *tokens*, which are categories of lexemes. Table 3.1 shows how the five lexemes in the string `int i = 20;` fall into four token categories. The next phase in verifying the validity of a program is determining whether the tokens are structured properly. The actual lexemes are not important in verifying the structure of a candidate sentence. The details of a lexeme (e.g., `i`) are abstracted in its token (e.g., `<identifier>`). For the program to be a sentence, the order of the tokens must conform to a context-free grammar. Here we are referring to the grammar of entire expressions rather than the (regular) grammar of individual lexemes. If a program is lexically valid, lexical analysis returns a list of tokens.

A *scanner*¹ (or *lexical analyzer*) is a software system that culls the lexical units from a program, validates them as lexemes, and returns a list of tokens. Parsing validates the order of the tokens in this list and, if it is valid, organizes this list of tokens into a parse tree. The system that validates a program string and, if valid, converts it into a parse tree is called a *front end* (and it constitutes both a scanner and parser; Figure 3.1). Notice how the two components of a front end in Figures 3.1 and 3.2 correspond to progressive types of sentence validity in Table 2.1.

The finite-state automaton (FSA), shown in Figure 3.3, recognizes both positive integers and legal identifiers in C. Table 3.2 illustrates how one might represent the transitions of that FSA as a two-dimensional array. The indices 1, 2, and 3 denote the current state of the machine, and the integer value in each cell denotes which state to transition to when a particular input character is encountered. For instance, if the machine is in state 1 and an integer in the range 1...9 is encountered, the machine transitions to state 3.

Because the theory behind scanners (i.e., finite-state automata, regular languages, and regular grammars) is well established, building a scanner is a mechanical process that can be automated by a computer program; thus, it is rarely done by hand. The scanner generator `lex` is a UNIX tool that accepts a set of regular expressions (in a `.l` file) as input and automatically generates a lexical analyzer in C that can recognize lexemes in the language denoted by those regular expressions; each call to the function `lex()` retrieves the next token. In other words, given a set of regular expressions, `lex` generates a scanner in C.

1. A *scanner* is also sometimes referred to as a *lexer*.

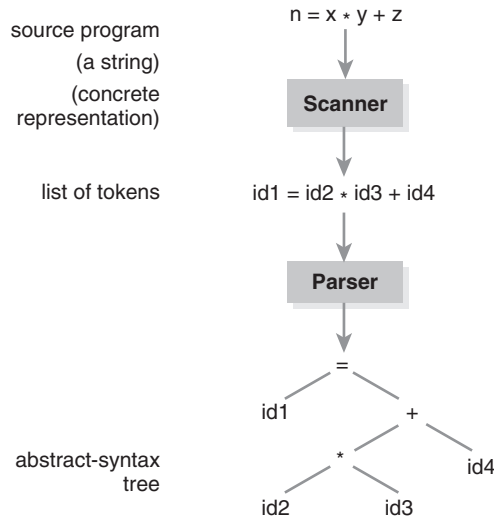


Figure 3.2 More detailed view of scanning and parsing.

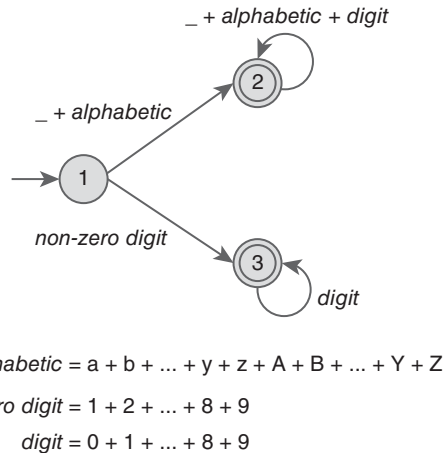


Figure 3.3 A finite-state automaton for a legal identifier and positive integer in C.

3.3 Parsing

Parsing (or *syntactic analysis*) is the process of determining whether a string is a sentence (in some language) and, if so, (typically) converting the *concrete representation* of it into an *abstract representation*, which generally facilitates the intended subsequent processing of it. A *concrete-syntax representation* of a program is typically a string (or a parse tree as shown in Chapter 2, where the terminals along the fringe of the tree from left-to-right constitute the input string). Since a program in concrete syntax is not readily processable, it must be parsed into an abstract representation, where the details of the concrete-syntax representation

input character	current state		
	1	2	3
—	2	2	ERROR
a + b + . . . + y + z	2	2	ERROR
A + B + . . . + Y + Z	2	2	ERROR
0	ERROR	2	3
1 + 2 + . . . + 8 + 9	3	2	3

Table 3.2 Two-Dimensional Array Modeling a Finite-State Automaton for a Legal Identifier and Positive Integer in C.

	<i>lexics</i>	<i>syntax</i>
<i>concrete</i>	lexeme $\in$	parse tree
$\downarrow$	<i>scanning</i> $\rightsquigarrow$	$\downarrow$
<i>abstract</i>	token $\in$	abstract-syntax tree
		$\rightsquigarrow$ <i>parsing</i>

Table 3.3 (Concrete) Lexemes and Parse Trees Vis-à-Vis (Abstract) Tokens and Abstract-Syntax Trees, Respectively

that are irrelevant to the subsequent processing are abstracted away. A *parse tree* and *abstract-syntax tree* are the syntactic analogs of a *lexeme* and *token* from *lexics*, respectively (Table 3.3). (See Section 9.5 for more details on abstract-syntax representations.) A *parser* (or *syntactic analyzer*) is the component of an interpreter or compiler that also typically converts the source program, once syntactically validated, into an abstract, or more easily manipulable, representation.

Often lexical and syntactic analysis are combined into a single phase (and referred to jointly as syntactic analysis) to obviate making multiple passes through the string representing the program. Furthermore, the syntactic validation of a program and the construction of an abstract-syntax tree for it can proceed in parallel. Note that parsing is independent of the subsequent processing planned on the tree: interpretation or compilation (i.e., translation) into another, typically, lower-level representation (e.g., x86 assembly code).

Parsers can be generally classified as one of two types: top-down or bottom-up. A top-down parser develops a parse tree starting at the root (or start symbol of the grammar), while a bottom-up parser starts from the leaves. (In Section 2.7, we implicitly conducted top-down parsing when we intuitively proved the validity of a string by building a parse tree for it beginning with the start symbol of the grammar.) There are two types of top-down parsers: table-driven and recursive descent. A table-driven, top-down parser uses a two-dimensional parsing table and a programmer-defined stack data structure to parse the input string. The parsing table is used to determine which move to apply given the non-terminal on the top of the stack and the next terminal in the input string. Thus, use of a table requires looking one token ahead in the input string without consuming it. The moves in the table are derived from production rules of the grammar. The

other type of top-down parsing, known as recursive-descent parsing, lends itself well to implementation.

3.4 Recursive-Descent Parsing

A seminal discovery in computer science was that the grammar used to *generate* sentences from the language can also be used to *recognize* (or *parse*) sentences from the language. This dual nature of a grammar is discernible in a recursive-descent parser, whose implementation follows directly from a grammar. The code for a recursive-descent parser mirrors the grammar for the language it parses. Thus, a grammar provides a design for the implementation of a recursive-descent parser.

Specifically, we construct a *recursive-descent parser* as a collection of functions, where each function corresponds to one non-terminal in the grammar and is responsible for recognizing the sub-language rooted at that non-terminal. The right-hand side of a production rule provides a design for the definition of the function corresponding to the non-terminal on the left-hand side of the rule. A non-terminal on the right-hand side translates into a call to the function corresponding to that non-terminal in the definition of the function corresponding to the non-terminal on the left-hand side. This type of parser is also called a *recursive-descent parser* because a non-terminal on the left side of a rule will often appear on the right side; thus, the parser recursively descends deeper into the grammar. A function for each non-terminal is implemented in this way until we arrive at functions for non-terminals with no non-terminals on the right-hand side of their production rules (i.e., the base case). Hence, a recursive-descent parser is a type of top-down parser. Sometimes a top-down parser is called a *predictive parser* because rather than starting with the string and working backward toward the start symbol, the parser predicts that the string conforms to the start symbol and, if proven incorrect, pursues alternative predictions.

Recursive-descent parsers are often written by hand. For instance, the popular gcc C compiler previously used an automatically yacc-generated, shift-reduce parser, but now uses a handwritten recursive-descent parser. Similarly, the clang C compiler² uses a handwritten, recursive-descent parser written in C++. The rationale behind the decision to use a recursive-descent parser is that it makes it easy for new developers to understand the code (i.e., simply mapping between the grammar and parser). Table 3.4 compares the table-drive and recursive-descent approaches to top-down parsing.

3.4.1 A Complete Recursive-Descent Parser

Consider the following grammar:

$$\begin{aligned} \langle \text{symbol-expr} \rangle &::= (\langle \text{s-list} \rangle) \mid x \mid y \mid z \\ \langle \text{s-list} \rangle &::= \langle \text{symbol-expr} \rangle [, \langle \text{s-list} \rangle] \end{aligned}$$

². clang is a unified front end for the C family of languages (i.e., C, Objective C, C++, and Objective C++).

Type of Top-down Parser	Parse Table Used	Parse Stack Used
Table-driven	explicit 2-D array data structure	explicit stack object in program
Recursive-descent	implicit/embedded in the code	implicit call stack of program

Type of Top-down Parser	Construction Complexity	Program Readability	Program Efficiency
Table-driven	complex; use generator	less readable	efficient
Recursive-descent	uncomplex; write by hand	more readable	efficient

Table 3.4 Implementation Differences in Top-down Parsers: Table-Driven Vis-à-Vis Recursive-Descent

where *<symbol-expr>* and *<s-list>* are non-terminals, *<symbol-expr>* is the start symbol, and *x*, *y*, *z*, *(*, *)*, and *,* are terminals. The following is code for a parser, with an embedded scanner, in Python for the language defined by this grammar:

```

1  import sys
2
3  # scanner
4  def validate_lexemes():
5      global sentence
6      for lexeme in sentence:
7          if (not valid_lexeme(lexeme)):
8              return False
9      return True
10
11 def valid_lexeme(lexeme):
12     return lexeme in ["(", ")", "x", "y", "z", ",", ""]
13
14 def getNextLexeme():
15     global lexeme
16     global lexeme_index
17     global sentence
18     global num_lexemes
19     global error
20
21     lexeme_index = lexeme_index + 1
22     if (lexeme_index < num_lexemes):
23         lexeme = sentence[lexeme_index]
24     else:
25         lexeme = " "
26
27 # parser
28
29 # <symbol-expr> ::= ( <s-list> ) | x | y | z
30 def symbol_expr():
31     global lexeme
32     global lexeme_index
33     global num_lexemes
34     global error
35     if (lexeme == "("):
36         getNextLexeme()

```

```

37         s_list()
38         if (lexeme != ")"):
39             error = True
40     elif lexeme not in ["x", "y", "z"]:
41         error = True
42     getNextLexeme()
43
44 # <s_list> ::= <symbol_expr> [ , <s_list> ]
45 def s_list():
46     global lexeme
47     symbol_expr()
48     # optional part
49     if lexeme == ',':
50         getNextLexeme()
51         s_list()
52
53 # main program
54 # read in the input sentences
55 for line in sys.stdin:
56     line = line[:-1] # remove trailing newline
57     sentence = line.split()
58
59     num_lexemes = len(sentence)
60
61     lexeme_index = -1
62     error = False
63
64     if (validate_lexemes()):
65         getNextLexeme()
66         symbol_expr()
67
68         # Either an error occurred or
69         # the input sentence is not entirely parsed.
70         if (error or lexeme_index < num_lexemes):
71             print("{}" is not a sentence.'.format(line))
72         else:
73             print("{}" is a sentence.'.format(line))
74     else:
75         print("{}" contains invalid lexemes and, thus, '
76             'is not a sentence.'.format(line))

```

Notice the one-to-one correspondence between non-terminals in the grammar and functions in the parser.

The parser accepts strings from standard input (one per line) until it reaches the end of the file (EOF) and determines whether each string is in the language defined by this grammar. Thus, it is helpful to think of this language using *<input>* as the start symbol and the following rule:

$$\langle \text{input} \rangle ::= \langle \text{input} \rangle \langle \text{symbol\_expr} \rangle \backslash n \mid \langle \text{symbol\_expr} \rangle \backslash n$$

where $\backslash n$ is a terminal.

Note that the program is factored into a scanner (lines 3–25) and recursive-descent parser (lines 27–51), as shown in Figure 3.1.

Input and Output

The lexical units in the input strings are whitespace delimited, and whitespace is ignored. Not all lexical units are assumed to be lexemes (i.e., valid). Notice

that this program recognizes two distinct error conditions. First, if a given string does not consist of lexemes, it responds with this message: "... " contains invalid lexemes and, thus, is not a sentence.. Second, if a given string consists of lexemes but is not a sentence according to the grammar, the parser responds with the message: "... " is not a sentence.. Note that the lexical error message takes priority over the parse error message. In other words, the parse error message is issued only if the input string consists entirely of lexemes. Only one line of output is written standard output per line of input.

Sample Session with the Parser

The following is a sample interactive session with the parser (> is simply the prompt for input and is the empty string in the parser):

```
> ( x )
"( x )" is a sentence.
> ( (
"( ( " is not a sentence.
> ( a )
"( a )" contains invalid lexemes and, thus, is not a sentence.
```

The scanner is invoked on line 64. The parser is invoked on line 66 by calling the function `sym_expr` corresponding to the start symbol `<symbol-expr>`. As functions are called while the string is being parsed, the run-time stack of activation records keeps track of the current state of the parse. If the stack is empty when the entire string is consumed, the string is a sentence; otherwise, it is not.

3.4.2 A Language Generator

The following Python program is a generator of sentences from the language defined by the grammar in this section:

```
1  import sys
2  import random;
3
4  # <symbol_expr> ::= ( <s_list> ) | x | y | z
5  def symbol_expr():
6      global num_tokens
7      global max_tokens
8
9      if (num_tokens < max_tokens):
10         if (random.randint (0, 1) == 0):
11             print(" ( ", end="")
12             num_tokens = num_tokens + 1
13             s_list()
14             print(" ) ", end="")
15             num_tokens = num_tokens + 1
16         else:
17             xyz = random.randint (0, 2)
18             if (xyz == 0):
19                 print("x ", end="")
20             elif (xyz == 1):
21                 print("y ", end="")
22             elif (xyz == 2):
```



```

23         print("z ", end="")
24         num_tokens = num_tokens + 1
25
26 # <s_list> ::= <symbol_expr> [ , <s_list> ]
27 def s_list():
28     global num_tokens
29     global max_tokens
30     symbol_expr()
31     # optional part
32     if (random.randint(0, 1) == 1 and num_tokens < max_tokens):
33         print(", ", end="")
34         s_list()
35
36 # main program
37 random.seed()
38
39 i = 0
40
41 num_sentences = int(sys.argv[1])
42 while (i < num_sentences):
43     max_tokens = random.randint(1, 100)
44     num_tokens = 0
45     symbol_expr()
46     print() # prints a newline
47     i = i + 1

```

The generator accepts a positive integer on the command line and writes that many sentences from the language to standard output, one per line. Notice that this generator, like the recursive-descent parser given in Section 3.4.1, has one procedure per non-terminal, where each such procedure is responsible for generating sentences from the sub-language rooted at that non-terminal.

Notice also that the generator produces sentences from the language in a random fashion. When several alternatives exist on the right-hand side of a production rule, the generator determines which non-terminal to follow randomly. The generator also generates sentences with a random number of lexemes. Each time it generates a sentence, it first generates a random number between the minimum number of lexemes necessary in a sentence and a maximum number that keeps the generated string within the character limit of the input strings to the parser (i.e., . . . characters). This random number serves as the maximum number of lexemes in the generated sentence. Every time the generator encounters an optional non-terminal (i.e., one enclosed in brackets), it flips a coin to determine whether it should pursue that path through the grammar. It pursues the path only if the flip indicates it should *and* if the number of lexemes generated so far is less than the random number of maximum lexemes generated.

3.5 Bottom-up, Shift-Reduce Parsing and Parser Generators

We engage in bottom-up parsing when we parse a string using the shift-reduce method (as we demonstrated in Section 2.7). The bottom-up nature refers to starting the parse with the terminals of the string and working backward (or

bottom-up) toward the start symbol of the grammar. In other words, a bottom-up parse of a string attempts to construct a rightmost derivation of the string in reverse (i.e., bottom-up). While parsing a string in this bottom-up fashion, we can also construct a parse tree for the sentence, if desired, by allocating nodes of the tree as we shift and setting pointers to pre-allocated nodes in the newly created internal nodes as we reduce. (We need not always build a parse tree; sometimes a traversal is enough, especially if semantic analysis or code generation phases will not follow the syntactic phase.)

Shift-reduce parsers, unlike recursive-descent parsers, are typically not written by hand. Like the construction of a scanner, the implementation of a shift-reduce parser is well grounded in theoretical formalisms and, therefore, can be automated. A *parser generator* is a program that accepts a syntactic specification of a language in the form of a grammar and automatically generates a parser from it. Parser generators are available for a wide variety of programming languages, including Python (PLY) and Scheme (SLLGEN). ANTLR (ANother Tool for Language Recognition) is a parser generator for a variety of target languages, including Java. Scanner and parser generators are typically used in concert with each other to automatically generate a front end for a language implementation (i.e., a scanner and parser).

The field of parser generation has its genesis in the classical UNIX tool `yacc` (yet another compiler compiler). The `yacc` parser generator accepts a context-free grammar in EBNF (in a `.y` file) as input and generates a shift-reduce parser in C for the language defined by the input grammar. At any point in a parse, the parsers generated by `yacc` always take the action (i.e., a shift or reduce) that leads to a successful parse, if one exists. To determine which action to take when more than one action will lead to a successful parse, `yacc` follows its default actions. (When `yacc` encounters a shift-reduce conflict, it shifts by default; when `yacc` encounters a reduce-reduce conflict, it reduces based on the first rule in lexicographic order in the `.y` grammar file.) The tools `lex` and `yacc` together constitute a scanner/parser generator system.³

The `yacc` language describes the rules of a context-free grammar and the actions to take when reducing based on those rules, rather than describing computation explicitly. Very high-level languages such as `yacc` are referred to as *fourth-generation languages* because three levels of language abstraction precede them: machine code, assembly language, and high-level language.

Recall (as we noted in Chapter 2) that while semantics can sometimes be reasonably modeled using a context-free grammar, which is a device for modeling the *syntactic structure* of language, a context-free grammar can always be used to model the *lexical structure* of language, since any regular language can be modeled by a context-free grammar. Thus, where scanning (i.e., lexical analysis) ends and parsing (i.e., syntactic analysis) begins is often blurred from both language design and implementation perspectives. Addressing semantics while parsing can

3. The GNU implementations of `lex` and `yacc`, which are commonly used in Linux, are named `flex` and `bison`, respectively.

obviate the need to make multiple passes through the input string. Likewise,⁴ addressing lexics while parsing can obviate the need to make multiple passes through the input string.

3.5.1 A Complete Example in `lex` and `yacc`

The following are `lex` and `yacc` specifications that generate a shift-reduce, bottom-up parser for the symbolic expression language presented previously in this chapter.

```

1  /* symexpr.l */
2  %{
3  #include <string.h>
4  extern char* temp;
5  extern int lexerror;
6  int yyerror(char* errmsg);
7  extern char* errmsg;
8  %}
9  %%
10 [xyz, ()] { strcat(temp,yytext); return *yytext; }
11 \n        { return *yytext; }
12 [ \t]      { strcat(temp,yytext); } /* ignore whitespace */
13 .          { strcat(temp,yytext);
14             sprintf(errmsg, "Invalid lexeme: '%c'.", *yytext);
15             yyerror(errmsg);
16             lexerror = 1; return *yytext; }
17 %%
18 int yywrap (void) {
19     return 1;
20 }
```

The pattern-action rules for the relevant lexemes are defined using UNIX-style regular expressions on lines 10–16. A pattern with outer square brackets matches exactly one of any of the characters within the brackets (lines 10 and 12) and `.` (line 13) matches any single character except a newline, which is matched on line 11.

```

1  /* symexpr.y */
2  %{
3  #include <stdio.h>
4  #include <string.h>
5  int yylineno;
6  int yydebug=0;
7  char* temp;
8  char* errmsg;
9  int lexerror = 0;
10 int yyerror(char* errmsg);
11 int yylex(void);
12 %}
13 %%
14 input: input sym_expr '\n' { printf ("\n%s\n" is an expression.\n", temp);
15                             *temp = '\0'; }
16     | sym_expr '\n'        { printf ("\n%s\n" is an expression.\n", temp);
17                             *temp = '\0'; }
18     | error '\n'           { if (lexerror) {
19                             printf ("\n%s\n" contains invalid ", temp);
20                             printf ("lexemes and, thus, ");
21                             printf ("is not a sentence.\n");

```

4. Though in the other direction along the expressivity scale.

```

22         lexerror = 0;
23     } else {
24         printf ("\\"%s\" is not an ", temp);
25         printf ("expression.\n");
26     }
27     *temp = '\\0';
28     yyclearin; /* discard lookahead */
29     yyerrok; }
30 ;
31 sym_expr: '(' s_list ')' { /* no action */ }
32         | 'x' { /* no action */ }
33         | 'y' { /* no action */ }
34         | 'z' { /* no action */ }
35 ;
36 s_list: sym_expr { /* no action */ }
37         | sym_expr ',' s_list { /* no action */ }
38 ;
39 %%
40 int yyerror(char *errmsg) {
41     fprintf(stderr, "%s\\n", errmsg);
42     return 0;
43 }
44 int main(void) {
45     temp = malloc (sizeof(*temp)*255);
46     errmsg = malloc (sizeof(*errmsg)*255);
47     yyparse();
48     free(temp);
49     return 0;
50 }

```

The shift-reduce pattern-action rules for the symbolic expression language are defined on lines 14–38. The patterns are the production rules of the grammar and are given to the left of the opening curly brace. Each action associated with a production rule is given between the opening and closing curly braces to the right of the rule and represented as C code. The action associated with a production rule takes place when the parser uses that rule to reduce the symbols on the top of the stack as demonstrated in Section 2.7.

Note that the actions in the second and third pattern-action rules (lines 31–38) are empty. In other words, there are no actions associated with the `sym_expr` and `s_list` production rules. (If we were building a parse or abstract-syntax tree, the C code to allocate the nodes of the tree would be included in the actions blocks of the second and third rules.) The first rule (lines 14–30) has associated actions and is used to accept one or more lines of input. If a line of input is a `sym_expr`, then the parser prints a message indicating that the string is a sentence. If the line of input does not parse as a `sym_expr`, it contains an `error` and the parser prints a message indicating that the string is not a sentence. The parser is invoked on line 47.

These scanner and parser specification files are compiled into an executable parser as follows:

```

$ ls
symexpr.l symexpr.y
$ flex symexpr.l # produces the scanner in lex.yy.c
$ ls
lex.yy.c symexpr.l symexpr.y
$ bison -t symexpr.y # produces the parser in symexpr.tab.c
$ ls

```

```

lex.yy.c symexpr.l symexpr.tab.c symexpr.y
$ gcc lex.yy.c symexpr.tab.c -o symexpr_parser
$ ls
lex.yy.c symexpr.l symexpr_parser symexpr.tab.c symexpr.y
$ ./symexpr_parser
( x )
"( x )" is a sentence.
( (
"( ( " is not a sentence.
( a )
Invalid lexeme: 'a'.
"( a )" contains invalid lexemes and, thus, is not a sentence.

```

Table 3.5 later in this chapter compares the top-down and bottom-up methods of parsing.

3.6 PLY: Python Lex-Yacc

PLY is a parser generator for Python akin to `lex` and `yacc` for C. In PLY, tokens are specified using regular expressions and a context-free grammar is specified using a variation of EBNF. The `yacc.yacc()` function is used to automatically generate the scanner/parser; it returns an object containing a parsing function. As with `yacc`, it is up to the programmer to specify the actions to be performed during parsing to build an *abstract-syntax representation* (Section 9.5).

3.6.1 A Complete Example in PLY

The following is the PLY analog of the `lex` and `yacc` specifications from Section 3.5 to generate a parser for the symbolic expression language:

```

1  from sys import stdin
2  import ply.lex as lex
3  import ply.yacc as yacc
4
5  # Grammar in EBNF:
6  # symexpr : ( slist ) | x | y | x
7  # slist   : symexpr [ , slist ]
8
9  # SCANNER
10
11 tokens = (
12     'X',
13     'Y',
14     'Z',
15     'LPAREN',
16     'RPAREN',
17     'COMMA'
18 )
19
20 t_X      = r'x'
21 t_Y      = r'y'
22 t_Z      = r'z'
23 t_LPAREN = r'\('
24 t_RPAREN = r'\)'
25 t_COMMA  = r','

```

```

26
27 t_ignore = ' \t'
28
29 def t_error(t):
30     raise ValueError("Invalid lexeme '{}'.format(t.value[0]))
31     t.scanner.skip(1)
32
33 # PARSER
34
35 def p_symexpr(p):
36     """symexpr : LPAREN slist RPAREN
37                | X
38                | Y
39                | Z """
40     p[0] = True
41
42 def p_slist(p):
43     """slist : symexpr
44              | symexpr COMMA slist"""
45     p[0] = True
46
47 def p_error(p):
48     raise SyntaxError("Parse error.")
49
50 # main program
51 scanner = lex.lex()
52 parser = yacc.yacc()
53
54 for line in stdin:
55     line = line[:-1] # remove trailing newline
56     try:
57         if parser.parse(line):
58             print("{} is a sentence.".format(line))
59         else:
60             print("{} is not a sentence.".format(line))
61     except ValueError as e:
62         print(e.args[0])
63         print("{} contains invalid lexemes and, thus, '
64               'is not a sentence.".format(line))
65     except SyntaxError:
66         print("{} is not a sentence.".format(line))

```

The tokens for the symbolic expression language are defined on lines 11–31 and the shift-reduce pattern-action rules are defined on lines 35–48. Notice that the syntax of the pattern-action rules in PLY differs from that in `yacc`. In PLY, the pattern-action rules are supplied in the form of a function definition. The docstring literal at the top of the function definition (i.e., the text between the two `"""`) specifies the production rule, and the part after the closing `"""` indicates the action to be taken. The scanner and parser are invoked on lines 51 and 52, respectively. Strings are read from standard input (line 54) with the newline character removed (line 55) and passed to the parser (line 57). The string is then tokenized and parsed. If the string is a sentence, the `parser.parse` function returns `True`; otherwise, it returns `False`. The parser is generated and run as follows:

```

$ ls
symexpr.py
$ python3.8 symexpr.py
Generating LALR tables

```

```

( x )
"( x )" is a sentence.
( (
"( ( " is not a sentence.
( a )
Invalid lexeme 'a'.
"( a )" contains invalid lexemes and, thus, is not a sentence.
$ ls
parser.out  parsetab.py  symexpr.py
$ python3.8 symexpr.py
( y )
"( y )" is a sentence.

```

3.6.2 Camille Scanner and Parser Generators in PLY

The following is a grammar in EBNF for the language Camille developed in Part III of this text:

```

<program> ::= <expression>
<expression> ::= <number>
<expression> ::= <identifier>
<expression> ::= <primitive> ({<expression>}+ (,))
<primitive> ::= + | - | * | incl | decl | zero? | eqv?
<expression> ::= if <expression> <expression> else <expression>
<expression> ::= let {<identifier> = <expression>}+ in <expression>
<expression> ::= let* {<identifier> = <expression>}+ in <expression>
<expression> ::= <function>
<function> ::= fun ({<identifier>}* (,)) <expression>
<expression> ::= ( <expression> {<expression>}* (,))
<expression> ::= letrec {<identifier> = <function>}+ in <expression>

```

The Camille language evolves throughout the course of Part III. This grammar is for a version of Camille used in Chapter 11. The following code is a PLY *scanner specification* for the tokens in the Camille language:

```

1  import re
2  import sys
3  import operator
4  import ply.lex as lex
5  import ply.yacc as yacc
6  from collections import defaultdict
7
8  # begin lexical specification #
9  tokens = ('NUMBER', 'PLUS', 'WORD', 'MINUS', 'MULT', 'DECL',
10           'INCL', 'ZERO', 'LPAREN', 'RPAREN', 'COMMA',
11           'IDENTIFIER', 'LET', 'EQ',
12           'IN', 'IF', 'ELSE', 'EQV', 'COMMENT')
13
14  keywords = ('if', 'else', 'incl', 'decl',
15             'in', 'let', 'zero?', 'eqv?')
16
17  keyword_lookup = {'if' : 'IF', 'else' : 'ELSE',
18                  'incl' : 'INCL', 'decl' : 'DECL', 'in' : 'IN',
19                  'let' : 'LET', 'zero?' : 'ZERO',

```

```

20             'eqv?' : 'EQV' }
21
22 t_PLUS      = r'\+'
23 t_MINUS     = r'\-'
24 t_MULT      = r'\*'
25 t_LPAREN    = r'\('
26 t_RPAREN    = r'\)'
27 t_COMMA     = r','
28 t_EQ        = r'='
29 t_ignore    = " \t"
30
31 def t_WORD(t):
32     r'[A-Za-z_][A-Za-z_0-9*?!]*'
33     pattern = re.compile("^[A-Za-z_][A-Za-z_0-9*?!]*$")
34
35     # if the identifier is a keyword, parse it as such
36     if t.value in keywords:
37         t.type = keyword_lookup[t.value]
38     # otherwise it might be a variable so check that
39     elif pattern.match(t.value):
40         t.type = 'IDENTIFIER'
41     # otherwise it is a syntax error
42     else:
43         print("Runtime error: Unknown word %s %d" %
44               (t.value[0], t.lexer.lineno))
45         sys.exit(-1)
46     return t
47
48 def t_NUMBER(t):
49     r'[-?]\d+'
50     # try to convert the string to an int, flag overflows
51     try:
52         t.value = int(t.value)
53     except ValueError:
54         print("Runtime error: number too large %s %d" %
55               (t.value[0], t.lexer.lineno))
56         sys.exit(-1)
57     return t
58
59 def t_COMMENT(t):
60     r'---.*'
61     pass
62
63 def t_newline(t):
64     r'\n'
65     # continue to next line
66     t.lexer.lineno = t.lexer.lineno + 1
67
68 def t_error(t):
69     print("Unrecognized token %s on line %d." % (t.value.rstrip(),
70                                                  t.lexer.lineno))
71
72 lexer = lex.lex()
73 # end lexical specification #

```

The following code is a PLY *parser specification* for the Camille language defined by this grammar:

```

73 class ParserException(Exception):
74     def __init__(self, message):
75         self.message = message

```



```

139 def p_arguments(t):
140     '''arguments : expression
141                 | expression COMMA arguments'''
142
143 def p_expressions(t):
144     '''expressions : expression
145                  | expression COMMA expressions'''
146
147 def p_let_statement(t):
148     '''let_statement : let_assignment
149                    | let_assignment let_statement'''
150
151 def p_letstar_statement(t):
152     '''letstar_statement : letstar_assignment
153                       | letstar_assignment letstar_statement'''
154
155 def p_letrec_statement(t):
156     '''letrec_statement : letrec_assignment
157                      | letrec_assignment letrec_statement'''
158
159 def p_let_assignment(t):
160     '''let_assignment : IDENTIFIER EQ expression'''
161
162 def p_letstar_assignment(t):
163     '''letstar_assignment : IDENTIFIER EQ expression'''
164
165 def p_letrec_assignment(t):
166     '''letrec_assignment : IDENTIFIER EQ rec_func_decl'''
167 # end syntactic specification #

```

Notice that the action part of each pattern-action rule is empty. Thus, this parser does not build an abstract-syntax tree. For a parser generator that builds an abstract-syntax tree (used later for interpretation in Chapters 10–11), see the listing at the beginning of Section 9.6.2.⁵ For the details of PLY, see <https://www.dabeaz.com/ply/>.

3.7 Top-down Vis-à-Vis Bottom-up Parsing

A hierarchy of parsers can be developed based on properties of grammars used in them (Table 3.5). Top-down and bottom-up parsers are classified as LL and LR parsers, respectively. The first L indicates that both read the input string from Left-to-right. The second character indicates the type of derivation the parser

Description of Parser	Parser Type	Reads Input	Derivation Constructed	Requisite Grammar	Recursion in Rules
Bottom-up	LR	Left-to-right	Rightmost	unambiguous	left-recursive [†]
Top-down	LL	Left-to-right	Leftmost	unambiguous	right-recursive [‡]

Table 3.5 Top-down Vis-à-Vis Bottom-up Parsers (Key: ‡ = requisite; † = preferred.)

⁵ These specifications have been tested and run in PLY 3.11. The scanner and parser generated by PLY from these specifications have been tested and run in Python 3.8.5.

Grammar Type	Grammar Ambiguity	Recursion in Rules	Grammar Construction	Grammar Readability
LR	unambiguous	left- or right-recursive	less restrictive	reasonable readable
LL	unambiguous	right-recursive only	restrictive	readable

Table 3.6 LL Vis-à-Vis LR Grammars (Note: $LL \subset LR$.)

constructs: Top-down parsers construct a Leftmost derivation, while bottom-up parsers construct a Rightmost derivation. Use of a parsing table in table-driven parsers, which can be top-down or bottom-up, often requires looking one token ahead in the input string without consuming it. These types of parsers are classified by prepending LA (for Look Ahead) before the first two characters and appending (n), where the integer n indicates the length of the look ahead required. For instance, the LR (or bottom-up) shift-reduce parsers generated by `yacc` are LALR(1) parsers (i.e., Look-Ahead, Left-to-right, Rightmost derivation parsers). These types of parsers also require the grammar used to be in a particular form. Both LL and LR parsers require an unambiguous grammar. Furthermore, an LL parser requires a right-recursive grammar earlier. Thus, there is a corresponding hierarchy of grammars (Table 3.6).

Conceptual Exercises for Chapter 3

Exercise 3.1 Explain why a *right-recursive* grammar is *required* for a *recursive-descent* parser.

Exercise 3.2 Why might it be *preferable* to use a *left-recursive* grammar with a *bottom-up* parser?

Programming Exercises for Chapter 3

Table 3.7 presents a mapping from the exercises here to some of the essential features of parsers discussed in this chapter.

Exercise 3.3 Implement a scanner, in any language, to print all lexemes in a C program.

Exercise 3.4 Consider the following grammar in EBNF:

$$\langle P \rangle ::= () \mid (\langle P \rangle) \mid () (\langle P \rangle) \mid (\langle P \rangle) \langle P \rangle$$

where $\langle P \rangle$ is a non-terminal and (and) are terminals.

Programming Exercise	Description of Language	Start from	Parser		Generator R-D (c)	Diagrammer		Parse Tree	
			R-D (a)	S-R (b)		R-D (d)	S-R (e)	R-D (f)	S-R (g)
Sections 3.4.1, 3.4.2, 3.5.1	<i>S-expressions</i>	N/A	✓	✓	✓	×	×	×	×
3.4	Dyck language	N/A	✓	✓	✓	×	×	×	×
3.5	Simple calculator	N/A	✓	✓	✓	✓	✓	✓	✓
3.6	Extended calculator	N/A	✓	✓	✓	×	×	×	×
3.7	Simple boolean expressions	N/A	✓	✓	✓	✓	✓	✓	✓
3.8	Extended boolean expressions	3.7	✓	✓	✓	×	×	×	×
3.9	English sentences	N/A	✓	✓	✓	✓	✓	✓	✓
3.10	Postfix expressions	N/A	×	×	×	×	×	×	×

Table 3.7 Parsing Programming Exercises in This Chapter, Including Their Essential Properties and Dependencies (Key: R-D = recursive-descent; S-R = shift-reduce.)

- (a) Implement a recursive-descent parser in any language that accepts strings from standard input (one per line) until EOF and determines whether each string is in the language defined by this grammar. Thus, it is helpful to think of this language using $\langle \text{input} \rangle$ as the start symbol and the rule:

$$\langle \text{input} \rangle ::= \langle \text{input} \rangle \langle P \rangle \backslash n \mid \langle P \rangle \backslash n$$

where $\backslash n$ is a terminal.

Factor your program into a scanner and recursive-descent parser, as shown in Figure 3.1.

You may not assume that each lexical unit will be valid and separated by exactly one space, or that each line will contain no leading or trailing whitespace. There are two distinct error conditions that your program must recognize. First, if a given string does not consist of lexemes, respond with this message: "... contains lexical units which are not lexemes and, thus, is not an expression., where ... is replaced with the input string, as shown in the interactive session following. Second, if a given string consists of lexemes but is not an expression according to the grammar, respond with this message: "... is not an expression., where ... is replaced with the input string, as shown in the interactive session following. Note that the "invalid lexemes" message takes priority over the "not an expression" message; that is, the "not an expression" message can be issued only if the input string consists entirely of valid lexemes.

You may assume that whitespace is ignored; that no line of input will exceed 4096 characters; that each line of input will end with a newline; and that no string will contain more than 200 lexical units.

Print only one line of output to standard output per line of input, and do not prompt for input. The following is a sample interactive session with the parser (> is simply the prompt for input and will be the empty string in your system):

```
> ()
"()" is a sentence.
> () ()
"() ()" is a sentence.
> (() )
"()" is a sentence.
> (() () ) ()
"(() () ) ()" is a sentence.
> (() () () ) ()
"(() () () ) ()" is a sentence.
> (a)
"(a)" contains lexical units which are not lexemes and, thus,
    is not a sentence.
> ) (
") (" is not a sentence.
> ) ()
"() ()" is not a sentence.
> ) () (
") () (" is not a sentence.
> (() ()
"() ()" is not a sentence.
> () () (
"() () ()" is not a sentence.
> (() () ()
"(() () ()" is not a sentence.
```

- (b) Automatically generate a shift-reduce, bottom-up parser by defining a specification of a parser for the language defined by this grammar in either `lex/yacc` or `PLY`.
- (c) Implement a generator of sentences from the language defined by the grammar in this exercise as an efficient approach to test-case generation. In other words, write a program to output sentences from this language. A simple way to build your generator is to follow the theme of recursive-descent parser construction. In other words, develop one procedure per non-terminal, where each such procedure is responsible for generating sentences from the sub-language rooted at that non-terminal. You can develop this generator from your recursive-descent parser by inverting each procedure to perform generation rather than recognition.

Your generator must produce sentences from the language in a random fashion. Therefore, when several alternatives exist on the right-hand side of a production rule, determine which non-terminal to follow randomly. Also, generate sentences with a random number of lexemes. To do so, each time you generate a sentence, generate a random number between the minimum number of lexemes necessary in a sentence and a maximum number that keeps the generated string within the character limit of the input strings to the parser from the problem. Use this random number to serve as the maximum number of

lexemes in the generated sentence. Every time you encounter an optional non-terminal (i.e., one enclosed in brackets), flip a coin to determine whether you should pursue that path through the grammar. Then pursue the path only if the flip indicates you should *and* if the number of lexemes generated so far is less than the random maximum number of lexemes you generated. Your generator must read a positive integer given at the command line and write that many sentences from the language to standard output, one per line.

Testing any program on various representative data sets is an important aspect of software development, and this exercise will help you test your parsers for this language.

Exercise 3.5 Consider the following grammar in EBNF:

$$\begin{aligned} \langle expr \rangle &::= \langle expr \rangle + \langle expr \rangle \\ \langle expr \rangle &::= \langle expr \rangle * \langle expr \rangle \\ \langle expr \rangle &::= - \langle expr \rangle \\ \langle expr \rangle &::= \langle integer \rangle \\ \langle integer \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid \dots \mid 2^{31}-1 \end{aligned}$$

where $\langle expr \rangle$ and $\langle integer \rangle$ are non-terminals and $+$, $*$, $-$, and $0, 1, 2, 3, \dots, 2^{31}-1$ are terminals.

Complete Programming Exercise 3.4 (parts a, b, and c) using this grammar, subject to all of the requirements given in that exercise.

The following is a sample interactive session with the parser:

```
> 2+3*4
"2+3*4" is an expression.
> 2+3*-4
"2+3*-4" is an expression.
> 2+3*a
"2+3*a" contains lexical units which are not lexemes and, thus,
    is not an expression.
> 2+*3*4
"2+*3*4" is not an expression.
```

- (d) At some point in your education, you may have encountered the concept of diagramming sentences. A diagram of a sentence (or expression) is a parse-tree-like drawing representing the grammatical (or syntactic) structure of the sentence, including parts of speech such as subject, verb, and object.

Complete Programming Exercise 3.4.a, but this time build a recursive-descent parser that writes a diagrammed version of the input string. Specifically, the output must be the input with parentheses around each non-terminal in the input string.

Do not build a parse tree to solve this problem. Instead, implement your recursive-descent parser to construct the diagrammed sentence as

demonstrated in the following Python and C procedures, respectively, that each parse and diagram a sub-sentence rooted at the non-terminal *<s-list>* from the grammar in Section 3.4.1:

```

1  # <s-list> ::= <symbol-expr> [ , <s-list> ]
2  def s_list():
3      global lexeme
4
5      print "("
6
7      symbol_expr()
8      # optional part
9      if lexeme == ',':
10         getNextLexeme()
11         s_list()
12
13     print ")"

1  /* <s-list> ::= <symbol-expr> [ , <s-list> ] */
2  bool s_list() {
3      bool valid;
4
5      printf("(");
6
7      valid = symbol_expr();
8      if (valid && nextLexeme != '\0')
9          /* optional part */
10         if (nextLexeme == ',') {
11             getNextLexeme();
12             valid = s_list();
13         }
14
15     printf(")");
16     return valid;
17 }

```

Print only one line of output to standard output per line of input as follows. Consider the following sample interactive session with the parser diagrammar (> is the prompt for input and is the empty string in your system):

```

> 2+3*4
"((2)+((3)*(4)))" is an expression.
> 2+3*-4
"((2)+((3)*(-(4))))" is an expression.
> 2+3*a
"2+3*a" contains lexical units which are not lexemes and, thus,
    is not an expression.
> 2+*3*4
"2+*3*4" is not an expression.

```

(e) Complete Programming Exercise 3.5.d using *lex/yacc* or *PLY*.

Hint: If using *lex/yacc*, use an array implementation of a stack that contains elements of type *char**. Also, use the *sprintf* function to convert an integer to a string. For example:

```

char* string_representation_of_an_integer =
    malloc (10*sizeof(*string_representation_of_an_integer));

```

```

/* prints the integer 789 to
   the string variable string_representation_of_an_integer */
sprintf (string_representation_of_an_integer, "%d", 789);
/* prints the string representation of the integer 789 to stdout */
printf ("%s", string_representation_of_an_integer);

```

- (f) Complete Programming Exercise 3.5.d, but this time build a parse tree in memory and traverse it to output the diagrammed sentence.
- (g) Complete Programming Exercise 3.5.f using `lex/yacc` or `PLY`.

Exercise 3.6 Consider the following grammar:

```

<expr> ::= <term> * <term>
<expr> ::= <term> - <term>
<expr> ::= <term>
<term> ::= <factor> / <factor>
<term> ::= <factor> + <factor>
<term> ::= <factor>
<factor> ::= <identifier> | <number> | (<expr>)
<identifier> ::= <alpha> <alphanumrest> | <alpha>
<alpha> ::= a | b | ... | y | z | A | B | ... | Y | Z | _
<alphanumrest> ::= <alphanum> <alphanumrest> | <alphanum>
<alphanum> ::= <alpha> | <digit>
<number> ::= <nonzerodigit> <rest> | <digit>
<rest> ::= <digit> <rest> | <digit>
<nonzerodigit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Complete Programming Exercise 3.4 (parts a, b, and c) using this grammar, subject to all of the requirements given in that exercise.

The following is a sample interactive session with the parser:

```

> ( 6 )
"( 6 )" is an expression.
> a
"a" is an expression.
> ( i )
"( i )" is not an expression.
> , a - 1
", a - 1" contains lexical units which are not lexemes and, thus,
is not an expression.
> ( ( a ) )
"(( a ))" is an expression.
> id * index - rate * 1001 - (r - 32) * key
"id * index - rate * 1001 - (r - 32) * key" is not an expression.
> ( ( ( a ) ) )
"((( a )))" is an expression.
> ; 10 - 10
"; 10 - 10" contains lexical units which are not lexemes and, thus,
is not an expression.

```



```

> 01 - 10
"01 - 10" is not an expression.
> a * b - c
"a * b - c" is not an expression.
> ( ( ( a a ) ) )
" ( ( ( a a ) ) )" is an expression.
> ( a ( a ) )
"( a ( a ) )" is not an expression.
> 2 * 3
"2 * 3" is an expression.
> ( )
" ( )" is not an expression.
> 2 * rate - ((3))
"2 * rate - ((3))" is not an expression.
> (
"(" is not an expression.
> ( f ( t ) ) )
"( f ( t ) ) )" is not an expression.
> f!a+u
"f!a+u" contains lexical units which are not lexemes and, thus,
  is not an expression.
> a*
"a*" is not an expression.
> _aaa+1
"_aaa+1" is an expression.
> ____aa+y
"____aa+y" is an expression.

```

Exercise 3.7 Consider the following grammar in BNF (not EBNF):

$$\begin{aligned}
 \langle \text{expr} \rangle &::= \langle \text{expr} \rangle \ \& \ \langle \text{expr} \rangle \\
 \langle \text{expr} \rangle &::= \langle \text{expr} \rangle \mid \langle \text{expr} \rangle \\
 \langle \text{expr} \rangle &::= \sim \langle \text{expr} \rangle \\
 \langle \text{expr} \rangle &::= \langle \text{literal} \rangle \\
 \langle \text{literal} \rangle &::= t \\
 \langle \text{literal} \rangle &::= f
 \end{aligned}$$

where t , f , $|$, $\&$, and $\sim$ are terminals.

Complete Programming Exercise 3.5 (parts a–g) using this grammar, subject to all of the requirements given in that exercise.

The following is a sample interactive session with the undiagramming parser:

```

> f | t & f | ~t
"f | t & f | ~t" is an expression.
> ~t | t | ~f & ~f & t & ~t | f
"~t | t | ~f & ~f & t & ~t | f" is an expression.
> f | t ; f | ~t
"f | t ; f | ~t" contains lexical units which are not lexemes and, thus,
  is not an expression.
> f | t & & f | ~t
"f | t & & f | ~t" is not an expression.

```

The following is a sample interactive session with the diagramming parser:

```
> f | t & f | ~t
"((f) | ((t) & (f))) | (~(t)))" is a diagrammed expression.
> ~t | t | ~f & ~f & t & ~t | f
"(((~(t)) | (t)) | (((~(f)) & (~(f))) & (t)) & (~(t)))) | (f))"
  is a diagrammed expression.
> f | t ; f
"f | t ; f" contains lexical units which are not lexemes and, thus,
  is not an expression.
> f | | t & ~t
"f | | t & ~t" is not an expression.
```

Exercise 3.8 Consider the following grammar in BNF (not EBNF):

```
<program> ::= (<declarations>, <expr>)
<declarations> ::= []
<declarations> ::= [ <varlist> ]
<varlist> ::= <var>
<varlist> ::= <var>, <varlist>
<expr> ::= <expr> & <expr>
<expr> ::= <expr> | <expr>
<expr> ::= ~ <expr>
<expr> ::= <literal>
<expr> ::= <var>
<literal> ::= t
<literal> ::= f
<var> ::= a...e
<var> ::= g...s
<var> ::= u...z
```

where $t, f, |, \&, [,], \sim$, and $a \dots e, g \dots s$, and $u \dots z$ are terminals.

Complete Programming Exercise 3.4 (parts a, b, and c) using this grammar, subject to all of the requirements given in that exercise.

The following is a sample interactive session with the undiagramming parser:

```
> ([, f | t & f | ~t)
"([, f | t & f | ~t)" is a program.
> ([, ~t | t | ~f & ~f & t & ~t | f)
"([, ~t | t | ~f & ~f & t & ~t | f)" is a program.
> ([p,q], ~t | p | ~e & ~f & t & ~q | r)
"([p,q], ~t | p | ~e & ~f & t & ~q | r)" is a program.
> ([, f | t ; f)
"([, f | t ; f)" contains lexical units which are not lexemes and, thus,
  is not a program.
> ([, f | | t & ~t)
"([, f | | t & ~t)" is not a program.
```

Exercise 3.9 Consider the following grammar in EBNF for some simple English sentences:

```

<sentence> ::= <subject><verb_phrase><object>
<subject>  ::= <noun_phrase>
<verb_phrase> ::= <verb> | <verb> <adv>
<object>   ::= <noun_phrase>
<verb>     ::= learn | lead | serve
<adv>      ::= yesterday | today | tomorrow
<noun_phrase> ::= [<adj_phrase>] <noun> [<prep_phrase>]
<noun>     ::= faith | hope | charity
<adj_phrase> ::= <adj> | <adj> <adj_phrase>
<adj>      ::= humble | patient | prudent
<prep_phrase> ::= <prep> <noun_phrase>
<prep>     ::= of | at | with

```

For simplicity, we ignore articles, punctuation, and capitalization, including the first word of the sentence, otherwise known as context.

Complete Programming Exercise 3.5 (parts a–g) using this grammar, subject to all of the requirements given in that exercise.

The following are a Java method and a Python function that each parse and diagram a sub-sentence rooted at the non-terminal `<adj>`:

```

static void adj() {
    if (lexeme.equals("humble") || lexeme.equals("patient") ||
        lexeme.equals("prudent")) {
        diagrammedSentence += "\"" + lexeme + "\"";
        getNextLexeme();
    } else {
        error = true;
    }
}

def adj():
    global diagrammedSentence
    global lexeme
    global error
    if lexeme in ["humble", "patient", "prudent"]:
        diagrammedSentence += "\"" + lexeme + "\""
        getNextLexeme()
    else:
        error = True

```

The following is a sample interactive session with the undiagramming parser:

```

> hope serve prudent humble charity
"hope serve prudent humble charity" is a sentence.
> prudent faith lead today humble hope with charity
"prudent faith lead today humble hope with charity" is a sentence.
> hope serve prudent hummble charity
"hope serve prudent hummble charity" contains lexical units which are
  not lexemes and, thus, is not a sentence.
> serve hope prudent humble charity
"serve hope prudent humble charity" is not a sentence.

```

The following is a sample interactive session with the diagramming parser:

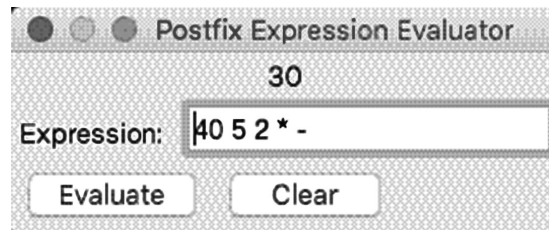
```
> hope serve prudent humble charity
(("hope") ("serve") (((("prudent" ("humble")) "charity"))))
> prudent faith lead today humble hope with charity
((((("prudent") "faith") ("lead" "today") (((("humble") "hope" ("with" ("charity"))))))))
> hope serve prudent hummble charity
"hope serve prudent hummble charity" contains lexical units which are not
lexemes and, thus, is not a sentence.
> serve hope prudent humble charity
"serve hope prudent humble charity" is not a sentence.
```

Exercise 3.10 Consider the following grammar for arithmetic expressions in postfix form:

```
<expr> ::= <expr> <expr> +
<expr> ::= <expr> <expr> -
<expr> ::= <expr> <expr> *
<expr> ::= <expr> <expr> /
<expr> ::= <number>
<number> ::= <nonzerodigit> <rest> | <digit>
<rest> ::= <digit> <rest> | <digit>
<nonzerodigit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Build a postfix expression evaluator using any programming language. Specifically, build a parser for the language defined by this grammar using a stack. When you encounter a number, push it on the stack. When you encounter an operator, pop the top two elements off the stack, compute the result of the operator applied to those two operands, and push the result on the stack. When the input string is exhausted, if there is only one number element on the stack, the string was a sentence in the language and the number on the stack is the result of evaluating the entire postfix expression.

Exercise 3.11 Build a graphical user interface, akin to that shown here, for the postfix expression evaluator developed in Programming Exercise 3.10.



Any programming language is permissible [e.g., HTML 5 and JavaScript to build a web interface to your evaluator; Java to build a stand-alone application; Qt (<https://doc.qt.io/qt-4.8/gettingstartedqt.html>, <https://zetcode.com/gui/qt5/>), Python (<https://wiki.python.org/moin/GuiProgramming>), Racket

Scheme (<https://docs.racket-lang.org/gui/>), or Squeak Smalltalk (<https://squeak.org/>)]. You could even build an Android or iOS app. All of these languages have a built-in or library stack data structure that you may use.

Exercise 3.12 Augment the PLY parser specification for Camille given in Section 3.6.2 with a read-eval-print loop (REPL) that accepts strings until EOF and indicates whether the string is a Camille sentence. Do not modify the code presented in lines 78–166 in the parser specification. Only add a function or functions at the end of the specification to implement the REPL.

Examples:

```
$ python3.8 camilleparse.py
Camille> +(-(35,33), incl(8))
"+(-(35,33), incl(8))" is a Camille sentence.
Camille> +(-(35,33), inc(8))
"+(-(35,33), inc(8))" is not a Camille sentence.
Camille> let a = 9 in a
"let a = 9 in a" is a Camille sentence.
Camille> let a = 9 in
"let a = 9 in" is not a Camille sentence.
```

3.8 Thematic Takeaways

- A seminal contribution to computer science is the discovery that grammars can be used as both *language-generation devices* and *language-recognition devices*.
- The structure of a recursive-descent parser follows naturally from the structure of a grammar, but the grammar must be in the proper form.

3.9 Chapter Summary

The source code of a program is simply a string of characters. After comments are purged from the string, scanning (or lexical analysis) partitions the string into the most atomic lexical units based on some delimiter (usually whitespace) and produces a list of these lexical units. The scanner, which models the regular grammar that defines the tokens of the programming language, then determines the validity of these lexical units. If all of the lexical units are lexemes (i.e., valid), the scanner returns a list of tokens—which is input to a parser. The parser, which models the context-free grammar that defines the structure or syntax of the language, determines whether the program is syntactically valid. Parsing (or syntactic analysis) determines whether a list of tokens is in the correct order and, if so, often structures this list into a parse tree. If the parser can construct a parse tree from the list of tokens, the program is syntactically valid; otherwise, it is not. If the program is valid, the result of parsing is typically a parse (or abstract-syntax) tree.

A variety of approaches may be used to build a parser. Each approach has requirements for the form of the grammar used and often offers complementary

advantages and disadvantages. Parsers can be generally classified as one of two types: top-down or bottom-up. A top-down parser builds a parse tree starting at the root (or start symbol of the grammar), while a bottom-up parser starts from the leaves. There are two types of top-down parsers: table-driven and recursive descent. A recursive-descent parser is a type of top-down parser that uses functions—one per non-terminal—and the internal run-time stack of activation records for function calls to determine the validity of input strings. The beauty of a recursive-descent parser is that the source code mirrors the grammar. Moreover, the parse table is implicit/embedded in the function definitions constituting the parser code. Thus, a recursive-descent parser is both readable and modifiable. Bottom-up parsing involves use of a shift-reduce method, whereby a rightmost derivation of the input string is constructed in reverse (i.e., the bottom-up nature refers to starting with the terminals of the string and working backward toward the start symbol of the grammar). There are also generators for bottom-up, shift-reduce parsers. The `lex` tool is a scanner generator for C; the `yacc` tool is a parser generator for C. In addition, scanner/parser generators are available for a variety of programming languages, including Python (PLY) and Java (e.g., ANTLR). A scanner and a parser constitute the syntactic component (sometimes called the *front end*) of a programming language implementation (e.g., interpreter or compiler), which we discuss in Chapter 4.

3.10 Notes and Further Reading

Layout-based syntactic grouping (i.e., indentation) originated in the experimental, and highly influential, family of languages ISWIM, described in Landin (1966).

We refer readers to Kernighan and Pike (1984, Chapter 8) and Niemann (n.d.) for discussion of automatically generating scanners and (shift-reduce, bottom-up parsers) parsers using `lex` and `yacc`, respectively, by defining specifications of the tokens and the grammar that defines the language of which parsed sentences are members. The classic text on *Lex and Yacc* by Levine, Mason, and Brown (1995) has been updated and titled *Flex and Bison* (Levine 2009). For an introduction to ANTLR, we refer readers to Parr (2012).

Chapter 4

Programming Language Implementation

So you are interpreters of interpreters?

— Socrates, *Io*

THE *front end* of a programming language implementation consists of a scanner and a parser. The output of the front end is typically an abstract-syntax tree. The actions performed on that abstract-syntax tree determine whether the *language implementation* is an *interpreter* or a *compiler*, or a combination of both—the topic of this chapter.

4.1 Chapter Objectives

- Describe the differences between a *compiler* and an *interpreter*.
- Explore a variety of *implementations* for programming languages.

4.2 Interpretation Vis-à-Vis Compilation

An *interpreter*, given the program input, traverses the abstract-syntax tree to *evaluate* and *directly execute* the program (see the right side of Figure 4.1 labeled “Interpreter”). There is no translation to object/bytecode involved in interpretation. “The interpreter for a computer language is just another program” (Friedman, Wand, and Haynes 2001, p. xi, Foreword, Hal Abelson). This observation is described as the most fundamental idea in computer programming (Friedman, Wand, and Haynes 2001). The input to an interpreter is (1) the source program to be executed and (2) the input of that source program. We say the input of the interpreter is the source program because to the programmer of the source program, the entire language implementation (i.e., Figure 4.1) is the interpreter rather than just the last component of it which accepts an

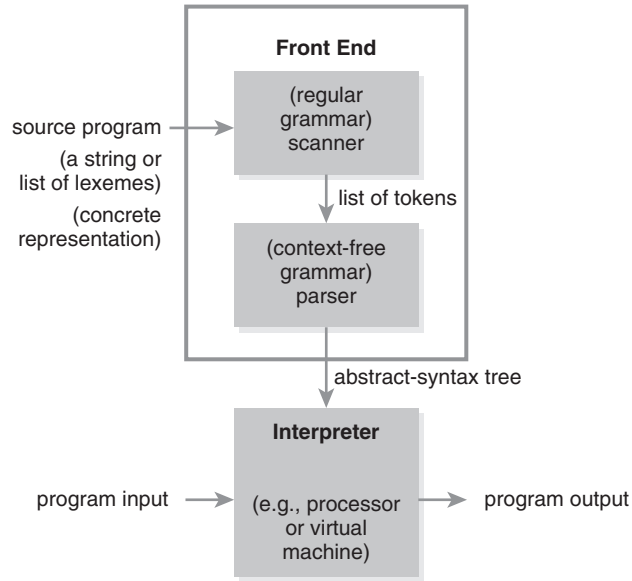


Figure 4.1 Execution by interpretation.

Data from Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 2001. *Essentials of Programming Languages*. 2nd ed. Cambridge, MA: MIT Press.

abstract-syntax tree as input labeled “Interpreter” (see the bottom component in Figure 4.1). The output of an interpreter is the output of the source program.

In contrast, a *compiler translates* the abstract-syntax tree (which is already an intermediate representation of the original source program) into another intermediate representation of the program (often assembly code), which is typically closer in similarity to the instruction set architecture (ISA) of the target processor intended to execute the program¹ (see the center of Figure 4.2 labeled “Compiler”). A compiler typically involves two subcomponents: the semantic analyzer and the code generator (neither of which is discussed here). Notice how the first three components used in the process of compilation (i.e., scanner, parser, semantic analyzer) in Figure 4.2 correspond to the three progressive types of sentence validity in Table 2.1.

Abstraction is the general concept referring to the idea that primitive details of an entity can be hidden (i.e., abstracted away) by adding a layer to that entity; this layer provides higher-level interfaces to those details such that the entity can be accessed and used without knowledge of its primitive details. Abstraction is a fundamental concept in computer science and recurs in many different contexts in the study of computer science. Progressively abstracting away from the details of the instruction set understood by the target processor has resulted in a series of programming languages, each at a higher level of abstraction than the prior:

1. This is not always true. For instance, the Java compiler `javac` outputs Java bytecode.

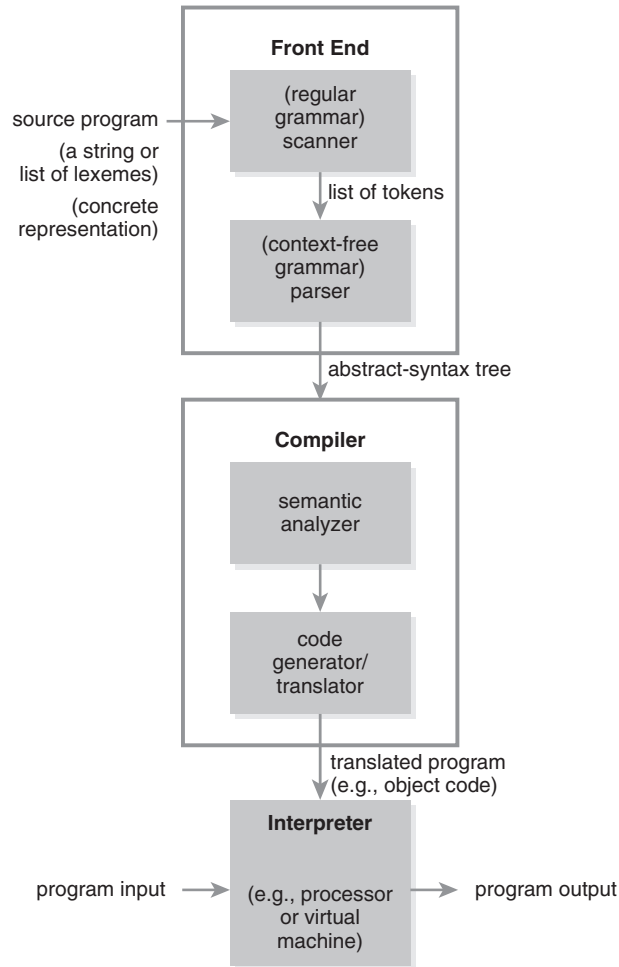


Figure 4.2 Execution by compilation.

Data from Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 2001. *Essentials of Programming Languages*. 2nd ed. Cambridge, MA: MIT Press.

4.	fourth-generation	language	(e.g., <code>lex</code> and <code>yacc</code>)
	↓		
3.	high-level	language	(e.g., Python, Java, and Scheme)
	↓		
2.	assembly	language	(e.g., MIPS)
	↓		
1.	machine	language	(e.g., x86)

Assembly languages (e.g., MIPS) replaced the binary digits of machine language with *mnemonics*—short English-like words that represent commands or data. High-level languages (e.g., Python) extend this abstraction with respect to control, procedure, and data. C is sometimes referred to as the lowest high-level language

because it provides facilities for manipulating machine addresses and memory, and inlining assembly language into C sources. Fourth-generation languages are referred to as such because they follow three prior levels. Note that machine language is not the end of abstraction. The 0s and 1s in object code are simply abstractions for electrical signals, and so on.

Compilation is typically structured as a series of transformations from the source program to an intermediate representation to another intermediate representation and so on, morphing the original source program so that it becomes closer, at each step, to the instruction set understood by the target processor, often until an assembly language program is produced. An *assembler*—not shown as a component in Figure 4.2—translates an assembly language program into machine code (i.e., object code). A compiler is simply a translator; it does not execute the source program—or the final translated representation of the program it produces—at all. Furthermore, its translation need not bring the source program any closer to the instruction set of the targeted platform. For instance, a system that translates a C program to a Java program is no less a compiler than a system that translates C code into assembly code. Another example is a $\text{\LaTeX}$ compiler from $\text{\LaTeX}$ source code—a high-level language for describing and typesetting documents—to PostScript—a language interpreted by printers. A PostScript document generated by a compiler can be printed by a printer, which is a hardware interpreter for PostScript (see approach ① in Figure 4.6 later in this chapter), or rendered on a display using a software interpreter for PostScript such as Ghostscript (see approach ② in Figure 4.6).

Web browsers are software interpreters (compiled into object code) that directly interpret HTML—a markup language describing the presentation of a webpage—as well as JavaScript and a variety of other high-level programming languages such as Dart.² (One can think of the front end of a language implementation as a compiler as well. The front end translates a source program—a string of characters—into an abstract-syntax tree—an intermediate representation.) Therefore, a more appropriate term for a compiler is *translator*. The term *compiler* derives from the use of the word to describe a program that *compiled* subroutines, which is now called a *linker*. Later in the 1950s the term *compiler*, shortened from “algebraic compiler,” was used—or misused—to describe a source-to-source translator conveying its present-day meaning (Bauer and Eickel 1975).

Sometimes students, coming from the perspective of an introductory course in computer programming in which they may have exclusively programmed using a compiled language, find it challenging to understand how the individual instructions in an interpreted program execute without being translated into object code. Perhaps this is because they know that in a computer system everything must be reduced to zeros and ones (i.e., object code) to execute. The following example demonstrates that an interpreter does not translate its source program into object code. Consider an interpreter that evaluates and runs a program written in the language *simple* with the following grammar:

2. <https://dart.dev>

$$\begin{aligned} \langle \text{simple} \rangle &::= \langle \text{digit} \rangle + \langle \text{digit} \rangle \\ \langle \text{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

The following is an interpreter, written in C, for the language *simple*:

```
#include<limits.h>
#include<stdio.h>
#include<ctype.h>

int main() {
    char string[LINE_MAX];
    /* sentences have exactly three non-whitespace characters */
    char program[4];

    int num1, num2, sum;
    int i = 0;
    int j = 0;

    /* fgets saves space for '\0' which is the null character */
    fgets (string, LINE_MAX, stdin);

    /* purge whitespace */
    while (string[i] != '\0') {
        if (!isspace(string[i])) {
            program[j] = string[i];
            j++;
            /* syntactic analysis */
            if (j == 4) {
                fprintf (stderr, "Program is invalid.\n");
                return -1;
            }
        }
        i++;
    }
    program[3] = '\0';

    /* lexical and syntactic analysis,
       note lack of semantic analysis */
    if (isdigit(program[0]) &&
        program[1] == '+' &&
        isdigit(program[2])) {

        /* subtracting the integer value of the
           ASCII character 0 from any ASCII digit
           returns the integer value of the digit
           (e.g., '2' - '0' = 2) */
        num1 = program[0] - '0';
        num2 = program[2] - '0';

        sum = num1 + num2;

        printf ("%d\n", sum);

        return 0;
    } else { /* invalid lexeme */
        fprintf (stderr, "Program is invalid.\n");
        return -2;
    }
}
```

A session with the *simple* interpreter follows:

```

1  $ gcc -o simple simple.c
2  $
3  $ file simple.c
4  simple.c: c program text, ASCII text
5  $
6  $ file simple
7  simple: Mach-O 64-bit executable x86_64
8  $
9  $ ./simple
10 2 + 3
11 5
12 $ ./simple
13 5+9
14 14
15 $ ./simple
16 3+ 8
17 11
18 $ ./simple
19 6 +0
20 6
21 $ ./simple
22 9   +       3
23 12
24 $ ./simple
25 123
26 Program is invalid.
27 $ ./simple
28 23 + 1
29 Program is invalid.
30 $ ./simple
31 2 + 3 + 4
32 Program is invalid.

```

The *simple* program `2 + 3` is never translated prior to execution. Instead, that program is read as input to the interpreter, which has been compiled into object code (i.e., the executable *simple*). It is currently executing on the processor and, therefore, has become part and parcel of the image of the *simple* interpreter process in memory (see Figure 4.3 and line 7 in the example session). In that sense, the *simple* program `2 + 3` has become part of the interpreter. An interpreter typically does not translate its source program into any representation other

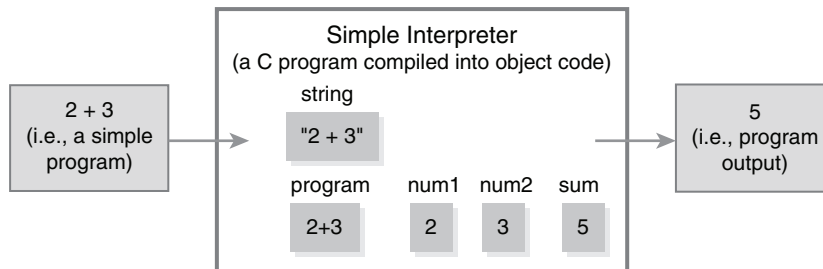


Figure 4.3 Interpreter for the language *simple*, illustrating that the *simple* program becomes part of the running interpreter process.

than an abstract-syntax tree or a similar data structure to facilitate subsequent evaluation.

In summary, an interpreter and a compiler each involve two major components. The first of these—the front end—is the same (see the top of Figures 4.1 and 4.2). The differences in the various approaches to implementation lie beyond the front end.

4.3 Run-Time Systems: Methods of Executions

Ultimately, the series of translations must end and a representation of the original source program must be interpreted [see the bottom of Figure 4.2 labeled “Interpreter” (e.g., processor).] Therefore, interpretation must, at some point, follow compilation. Interpretation can be performed by the most primitive of interpreters—a hardware interpreter called a processor—or by a software interpreter—which itself is just another computer program being interpreted.

Interpretation by the processor is the more common and traditional approach to execution after compilation (for purposes of speed of execution; see approach ① in Figure 4.6). It involves translating the source program all the way down, through the use of an assembler, to object code (e.g., x86). This more traditional style is depicted in the language-neutral diagram in Figure 4.4. For instance, `gcc` (i.e., the GNU C compiler) translates a C program into object code (e.g., `program.o`). For purposes of brevity, we omit the optional, but common, code optimization step and the necessary linking step from Figures 4.2 and 4.4. Often the code optimization phase of compilation is part of the *back end* of a language implementation.

An example of the final representation being evaluated by a software interpreter is a compiler from Java source code to Java bytecode, where the resulting bytecode is executed by the Java Virtual Machine—a software interpreter. These systems are sometimes referred to as *hybrid language implementations* (see approach ② in Figure 4.6). They are a hybrid of compilation and interpretation.³ Using a hybrid approach, high-level language is decoded only once and compiled into an architecturally neutral, intermediate form (e.g., bytecode) that is portable; in other words, it can be run on any system equipped with an interpreter for it. While the intermediate code cannot be interpreted as fast as object code, it is interpreted faster than the original high-level source program.

While we do not have a hardware interpreter (i.e., processor or machine) that natively executes programs written in high-level languages,⁴ an interpreter or compiler creates a *virtual machine* for the language of the source program (i.e., a computer that *virtually* understands that language). Therefore, an interpreter $\mathcal{I}_L$ for language L is a virtual machine for executing programs written in language L . For example, a Scheme interpreter creates a virtual Scheme computer. Similarly,

3. Any language implementation involving compilation must eventually involve interpretation; therefore, all language implementations involving compilation can be said to be hybrid systems. Here, we refer to hybrid systems as only those that compile to a representation interpreted by a compiled software interpreter (see approach ② in Figure 4.6).

4. A Lisp chip has been built as well as a Prolog computer called the Warren Abstract Machine.

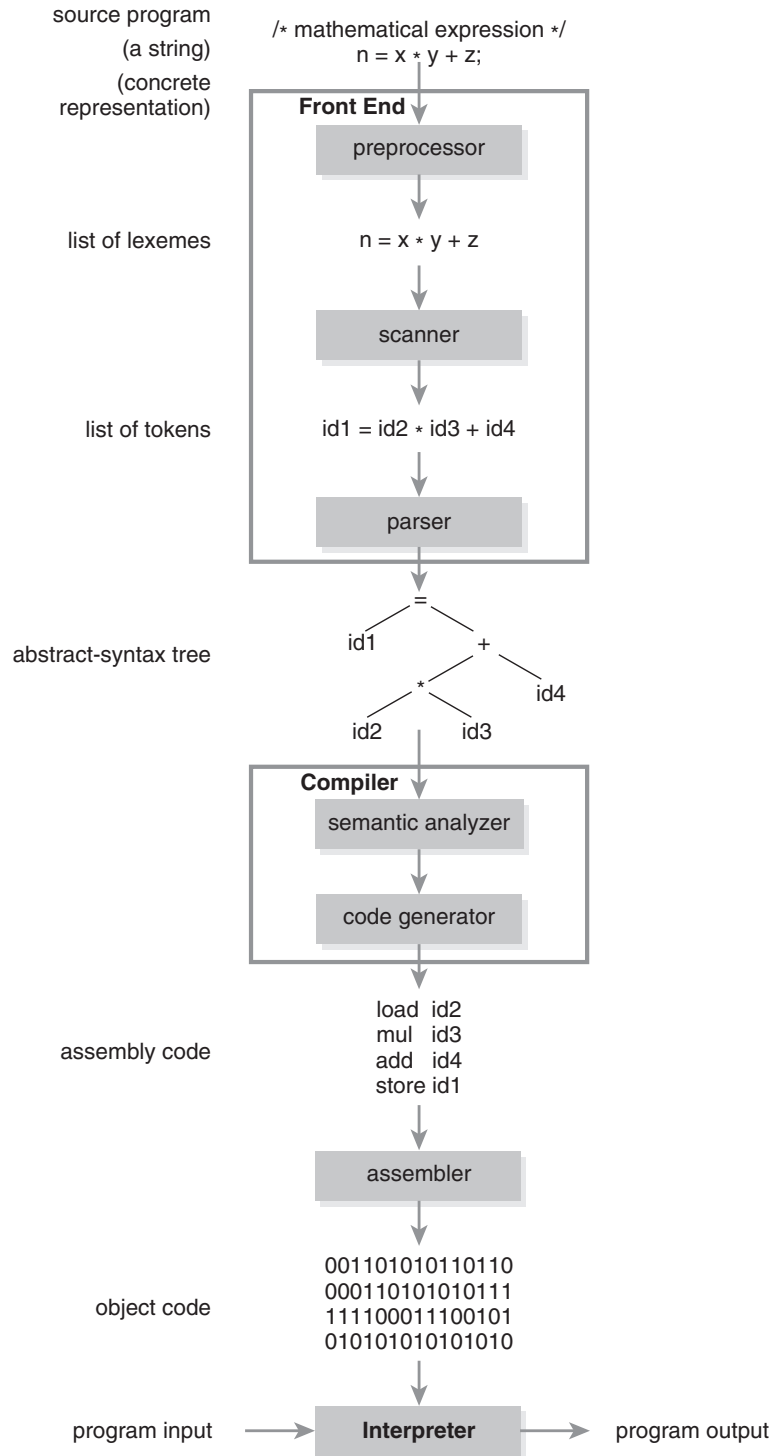


Figure 4.4 Low-level view of execution by compilation.

a compiler $C_{L \rightarrow L'}$ from a language L to L' can translate a program in language L either to a language (i.e., L') for which an interpreter executable for the target processor exists (i.e., $\mathcal{I}_{L'}$); alternatively, it can translate the program directly to code understood by the target processor. Thus, the $(C_{L \rightarrow L'}, \mathcal{I}_{L'})$ pair also serves as a virtual machine for language L . For instance, a compiler from Java source code to Java bytecode and a Java bytecode interpreter—the (javac, java) pair—provide a virtual Java computer.⁵ Programs written in the C# programming language within .NET run-time environment are compiled, interpreted, and executed in a similar fashion.

Some language implementations delay/perform the translation of (parts of) the final intermediate representation produced into object code until run-time. These systems are called *Just-in-Time (JIT) implementations* and use *just-in-time compilation*.

Ultimately, program execution relies on a hardware or software interpreter. (We build a series of progressive language interpreters in Chapters 10–12, where program execution by interpretation is the focus.)

This view of an interpreter as a virtual machine is assumed in Figure 4.1 where at the bottom of that figure the interpreter is given the abstract-syntax tree and the program input as input and executes the program directly to produce program output. Unless that interpreter (at the bottom) is a hardware processor and its input is object code, that figure is an abstraction of another process because the interpreter—a program like any other—needs to be executed (i.e., interpreted or compiled itself). Therefore, a lower-level presentation of interpretation is given in Figure 4.5.

Specifically, an interpreter compiled into object code is interpreted by a processor (see approach ③ of Figure 4.6). In addition to accepting the interpreter as input, the processor accepts the source program, or its abstract-syntax tree and the input of the source program. However, an interpreter for a computer language need not be compiled directly into object code. A software interpreter also can be interpreted by another (possibly the same) software interpreter, and so on—see approach ④ of Figure 4.6—creating a stack of interpreted software interpreters. At some point, however, the final software interpreter must be executed on the target processor. Therefore, program execution through a software interpreter ultimately depends on a compiler because the interpreter itself or the final descendant in the stack of software interpreters must be compiled into object code to run—unless the software interpreter is originally written in object code. For instance, the *simple* interpreter given previously is written in C and compiled into object code using gcc (see approach ③ of Figure 4.6). The execution of a compiled program depends on either a hardware or software interpreter. Thus, compilation and interpretation are mutually dependent upon each other in regard to program execution (Figure 4.7).

5. The Java bytecode interpreter (i.e., java) is typically referred to as the Java Virtual Machine or JVM by itself. However, it really is a virtual machine for Java bytecode rather than Java. Therefore, it is more accurate to say that the Java compiler and Java bytecode interpreter (traditionally, though somewhat inaccurately, called a JVM) together provide a virtual machine for Java.

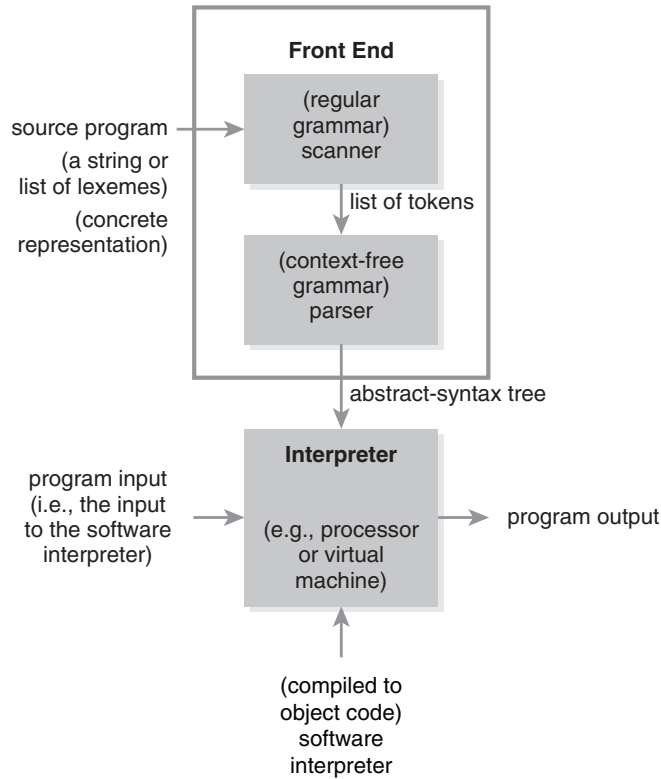


Figure 4.5 Alternative view of execution by interpretation.

Figure 4.6 summarizes the four different approaches to programming language implementation described here. Each approach is in a box that is labeled with a circled number and presented here in order from fastest to slowest execution:

- ① **Traditional compilation** directly to object code (e.g., Fortran, C)
- ② **Hybrid systems:** interpretation of a compiled, final representation through a compiled interpreter (e.g., Java)
- ③ **Pure interpretation** of a source program through a compiled interpreter (e.g., Scheme, ML)
- ④ Interpretation of either a source program or a compiled final representation through a **stack of interpreted software interpreters**

The study of language implementation and methods of execution, as depicted in Figure 4.6 through progressive levels of compilation and/or interpretation, again brings us face-to-face with the concept of abstraction.

Note that the figures in this section are conceptual: They identify the major components and steps in the interpretation and compilation process independent of any particular machine or computer architecture and are not intended to

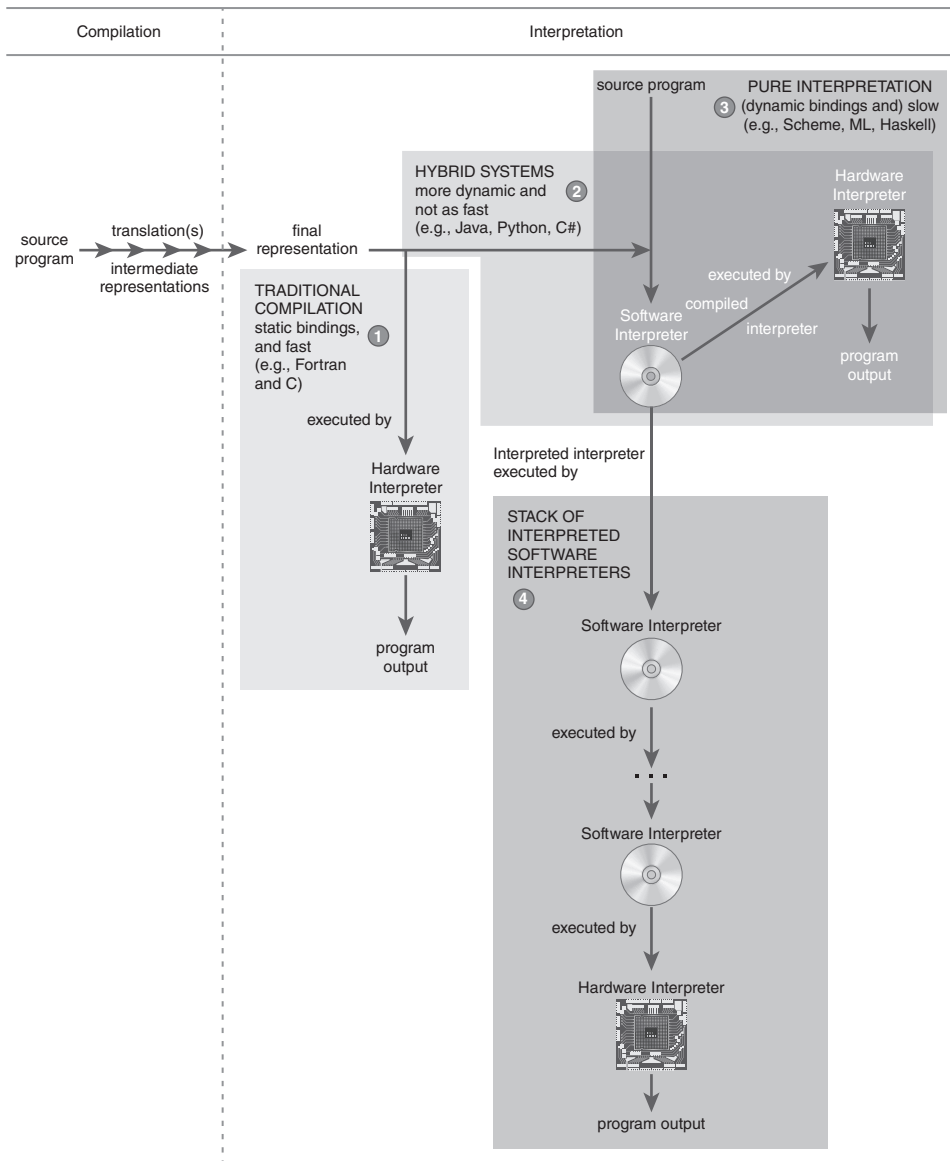


Figure 4.6 Four different approaches to language implementation.

model any particular interpreter or compiler. Some of these steps can be combined to obviate multiple passes through the input source program or representations thereof. For instance, we discuss in Section 3.3 that lexical analysis can be performed during syntactic analysis. We also mention in Section 2.8 that mathematical expressions can be evaluated while being syntactically validated. We revisit Figures 4.1 and 4.5 in Part III, where we implement fundamental

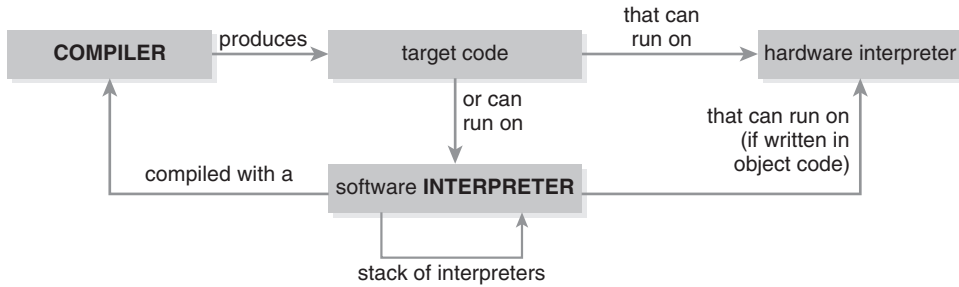


Figure 4.7 Mutually dependent relationship between compilers and interpreters in regard to program execution.

concepts of programming languages through the implementation of interpreters operationalizing those concepts.

4.4 Comparison of Interpreters and Compilers

Table 4.1 summarizes the advantages and disadvantages of compilation and pure interpretation. The primary difference between the two approaches is speed of execution. Interpreting high-level language is slower than interpreting object code primarily because *decoding* high-level statements and expressions is slower than decoding machine instructions. Moreover, a statement must be decoded as many times as it is executed in a program, even though it may appear in the program only once and the result of that decoding is the same each time. For instance, consider the following loop in a C fragment, which computes $2^{1,000,000}$ iteratively:⁶

```

int i=0;
int result=2;
for (i=1; i < 1000000; i++)
    result *= 2;

```

If this program was purely interpreted, the statement `result *= 2` would be decoded once less than 1 million times! Thus, not only does a software interpreter decode a high-level statement such as `result *= 2` more slowly than the processor decodes the analogous machine instruction, but that performance degradation is compounded by repeatedly decoding the same statement every time it is executed. An interpreter also typically requires more run-time space because the run-time environment—a data structure that provides the bindings of variables—is required during interpretation (Chapter 6). Moreover, often the source program is represented internally with a data structure designed for convenient access, interpretation, and modification rather than one with minimal

6. This code will not actually compute $2^{1,000,000}$ because attempting to do so will overflow the integer variable. This code is purely for purposes of discussion.

Implementation	Advantages	Disadvantages
Traditional Compiler	fast execution; compile once, run repeatedly	inconvenient program development; no REPL; less source-level debugging; less run-time flexibility
Pure Interpreter	convenient program development; REPL; direct source-level debugging; run-time flexibility	slow execution (decoding); often requires more run-time space

Table 4.1 Advantages and Disadvantages of Compilers and Interpreters

space requirements (Chapter 9). Often the internal representation of the source program accessed and manipulated by an interpreter is an abstract-syntax tree. An *abstract-syntax tree*, like a parse tree, depicts the structure of a program. However, unlike a parse tree, it does not contain non-terminals. It also structures the program in a way that facilitates interpretation (Chapters 10–12).

The advantages of a pure interpreter and the disadvantages of a traditional compiler are complements of each other. At a core level, program development using a compiled language is inconvenient because every time the program is modified, it must be recompiled to be tested and often the programmer cycles through a program-compile-debug-recompile loop ad nauseam. Program development with an interpreter, by comparison, involves one less step. Moreover, if provided with an interpreter, a read-eval-print loop (REPL) facilitates testing and debugging program units (e.g., functions) in isolation of the rest of the program, where possible.

Since an interpreter does not translate a program into another representation (other than an abstract-syntax representation), it does not obfuscate the original source program. Therefore, an interpreter can more accurately identify source-level (i.e., syntactic) origins (e.g., the name of an array whose index is out-of-bounds) of run-time errors and refer directly to lines of code in error messages with more precision than is possible in a compiled language. A compiler, due to translation, may not be able to accurately identify the origin of a compile-time error in the original source program by the time the error is detected. Run-time errors in compiled programs are similarly difficult to trace back to the source program because the target program has no knowledge of the original source program. Such run-time feedback can be invaluable to debugging a program. Therefore, the mechanics of testing and debugging are streamlined and cleaner using an interpreted, as opposed to a compiled, language.

Also, consider that a compiler involves three languages: the source and target languages, and the language in which the compiler is written. By contrast, an interpreter involves only two languages: the source language and the language in which the interpreter is written—sometimes called the *defining programming language* or the *host language*.

4.5 Influence of Language Goals on Implementation

The goals of a language (e.g., speed of execution, ease of development, safety) influence its design choices (e.g., static or dynamic bindings). Historically, both of these factors have had an influence on language implementation (e.g., interpretation or compilation). For instance, Fortran and C programs are intended to execute fast and, therefore, are compiled. The speed of the executable produced by a compiler is a direct result of the efficient decoding of machine instructions (vis-à-vis high-level statements) at run-time coupled with few semantic checks at run-time. Static bindings also support fast program execution. It is natural to implement a language designed to support static bindings through compilation because establishing those bindings and performing semantic checks for them can occur at compile time so they do not occupy CPU cycles at run-time—yielding a fast executable. A compiler for a language supporting static bindings need not generate code for performing semantic checks at run-time in the target executable.⁷

UNIX shell scripts, by contrast, are intended to be quick and easy to develop and debug; thus, they are interpreted. It is natural and easier to interpret programs in a language with dynamic bindings (e.g., identifiers that can be bound to values of any type at run-time), including Scheme, since the necessary semantic checks cannot be performed before run-time. Compiling programs written in languages with dynamic bindings requires generating code in the target executable for performing semantic checks at run-time. Interpreted languages can also involve static bindings. Scheme, for example, uses static scoping. If a language is implemented with an interpreter, the static bindings in a program written in that language do not present an opportunity to improve the run-time speed of the interpreted program as a compiler would. Therefore, the use of static bindings in an interpreted language must be justified by reasons other than improving run-time performance.

However, there is nothing intrinsic in a programming language (i.e., in its definition) that precludes it from being implemented through interpretation or compilation. For instance, we can build an interpreter for C, which is traditionally a compiled language. An interpretive approach to implementing C is contrary to the design goals of C (i.e., efficiency) and provides no reasonable benefit to justify the degradation in performance. Similarly, compilers for Scheme are available. The programming language Clojure is a dialect of Lisp that is completely dynamic, yet is compiled to Java bytecode and runs on the JVM. The time required for these run-time checks is tolerated because of flexibility that dynamic bindings lend to program development.⁸ Binding is the topic of Chapter 6.

In cases where an implementation provides both an interpreter and a compiler (to object code) for a language (e.g., Scheme), the interpreter can be used for (speedy and flexible) program development, while the compiler can be reserved for producing the final (fast-executing) production version of software.

7. Similarly, time spent optimizing object code at compile time results in a faster executable. This is a worthwhile trade-off because compilation is a “compile once, run repeatedly” proposition—once a program is stable, compilation is no longer performed.

8. The speed of compilation decreases with the generation of code for run-time checks as well.

Programming Exercises for Chapter 4

Table 4.2 presents the interpretation programming exercises in this chapter annotated with the prior exercises on which they build. Table 4.3 presents the features of the parsers used in each subpart of the programming exercises in this chapter.

Exercise 4.1 Reconsider the following *context-free grammar* defined in EBNF from Programming Exercise 3.5:

```

<program> ::= <program> <expr> \n | <expr> \n
<expr>    ::= <expr> + <expr>
<expr>    ::= <expr> * <expr>
<expr>    ::= - <expr>
<expr>    ::= <integer>
<integer> ::= 0 | 1 | 2 | 3 | ... | 231-1

```

PE	Description of Language	Extends	Start from											
			(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)
PE 4.1	Simple calculator	PE 3.5	3.5.a	3.5.b	4.1.a	4.1.b	3.5.d	3.5.e	3.5.f	3.5.g	4.1.e	4.1.f	4.1.g	4.1.h
PE 4.2	Simple boolean expressions	PE 3.7	3.7.a	3.7.b	4.2.a	4.2.b	3.7.d	3.7.e	3.7.f	3.7.g	4.2.e	4.2.f	4.2.g	4.2.h
PE 4.3	Extended boolean expressions	PE 3.8	3.8.a	3.8.b	4.3.a	4.3.b	N/A	N/A	N/A	N/A	4.3.e	4.3.f	4.3.g	4.3.h

Table 4.2 Interpretation Programming Exercises in This Chapter Annotated with the Prior Exercises on Which They Build (Key: PE = programming exercise.)

Subpart	R-D	S-R	Build Tree	Diagram	Decorate	Interpret
(a)	✓	×	×	×	×	✓
(b)	×	✓	×	×	×	✓
(c)	✓	×	✓	×	×	✓
(d)	×	✓	✓	×	×	✓
(e)	✓	×	×	✓	×	✓
(f)	×	✓	×	✓	×	✓
(g)	✓	×	✓	✓	×	✓
(h)	×	✓	✓	✓	×	✓
(i)	✓	×	×	×	✓	✓
(j)	×	✓	✓	×	✓	✓
(k)	✓	×	✓	×	✓	✓
(l)	×	✓	×	×	✓	✓

Table 4.3 Features of the Parsers Used in Each Subpart of the Programming Exercises in This Chapter (Key: R-D = recursive-descent; S-R = shift-reduce.)

where $\langle \text{expr} \rangle$ and $\langle \text{integer} \rangle$ are non-terminals and $+$, $*$, $-$, and $1, 2, 3, \dots, 2^{31}-1$ are terminals.

- (a) Extend your program from Programming Exercise 3.5.a to interpret programs. Normal precedence rules hold: $-$ has the highest, $*$ has the second highest, and $+$ has the lowest. Assume left-to-right associativity. The following is sample input and output for the expression evaluator ($>$ is simply the prompt for input and will be the empty string in your system):

```
> 2+3*4
14
> 2+3*-4
-10
> 2+3*a
"2+3*a" contains invalid lexemes and, thus, is not an expression.
> 2+*3*4
"2+*3*4" is not an expression.
```

Do not build a parse tree to solve this problem. Factor your program into a recursive-descent parser (i.e., solution to Programming Exercise 3.5.a) and an interpreter as shown in Figure 4.1.

- (b) Extend your program from Programming Exercise 3.5.b to interpret expressions as shown in Programming Exercise 4.1.a. Do not build a parse tree to solve this problem. Factor your program into a shift-reduce parser (solution to Programming Exercise 3.5.b) and an interpreter as shown in Figure 4.1.
- (c) Complete Programming Exercise 4.1.a, but this time build a parse tree and traverse it to evaluate the expression.
- (d) Complete Programming Exercise 4.1.b, but this time build a parse tree and traverse it to evaluate the expression.
- (e) Extend your program from Programming Exercise 3.5.d to interpret expressions as shown here:

```
> 2+3*4
((2)+((3)*(4))) = 14
> 2+3*-4
((2)+((3)*(-(4)))) = -10
> 2+3*a
"2+3*a" contains invalid lexemes and, thus, is not an expression.
> 2+*3*4
"2+*3*4" is not an expression.
```

- (f) Extend your program from Programming Exercise 3.5.e to interpret expressions as shown in Programming Exercise 4.1.e.
- (g) Extend your program from Programming Exercise 3.5.f to interpret expressions as shown in Programming Exercise 4.1.e.

- (h) Extend your program from Programming Exercise 3.5.g to interpret expressions as shown in Programming Exercise 4.1.e.
- (i) Complete Programming Exercise 4.1.e, but this time, rather than diagramming the expression, decorate each expression with parentheses to indicate the order of operator application and interpret expressions as shown here:

```
> 2+3*4
(2+(3*4)) = 14
> 2+3*-4
(2+(3*(-4))) = -10
> 2+3*a
"2+3*a" contains invalid lexemes and, thus, is not an expression.
> 2+*3*4
"2+*3*4" is not an expression.
```

- (j) Complete Programming Exercise 4.1.f with the same addendum noted in part i.
- (k) Complete Programming Exercise 4.1.g with the same addendum noted in part i.
- (l) Complete Programming Exercise 4.1.h with the same addendum noted in part i.

Exercise 4.2 Reconsider the following *context-free grammar* defined in BNF (not EBNF) from Programming Exercise 3.7:

```
<expr> ::= <expr> & <expr>
<expr> ::= <expr> | <expr>
<expr> ::= ~ <expr>
<expr> ::= <literal>
<literal> ::= t
<literal> ::= f
```

where t , f , $|$, $\&$, and $\sim$ are terminals that represent true, false, or, and, and not, respectively. Thus, sentences in the language defined by this grammar represent logical expressions that evaluate to true or false.

Complete Programming Exercise 4.1 (parts a–l) using this grammar, subject to all of the requirements given in that exercise. Specifically, build a parser and an interpreter to evaluate and determine the order in which operators of a logical expression are evaluated. Normal precedence rules hold: $\sim$ has the highest, $\&$ has the second highest, and $|$ has the lowest. Assume left-to-right associativity.

The following is a sample interactive session with the pure interpreter:

```
> f | t & f | ~t
false
> ~t | t | ~f & ~f & t & ~t | f
true
> f | t ; f | ~t
"f | t ; f | ~t" contains invalid lexemes and, thus, is not an expression.
> f | t & & f | ~t
"f | t & & f | ~t" is not an expression.
```


The following is a sample interactive session with the diagramming interpreter:

```
> f | t & f | ~t
(((f) | ((t) & (f))) | (~t)) is false.
> ~t | t | ~f & ~f & t & ~t | f
((((~t) | t) | (((~f) & (~f)) & t) & (~t))) | f) is true.
> f | t ; f
"f | t ; f" contains invalid lexemes and, thus, is not an expression.
> f | | t & ~t
"f | | t & ~t" is not an expression.
```

The following is a sample interactive session with the decorating (i.e., parentheses-for-operator-precedence) interpreter:

```
> f | t & f | ~t
((f | (t & f)) | (~t)) is false.
> ~t | t | ~f & ~f & t & ~t | f
((((~t) | t) | (((~f) & (~f)) & t) & (~t))) | f) is true.
> f | t ; f
"f | t ; f" contains invalid lexemes and, thus, is not an expression.
> f | | t & ~t
"f | | t & ~t" is not an expression.
```

Exercise 4.3 Reconsider the following *context-free grammar* defined in BNF (not EBNF) from Programming Exercise 3.8:

```

<program> ::= (<declarations>, <expr>)
<declarations> ::= []
<declarations> ::= [ <varlist> ]
<varlist> ::= <var>
<varlist> ::= <var>, <varlist>
<expr> ::= <expr> & <expr>
<expr> ::= <expr> | <expr>
<expr> ::= ~ <expr>
<expr> ::= <literal>
<expr> ::= <var>
<literal> ::= t
<literal> ::= f
<var> ::= a...e
<var> ::= g...s
<var> ::= u...z
```

where t , f , $|$, $\&$, and $\sim$ are terminals that represent true, false, or, and, and not, respectively, and all lowercase letters except for f and t are terminals, each representing a variable. Each variable in the variable list is bound to true in the expression. Any variable used in any expression not contained in the variable list is assumed to be false. Thus, programs in the language defined by this grammar represent logical expressions, which can contain variables, that can evaluate to true or false.

Complete Programming Exercise 4.1 (parts a–d and i–l) using this grammar, subject to all of the requirements given in that exercise.

Specifically, build a parser and an interpreter to evaluate and determine the order in which operators of a logical expression with variables are evaluated. Normal precedence rules hold: $\sim$ has the highest, $\&$ has the second highest, and $|$ has the lowest. Assume left-to-right associativity.

The following is a sample interactive session with the pure interpreter:

```
> ([], f | t & f | ~t)
false
> ([], ~t | t | ~f & ~f & t & ~t | f)
true
> ([p,q], ~t | p | ~e & ~f & t & ~q | r)
true
> ([], f | t ; f)
"([], f | t ; f)" contains invalid lexemes and, thus, is not a program.
> ([], f | | t & ~t)
"([], f | | t & ~t)" is not a program.
```

The following is a sample interactive session with the parentheses-for-operator-precedence interpreter:

```
> ([], f | t & f | ~t)
((f | (t & f)) | (~t)) is false.
> ([], ~t | t | ~f & ~f & t & ~t | f)
((((~t) | t) | (((~f) & (~f)) & t) & (~t))) | f) is true.
> ([p,q], ~t | p | ~e & ~f & t & ~q | r)
((((~t) | p) | (((~e) & (~f)) & t) & (~q))) | r) is true.
> ([], f | t ; f)
"([], f | t ; f)" contains invalid lexemes and, thus, is not a program.
> ([], f | | t & ~t)
"([], f | | t & ~t)" is not a program.
```

Notice that this language is context-sensitive because variables must be declared before they are used. For example, $([a], b | t)$ is syntactically, but not semantically, valid.

4.6 Thematic Takeaways

- Languages lend themselves to implementation through either interpretation or compilation, but usually not through both.
- An interpreter or compiler for a computer language creates a virtual machine for the language of the source program (i.e., a computer that virtually understands the language).
- Compilers and interpreters are often complementary in terms of their advantages and disadvantages. This leads to the conception of hybrid implementation systems.

- Compilation results in a fast executable; interpretation results in slow execution because it takes longer to decode high-level program statements than machine instructions.
- Interpreters support run-time flexibility in the source language, which is often less practical in compiled languages.
- Trade-offs between speed of execution and speed of development have been factors in the evolution and implementation of programming languages.
- The goals of a language (e.g., speed of execution, speed of development) and its design choices (e.g., static or dynamic bindings) have historically influenced the implementation approach of the language (e.g., interpretation or compilation).

4.7 Chapter Summary

There are a variety of ways to implement a programming language. All language implementations have a syntactic component (or front end) that determines whether the source program is valid and, if so, produces an abstract-syntax tree. Language implementations vary in how they process this abstract-syntax tree.

Two traditional approaches to language implementation are compilation and interpretation. A compiler translates the abstract-syntax tree through a series of transformations into another representation (e.g., assembly code) typically closer to the instruction set architecture of the target processor intended to execute the program. The output of a compiler is a version of the source program in a different language. An interpreter traverses the abstract-syntax tree to evaluate and directly execute the program. The input to an interpreter is both the source program to be executed and the input of that source program. The output of an interpreter is the output of the source program. Ultimately, the final representation (e.g., x86 object code) produced by a compiler (or assembler) must be interpreted—traditionally by a hardware interpreter (e.g., an x86 processor).

Languages in which the final representation produced by a compiler is interpreted by a software interpreter are implemented using a hybrid system. For instance, the Java compiler translates Java source code to Java bytecode, and the Java bytecode interpreter then interprets the Java bytecode to produce program output. Just as a compiler can produce a series of intermediate representations of the original source program en route to a final representation, a source program can be interpreted through a series of software interpreters (i.e., the source program is interpreted by a software interpreter, which is itself interpreted by a software interpreter, and so on). As a corollary, compilers and interpreters are mutually dependent on each other. A compiler is dependent on either a hardware or software interpreter; a software interpreter is dependent on a compiler so that the interpreter itself can be translated into object code and run.

Compilers and interpreters are often complementary in terms of their advantages and disadvantages—hence the conception of hybrid implementation systems. The primary advantage of compilation is production of a fast executable. Interpretation results in slow execution because it takes longer to decode (and

re-decode) high-level program statements than machine instructions. However, interpreters support run-time flexibility in the source language, which is often less practical in compiled languages. The interplay of language goals (e.g., speed of execution, speed of development), language design choices (e.g., static or dynamic bindings), and execution environment (e.g, WWW) have historically influenced both the evolution and the implementation of programming languages.

4.8 Notes and Further Reading

For a more detailed, internal view into all of the phases of execution through compilation and the interfaces between them, we refer the reader to Appel (2004, Figure 1.1, p. 4).

Chapter 5

Functional Programming in Scheme

A functional programming language gives a simple model of programming: one value, the result, is computed on the basis of others, the inputs.

— Simon Thompson, *Haskell: The Craft of Functional Programming* (2007)

The spirit of Lisp hacking can be expressed in two sentences. Programming should be fun. Programs should be beautiful.

— Paul Graham, *ANSI Common Lisp* (1996)

[L]earning Lisp will teach you more than just a new language—it will teach you new and more powerful ways of thinking about programs.

— Paul Graham, *ANSI Common Lisp* (1996)

A minute to learn . . . A lifetime to master.

— Slogan for the game *Othello*

FUNCTIONAL programs operate by returning values rather than modifying variables—which is how imperative programs work. In other words, expressions (all of which return a value) rather than statements are used to affect computation. There are few statements in functional programs, if any. As a result, there are few or no side effects in functional programs—of course, there is I/O—so bugs have only a local effect. In this chapter, we study functional programming in the context of the Scheme programming language.

5.1 Chapter Objectives

- Foster a recursive-thought process toward program design and implementation.
- Understand the fundamental tenets of functional programming for practical purposes.
- Explore techniques to improve the efficiency of functional programs.
- Demonstrate by example the ease with which data structures and programming abstractions are constructed in functional programming.
- Establish an understanding of programming in Scheme.

5.2 Introduction to Functional Programming

Functional programming has its basis in λ -calculus and involves a set of tenets, including the use of a primitive list data structure, discussed here.

5.2.1 Hallmarks of Functional Programming

In languages supporting a functional style of programming, functions are first-class entities (i.e., functions are treated as *values*) and often have types associated with them—just as one might associate the type `int` with a variable `i` in an imperative program. Recall that a *first-class entity* is an object that can be stored, passed as an argument, and returned as a value. Since all functions must return a value, there is no distinction between the terms subprogram, subroutine, procedure, and function in functional programming. (Typically, the distinction between a function and a procedure is that a *function* returns a value [e.g., `int f(int x)`], while a *procedure* does not return a value and is typically evaluated for side effect [e.g., `void print(int x)`].)¹ Recursion, rather than iteration, is the primary mechanism for repetition. Languages supporting functional programming often use automatic garbage collection and usually do not involve direct manipulation of pointers by the programmer. (Historically, languages supporting functional programming were considered languages for artificial intelligence, but this is no longer the case.)

5.2.2 Lambda Calculus

Functional programming is based on λ -calculus—a mathematical theory of functions and formal model for computation (equivalent to a Turing machine) developed by mathematician and logician Alonzo Church in 1928–1929 and published in 1932.² The λ -calculus is a language that is helpful in the study of programming languages. The following is the grammar of λ -calculus.

1. This distinction may be a remnant of the Pascal programming language, which used the `function` and `procedure` lexemes in the definition of a function and a procedure, respectively.

2. Alonzo Church was Alan Turing's PhD advisor at Princeton University from 1936 to 1938.

$$\begin{aligned}
\langle \text{expression} \rangle &::= \langle \text{identifier} \rangle \\
\langle \text{expression} \rangle &::= (\text{lambda } (\langle \text{identifier} \rangle) \langle \text{expression} \rangle) \\
\langle \text{expression} \rangle &::= (\langle \text{expression} \rangle \langle \text{expression} \rangle)
\end{aligned}$$

These three production rules correspond to an identifier, a function definition, and a function application (respectively, from top to bottom). Formally, this is called the *untyped λ -calculus*.

5.2.3 Lists in Functional Programming

Lists are the primitive, built-in data structure used in functional programming. All other data structures can be constructed from lists. A *list* is an ordered collection of items. (Contrast a list with a *set*, which is an unordered collection of unique items [i.e., without duplicates], or a *bag*, which is an unordered collection of items, possibly with duplicates.)

We need to cultivate the habit of thinking recursively and, in particular, specifying data structures recursively. Formally, a list is either empty or a pair of pointers: one to the head of the list and one to the tail of the list, which is also a list.

$$\begin{aligned}
\langle \text{list} \rangle &::= \text{empty} \\
\langle \text{list} \rangle &::= \langle \text{element} \rangle \langle \text{list} \rangle
\end{aligned}$$

Conceptual Exercises for Section 5.2

Exercise 5.2.1 A fictitious *language Q* that supports functional programming contains the following production in its grammar to specify the syntax of its *if* construct:

$$\langle \text{expression} \rangle ::= (\text{if } \langle \text{expression} \rangle \langle \text{expression} \rangle \langle \text{expression} \rangle)$$

The semantics of an expression generated using this rule in *Q* are as follows: If the value of the first expression (on the right-hand side) is true, return the value of the second expression (on the right-hand side). Otherwise, return the value of the third expression (on the right-hand side). In other words, the third expression on the right-hand side (the “else” part) is mandatory.

Why does language *Q* *not* permit the third expression on the right-hand side to be optional? In other words, why is the following production rule absent from the grammar of *Q*?

$$\langle \text{expression} \rangle ::= (\text{if } \langle \text{expression} \rangle \langle \text{expression} \rangle)$$

Exercise 5.2.2 Notice that there is no direct provision in the λ -calculus grammar for integers. Investigate the concept of *Church Numerals* and define the integers 0, 1, and 2 in λ -calculus. When done, define an increment function in λ -calculus, which adds one to its only argument and returns the result. Also, define addition and multiplication functions in λ -calculus, which adds and multiplies its two

arguments and returns the result, respectively. You may only use the three production rules in λ -calculus to construct these numbers and functions.

Exercise 5.2.3 Write a simple expression in λ -calculus that creates an infinite loop.

5.3 Lisp

5.3.1 Introduction

Lisp (List processing)³ was developed by John McCarthy and his students at MIT in 1958 for artificial intelligence (McCarthy 1960). (Lisp is, along with Fortran, one of the two oldest programming languages still in use.) An understanding of Lisp will both improve your ability to learn new languages with ease and help you become a more proficient programmer in your language of choice. In this sense, Lisp is the Latin of programming languages.

There are two dialects of Lisp: Scheme and Common Lisp. Scheme can be used for teaching language concepts; Common Lisp is more robust and often preferred for developing industrial applications. Scheme is an ideal programming language for exploring language semantics and implementing language concepts, and we use it in that capacity particularly in Chapters 6, 8, 12, and 13. In this text, we use the Racket programming language, which is based on Scheme, for learning Lisp. Racket is a dialect of Scheme well suited for this course of study.

Much of the power of Lisp can be attributed to its uniform representation of Lisp program code and data as lists. A Lisp program is expressed as a Lisp list. Recall that lists are the fundamental and only primitive Lisp data structure. Because the ability to leverage the power Lisp derives from this uniform representation, we must first introduce Lisp lists (i.e., data).

5.3.2 Lists in Lisp

Lisp has a simple, uniform, and consistent syntax. The only two syntactic entities are atoms and lists. Lists can contain atoms or lists, or both. Lists are heterogeneous in Lisp, meaning they may contain values of different types. Heterogeneous lists are more flexible than homogeneous lists. We can represent a homogeneous list with a heterogeneous list, but the reverse is not possible. Remember, the syntax (i.e., representation) for Lisp code and data is the same. The following are examples of Lisp lists:

```
(1 2 3)
(x y z)
(1 (2 3))
((x) y z)
```

Here, 1, 2, 3, x, y, and z are atoms from which these lists are constructed. The lists (1 (2 3)) and ((x) y z) each contain a sublist.

3. Some jokingly say Lisp stands for Lots of Irritating Superfluous Parentheses.

Formally, Lisp syntax (programs or data) is made up of S-expressions (i.e., symbolic expressions). “[A]n *S-expression* is either an atom or a (possibly empty) list of S-expressions” (Friedman and Felleisen 1996a, p. 92). An S-expression is defined with BNF as follows:

```

<symbol-expr> ::= <symbol>
<symbol-expr> ::= <s-list>
      <s-list> ::= ()
      <s-list> ::= (<list-of-symbol-expr>)
<list-of-symbol-expr> ::= <symbol-expr>
<list-of-symbol-expr> ::= <symbol-expr> <list-of-symbol-expr>

```

The following are more examples of S-expressions:

```

(1 2 3)
(x 1 y 2 3 z)
(((Nothing))) ((will) (()()) (come ()) (of nothing)))

```

Conceptual Exercises for Section 5.3

Exercise 5.3.1 Are arrays in C++ *homogeneous*? Explain.

Exercise 5.3.2 Are arrays in Java *heterogeneous*? Explain.

Exercise 5.3.3 Describe an *<s-list>* using English, not BNF. Be complete.

5.4 Scheme

The Scheme programming language was developed at the MIT AI Lab by Guy L. Steele and Gerald Jay Sussman between 1975 and 1980. Scheme predates Common Lisp and influenced its development.

5.4.1 An Interactive and Illustrative Session with Scheme

The following is an interactive session with Scheme:⁴

```

1  > 1
2  1
3  > 2
4  2
5  > 3
6  3
7  > +
8  #<procedure: +>
9  > #t
10 #t
11 > #f
12 #f

```

4. We use the Racket language implementation in this text when working with Scheme code. See <https://racket-lang.org>.

```

13 > (+ 1 2)
14 3
15 > (+ 1 2 3)
16 6
17 > (lambda (x) (+ x 1))
18 #<procedure>
19 > ((lambda (x) (+ x 1)) 2)
20 3
21 > (define increment (lambda (x) (+ x 1)))
22 > increment
23 #<procedure:increment>
24 > (increment 2)
25 3
26 ;;; a power function
27 > (define pow
28 >   (lambda (x n)
29 >     (cond
30 >       ((zero? n) 1)
31 >       (else (* x (pow x (- n 1)))))))

```

As shown in this session, the Scheme interpreter operates as a simple interactive *read-eval-print* loop (REPL; sometimes called an *interactive top-level*). Literals evaluate as themselves (lines 1–12). The atoms `#t` and `#f` represent true and false, respectively. More generally, to evaluate an atom, the interpreter looks up the atom in the environment and returns the value associated with it. A *referencing environment* is a set of name–value pairs that associates symbols with their current bindings at any point in a program in a language implementation (e.g., on line 19 of the interactive session the symbol `x` is bound to the value 2 in the body of the `lambda` expression). Literals do not require a lookup in the environment. On line 7, we see that the symbol `+` is associated with a procedure in the environment. Lisp and Scheme use prefix notation for expressions (lines 13, 15, 19, and 24). C uses prefix notation for function calls [e.g., `f(x)`], but infix notation for expressions (e.g., `2+3*4`). Lisp and Scheme, by contrast, consistently use prefix notation for all expressions [e.g., `(f x)` and `(+ 2 (* 3 4))`].

The reserved word `lambda` on line 17 introduces a function. Specifically, an anonymous (i.e., nameless) function (also called a *constant function*, *literal function*, or *lambda expression*) is defined in line 17. Readers may be more familiar with accessing anonymous data in programs through references (e.g., `Circle c = new Circle();` in Java). Languages supporting functional programming extend that anonymity to functions. We can also invoke functions literally, as is done on line 19. Support for anonymous functions has been implemented in multiple contemporary languages, including Python, Go, and Java.

Notice that this function definition (line 17) follows the second production rule in the grammar of λ -calculus. The list immediately following the `lambda` is the parameter list of the function, and the list immediately following the parameter list is the body of the function. This function increments its argument by 1 and returns the result. It is a literal function and the interpreter returns it as such (line 18); a lookup in the environment is unnecessary. Line 19 defines the same literal function, but also invokes it with the argument 2. Notice that this line of code conforms to the third production rule in the grammar of λ -calculus (i.e., functional application). The result of the application is 3 (line 20). The reserved word `define`

binds (in the environment) the identifier immediately following it with the result of the evaluation of the expression immediately following the identifier. Thus, line 21 associates (in the environment) the identifier `increment` with the function defined on line 21. Lines 22–25 confirm that the function is bound to the identifier `increment`. Line 24 invokes the `increment` function by name; that is, now that the function name is in the environment, it need not be used literally.

Lines 27–31 define a function `pow` that, given a base `x` and non-negative exponent `n`, returns the base raised to the exponent (i.e., x^n). This function definition introduces the control construct `cond`, which works as follows. It accepts a series of lists and evaluates the first element of each list (from top to bottom). As soon as the interpreter finds a first element that evaluates to true, it evaluates the tail of that list and returns the result. In the context of `cond`, `else` always evaluates to true. The built-in Scheme function `zero?` returns `#t` if its argument is equal to zero and `#f` otherwise. Functions with a boolean return type (i.e., those that return either `#t` or `#f`) are called *predicates*. Built-in predicates in Scheme typically end with a question mark (?); we recommend that the programmer follow this convention when naming user-defined functions as well.

Two types of parameters exist: actual and formal. *Formal parameters* (also known as *bound variables* or simply *parameters*) are used in the declaration and definition of a function. Consider the following function definition:

```
1 // x and y are the formal parameters
2 int add (int x, int y) {
3     return (x+y);
4 }
```

The identifiers `x` and `y` on line 2 are formal parameters. *Actual parameters* (or *arguments*) are passed to a function in an invocation of a function. For instance, when invoking the preceding function as `add(a, b)`, the identifiers `a` and `b` are actual parameters. Throughout this text, we refer to identifiers in the declaration of a function as *parameters* (of the function) and values passed in a function call as *arguments* (to the function).

Notice that the `pow` function uses recursion for repetition. A recursive solution often naturally mirrors the specification of the problem. Cultivating the habit of thinking recursively can take time, especially for those readers from an imperative or object-oriented background. Therefore, we recommend you follow these two steps to develop a recursive solution to any problem.

1. Identify the smallest instance of the problem—the *base case*—and solve the problem for that case only.
2. Assume you already have a solution to the penultimate (in size) instance of the problem named $n - 1$. Do not try to solve the problem for that instance. Remember, you are assuming it is already solved for that instance. Now given the solution for this $n - 1$ case, extend that solution for the case n . This extension is much easier to conceive than an original solution to the problem for the $n - 1$ or n cases.

For instance,

1. The base case of the `pow` function is $n = 0$, for which the solution is 1.
2. Assuming we have the solution for the case $n - 1$, all we have to do is multiply that solution by x to obtain the solution for the case n .

This is the crux of recursion (see *Design Guideline 1: General Pattern of Recursion* in Table 5.7 at the end of the chapter). With time and practice, you will master this technique for recursive-function definition and no longer need to explicitly follow these two steps because they will become automatic to you. Eventually, you will become like those who learned Scheme as a first programming language, and find iterative thinking and iterative solutions to problems more difficult to conceive than recursive ones.

At this point, a cautionary note is necessary. We advise against solving problems iteratively and attempting a translation into a recursive style. Such an approach is unsustainable. (Anyone who speaks a foreign natural language knows that it is impossible to hold a synchronous and effortlessly flowing conversation in that language while thinking of how to respond in your native language and translating the response into the foreign language while your conversation partner is speaking.) Recursive *conception* of problems and recursive *thinking* are fundamental prerequisites for functional programming.

It is also important to note that in Lisp and Scheme, values (not identifiers) have types. In a sense, Lisp is a *typeless* language—any value can be bound to any identifier. For instance, in the `pow` function, the base `x` has not been declared to be of any specific type, as is typically required in the signature of a function declaration or definition. The identifier `x` can be bound to value of any time at run-time. However, only a binding to an integer or a real number will produce a meaningful result due to the nature of the multiplication (`*`) function. The ability to bind any identifier to any type at run-time—a concept called *manifest typing*—relieves the programmer from having to declare types of variables, requires less planning and design, and provides a more flexible, malleable implementation. (Manifest typing is a feature that supports the oil painting metaphor discussed in Chapter 1.)

Notice there are no side effects in the session with the Scheme interpreter. Notice also that a semicolon (`;`) introduces a comment that extends until the end of the line (line 26). The short interactive session demonstrates the crux of functional programming: evaluation of expressions that involve storing and retrieving items from the environment, defining functions, and applying them to arguments.

Notice that the λ -calculus grammar, given in Section 5.2.2., does not have a provision for a `lambda` expression with more than one argument. (Functions that take one, two, three, and n arguments are called *unary*, *binary*, *ternary*, and *n -ary* functions, respectively.) That is because λ -calculus is designed to provide the minimum constructs necessary for describing computation. In other words, λ -calculus is a mathematical model of computation, not a practical implementation. Any `lambda` expression in Scheme with more than one argument

can be mechanically converted to a series of nested `lambda` expressions in λ -calculus, each of which has only one argument. For instance,

```
> ((lambda (x y)
>   (+ x y)) 1 2)
3
```

is semantically equivalent to

```
> ((lambda (x)
>   ((lambda (y)
>     (+ x y)) 2)) 1)
3
```

Thus, syntax for defining a function with more than one argument is syntactic sugar. Recall that *syntactic sugar* is special, typically terse, syntax in a language that serves only as a convenient method for expressing syntactic structures that are traditionally represented in the language through uniform and often long-winded syntax. (To help avoid syntax errors, we recommend using an editor that matches parentheses [e.g., `vi` or `emacs`] while programming in Scheme.)

5.4.2 Homoiconicity: No Distinction Between Program Code and Data

Much of the power of Lisp is derived from its uniform representation of program code and data in syntax and memory. Lisp programs are S-expressions. Because the only primitive data structure in Lisp is a list (represented as an S-expression), Lisp data is represented as an S-expression. A language that does not make a distinction between programs and data objects is called a *homoiconic* language. In other words, a language whose programs are represented as a data structure of a primitive (data) type in the language itself is a homoiconic language; that is, it has the property of *homoiconicity*.⁵ Prolog, Tcl, Julia, and XSLT are also homoiconic languages, while Go, Java, C++, and Haskell are not. Lisp was the first homoiconic language, and much of the power of Lisp results from its inherent homoiconic nature. Homoiconicity leads to some compelling implications, including the ability to change language semantics. We discuss the advantages of a homoiconic language in Section 12.9, which will be more palatable after we have acquired experience with building language interpreters in Part III. For now it suffices to say that since a Lisp program is represented in the same way as Lisp data, a Lisp program can easily read or write another Lisp program.

Given the uniform representation of program code and data in Lisp, programmers must indicate to the interpreter when to evaluate an S-expression as code and when to treat it as data—because otherwise the two are indistinguishable. The built-in Scheme function `quote` prevents the interpreter

5. The words *homo* and *icon* are of Greek origin and mean *same* and *representation*, respectively.

from evaluating an S-expression; that is, adding quotes protects expressions from evaluation. Consider the following transcript of a session with Scheme:

```
1 > (quote a)
2 a
3 > 'b
4 b
5 > '(a b c d)
6 (a b c d)
7 > (quote (1 2 3 4))
8 (1 2 3 4)
```

The `'` symbol (line 5) is a shorthand notation for `quote`—the two can be used interchangeably. For purposes of terseness of exposition, we exclusively use `'` throughout this text. If the `a` and `b` (on lines 1 and 3, respectively) were not quoted, the interpreter would attempt to retrieve a value for them in the language environment. Similarly, if the lists on lines 5 and 7 were not quoted, the interpreter would attempt to evaluate those S-expressions as functional applications (e.g., the function `a` applied to the arguments `b`, `c`, and `d`). Thus, you should use the `quote` function if you want an S-expression to be treated as data and not code; do not use the `quote` function if you want an S-expression to be evaluated as program code and not to be treated as data. Symbols do not evaluate to themselves unless they are preceded with a quote. Literals (e.g., `1`, `2.1`, `"hello"`) need not be quoted.

Conceptual Exercise for Section 5.4

Exercise 5.4.1 Two criteria on which to evaluate programming languages are *readability* and *writability*. For instance, the verbosity in COBOL makes it a readable, but not a writable, language. By comparison, all of the parentheses in Lisp make it a neither readable nor writable. Why did the language designers of Lisp decide to include so many parentheses in its syntax? What advantage does such a syntax provide at the expense of compromising readability and writability?

Programming Exercises for Section 5.4

Exercise 5.4.2 Define a *recursive* Scheme function `square` that accepts only a *positive* integer `n` and returns the square of `n` (i.e., n^2). Your definition of `square` must not contain a `let`, `let*`, or `letrec` expression or any other Scheme constructs that have yet to be introduced. Do not use any user-defined auxiliary, helper functions.

Examples:

```
> (square 1)
1
> (square 2)
4
> (square 3)
9
> (square 4)
16
```

Definitions such as the following are *not* recursive:

```
(define square
  (lambda (n)
    (* n n)))

(define square
  (lambda (n)
    (cond
      ((eqv? 1 n) 1)
      (else (* (* n n) (square 1))))))
```

To be recursive, a function must not only call itself, but must do so in a way such that each successive recursive call reduces the problem to a smaller problem.

Exercise 5.4.3 Define a *recursive* Scheme function `cube` that accepts only an integer x and returns x^3 . Do not use any user-defined auxiliary, helper functions. Use only three lines of code. *Hint*: Define a recursive squaring function first (Programming Exercise 5.4.2).

Exercise 5.4.4 Define a Scheme function `applytoall` that accepts two arguments, a function and a list, applies the function to every element of the list, and returns a list of the results.

Examples:

```
> (applytoall (lambda (x) (* x x)) '(1 2 3 4 5 6))
(1 4 9 16 25 36)

> (applytoall (lambda (x) (list x x)) '(hello world))
((hello hello) (world world))
```

This pattern of recursion is encapsulated in a universal *higher-order function*: `map`.

5.5 cons Cells: Building Blocks of Dynamic Memory Structures

To develop functions that are more sophisticated than `pow`, we need to examine how lists are represented in memory. Such an examination helps us conceptualize and conceive abstract data structures and design algorithms that operate on and manipulate those structures to solve a variety of problems. In the process, we also consider how we can use BNF to define data structures inductively. (Recall that in Lisp, code and data are one and the same.) In a sense, all programs are interpreters, so the input to those programs must conform to the grammar of some language. Therefore, as programmers, we are also language designers. A well-defined recursive data structure naturally lends itself to the development of recursive algorithms that operate on that structure. An important theme of a course on data structures and algorithms is that data structures and algorithms are natural reflections of each other. In turn, “when defining a program based on structural induction, the structure of the program should be patterned after

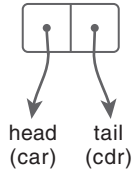


Figure 5.1 List box representation of a `cons` cell.

the structure of the data” (Friedman, Wand, and Haynes 2001, p. 12). We move onward, bearing these two themes in mind.

5.5.1 List Representation

In Lisp, a list is represented as a `cons` cell, which is a pair of pointers (Figure 5.1):

- a pointer to the head of the list *as an atom or a list* (known as the `car`⁶)
- a pointer to the tail of the list *as a list* (known as the `cdr`)

The function `cons` constructs (i.e., allocates) new memory—it is the Scheme analog of `malloc(16)` in C (i.e., it allocates memory for two pointers of 8 bytes each). The running time of `cons` is constant [i.e., $O(1)$]. Cons cells are the building blocks of dynamic memory structures, such as binary trees, that can grow and shrink at run-time.

5.5.2 List-Box Diagrams

A `cons` cell can be visualized as a pair of horizontally adjacent square boxes (Figure 5.1). The box on the left contains a pointer to the `car` (the head) of the list, while the box on the right holds a pointer to the `cdr` (the tail) of the list. Syntactically, in Scheme (and in this text), a full stop (`.`) is used to denote the vertical partition between the boxes. For instance, the list `'(a b)` is equivalent to the list `(a . (b))` and both are represented in memory the same way. The diagram in Figure 5.2, called a *list-box*, depicts the memory structure created for this list, where a `cdr` box with a diagonal line from the bottom left corner to the top right corner denotes the empty list [i.e., `()`]. Similarly, Figure 5.3 illustrates the list `'(a b c)`. The *dot notation* makes the distinction between the `car` and `cdr` explicit. When the `cdr` of a list is not a list, the list is not a *proper list* and is called an *improper list*. The list `'(a . b)` (Figure 5.4) is an improper list.

The dot notation also helps reveal another important and pioneering aspect of Lisp—namely, that *everything is a pointer*, even though nothing appears to be because of *implicit pointer dereferencing*. This is yet another example of uniformity and consistency in the language. Uniform and consistent languages are easy to

6. The names of the functions `car` and `cdr` are derived from the IBM 704 computer, the computer on which Lisp was first implemented (McCarthy 1981). A word on the IBM 704 had two fields, named address and decrement, which could each store a memory address. It also had two machine instructions named `CAR` (contents of address register) and `CDR` (contents of decrement register), which returned the values of these fields.

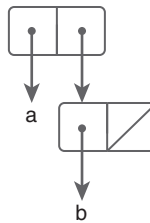


Figure 5.2 $'(a\ b) = '(a\ .\ (b))$

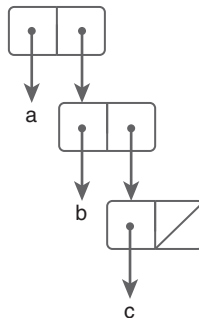


Figure 5.3 $'(a\ b\ c) = '(a\ .\ (b\ c)) = '(a\ .\ (b\ .\ (c)))$

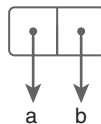


Figure 5.4 $'(a\ .\ b)$

learn and use. English is a difficult language to learn because of the numerous exceptions to the voluminous set of rules (e.g., *i* before *e* except after *c*⁷). Similarly, many programming languages are inconsistent in a variety of aspects. For instance, all objects in Java must be accessed through a reference (i.e., you cannot have a direct handle to an object in Java); moreover, Java uses implicit dereferencing. However, Java is not entirely uniform in this respect because only objects—not primitives such as `ints`—are accessed through references. This is not the case in C++, where a programmer can access an object directly or through a reference.

Understanding how dynamic memory structures are represented through list-box diagrams is the precursor to building and manipulating abstract data structures. Figures 5.5–5.8 depict the list-boxes for the following lists:

$'((a)\ (b)\ ((c)))$

$'((a)\ b)\ c)$

7. There are more exceptions to this rule than adherents.

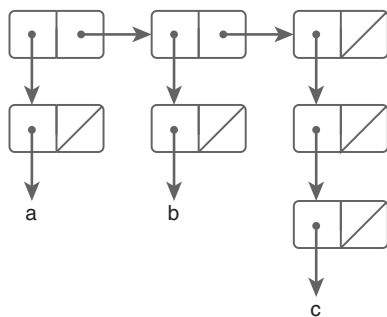


Figure 5.5 ' ((a) (b) ((c))) = ' ((a) . ((b) ((c)))) = ' ((a) . ((b) . ((c)))))

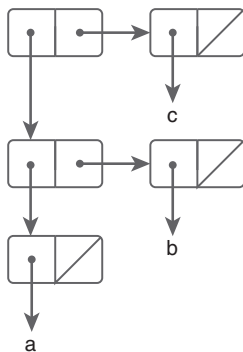


Figure 5.6 ' (((a) b) c)

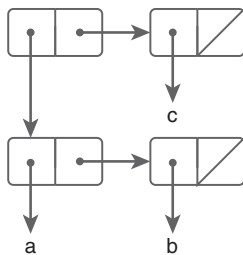


Figure 5.7 ' ((a b) c) = ' (((a) b) . (c)) = ' (((a) . (b)) . (c))

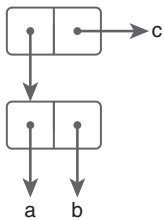


Figure 5.8 ' ((a . b) . c)

```
' ( (a b) c)
' ( (a . b) . c)
```

Note that Figures 5.6 and 5.8 depict improper lists. The following transcript illustrates how the Scheme interpreter treats these lists. The `car` function returns the value pointed to by the left side of the list-box, and the `cdr` function returns the value pointed to by the right side of the list-box.

```
> ' (a b)
(a b)
> ' (a . (b))
(a b)
>
> ' (a b c)
(a b c)
>
> (car ' (a b c))
a
> (cdr ' (a b c))
(b c)
>
> ' (a . (b c))
(a b c)
>
> (car ' (a . (b c)))
a
> (cdr ' (a . (b c)))
(b c)
>
> ' (a . (b . (c)))
(a b c)
>
> (car ' (a . (b . (c))))
a
> (cdr ' (a . (b . (c))))
(b c)
>
> ' (a . b)
(a . b)
>
> (car ' (a . b))
a
> (cdr ' (a . b))
b
>
> ' ((a) (b) ((c)))
((a) (b) ((c)))
>
> (car ' ((a) (b) ((c))))
(a)
> (cdr ' ((a) (b) ((c))))
((b) ((c)))
>
> ' ((a) . ((b) ((c))))
((a) (b) ((c)))
>
> (car ' ((a) . ((b) ((c)))))
(a)
```

```

> (cdr '((a) . ((b) ((c)))))
((b) ((c)))
>
> '((a) . ((b) . ((c)))))
((a) (b) ((c)))
>
> (car '((a) . ((b) . ((c)))))
(a)
> (cdr '((a) . ((b) . ((c)))))
((b) ((c)))
>
> '((a) b c)
((a) b c)
>
> (car '((a) b c))
((a) b)
> (cdr '((a) b c))
(c)
>
> '((a) b . (c))
((a) b c)
>
> (car '((a) b . (c)))
((a) b)
> (cdr '((a) b . (c)))
(c)
>
> '((a) . (b)) . (c)
((a) b c)
>
> (car '((a) . (b)) . (c))
((a) b)
> (cdr '((a) . (b)) . (c))
(c)
>
> '(a . b) . c
((a . b) . c)
>
> (car '(a . b) . c)
(a . b)
> (cdr '(a . b) . c)
c

```

```

1 > (list 'a 'b 'c)
2 (a b c)

```

When working with lists, always follow *The Laws of car, cdr, and cons* (Friedman and Felleisen 1996a):

The Law of `car`: The primitive `car` is defined only for non-empty lists (p. 5).

The Law of `cdr`: The primitive `cdr` is only defined for non-empty lists. The `cdr` of a non-empty list is always another list (p. 7).

The Law of `cons`: The primitive `cons` accepts two arguments. The second argument to `cons` must be a list [(so to construct only proper lists)]. The result is a list (p. 9).

Conceptual Exercise for Section 5.5

Exercise 5.5.1 Give the *list-box* notation for the following lists:

(a) (a (b (c (d))))

(b) (a (b) (c (d)) (e) (f))

(c) (((a) b) c) d)

(d) ((a . b) (c . d))

5.6 Functions on Lists

Armed with an understanding of (1) the core computational model in Lisp— λ -calculus; (2) the recursive specifications of data structures and recursive definitions of algorithms; and (3) the representation of lists in memory, we are prepared to develop functions that operate on data structures.

5.6.1 A List length Function

Consider the following function `length1`,⁸ which given a list, returns the length of the list:

```
(define length1
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (+ 1 (length1 (cdr l)))))))
```

The built-in Scheme predicate `null?` returns true if its argument is an empty list and false otherwise. The built-in Scheme predicate `empty?` can be used for this purpose as well.

Notice that the pattern of the recursion in the preceding function is similar to that used in the `pow` function in Section 5.4.1. Defining functions in Lisp can be viewed as pattern application—recognizing the pattern to which a problem fits, and then adapting that pattern to the details of the problem (Friedman and Felleisen 1996a).

5.6.2 Run-Time Complexity: `append` and `reverse`

A built-in Scheme function that is helpful for illustrating issues of efficiency with lists is `append`:⁹

8. When defining a function in Scheme with the same name as a built-in function (e.g., `length`), we use the name of the built-in function with a `1` appended to the end of it as the name of the user-defined function (e.g., `length1`), where appropriate, to avoid any confusion and/or clashes (in the interpreter) with the built-in function.

9. The function `append` is built into Scheme and accepts an arbitrary number of arguments, all of which must be proper lists. The version we define is named `append1`.

```

1  (define append1
2    (lambda (x y)
3      (cond
4        ((null? x) y)
5        (else (cons (car x) (append1 (cdr x) y))))))

```

Intuitively, `append` works by recursing through the first list and `cons`ing the `car` of each progressively smaller, first list to the appendage of the `cdr` of each progressively smaller list with the second list. Recall that the `cons` function is a constant operation—it allocates space for two pointers and copies the pointers of its two arguments into those fields—and recursion is not involved. The `append` function works differently: It deconstructs the first list and creates a new `cons` cell for each element. In other words, `append` makes a complete copy of its first argument. Therefore, the run-time complexity of `append` is linear [or $O(n)$] in the size of the first list. Unlike the first list, which is not contained in the resulting list (i.e., it is automatically garbage collected), the `cons` cell of the second list remains intact and is present in the resulting appended list—it is the `cdr` of the list whose `car` is the last element of the first list. To reiterate, `cons` and `append` are not the same function. To construct a proper list, `cons` accepts an atom and a list. To do the same, `append` accepts a list and a list.

While the running time of `append` is not constant like that of `cons`, it is also not polynomial [e.g., $O(n^2)$]. However, the effect of the less efficient `append` function is compounded in functions that use `append` where the use of `cons` would otherwise suffice. For instance, consider the following `reverse`¹⁰ function, which accepts a list and returns the list reversed:

```

(define reversel
  (lambda (l)
    (cond
      ((null? l) '())
      (else (append (reversel (cdr l)) (cons (car l) '())))))

```

Using the strategy discussed previously for developing recursive solutions to problems, we know that the reverse of the empty list is the empty list. To extend the reverse of a list of $n - 1$ items to that of n items, we append the remaining item *as a list* to the reversed list of $n - 1$ items. For instance, if we want to reverse a list `(a b c)`, we assume we have the reversed `cdr` of the original list [i.e., the list `(c b)`] and we append the `car` of the original list *as a list* [i.e., `(a)`] to that list [i.e., resulting in `(c b a)`]. The following example illustrates how, in reversing the list `(a b c)`, the expression in the `else` clause is expanded (albeit implicitly on the run-time stack):

```

1  (append (reversel '(b c)) (cons a '()))
2  (append (reversel '(b c)) '(a))
3  (append (append (reversel '(c)) (cons b '())) '(a))
4  (append (append (reversel '(c)) '(b)) '(a))
5  ;; base case
6  (append (append (append (reversel '()) (cons c '())) '(b)) '(a))
7  (append (append (append '() '(c)) '(b)) '(a))

```

10. The function `reverse` is built into Scheme. The version we define is named `reversel`.

```

8  (append (append '(c) '(b)) '(a))
9  (append '(c b) '(a))
10 (append '(c b a))

```

Notice that rotating this expansion 90 degrees left forms a parabola showing how the run-time stack grows until it reaches the base case of the recursion (line 6) and then shrinks. This is called *recursive-control behavior* and is discussed in more detail in Chapter 13.

As this expansion illustrates, reversing a list of n items requires $n - 1$ calls to `append`. Recall that the running time of `append` is linear, $O(n)$. Therefore, the run-time complexity of this definition of `reverse1` is $O(n^2)$, which is unsettling. Intuitively, to reverse a list, we need pass through it only once; thus, the upper bound on the running time should be no worse than $O(n)$. The difference in running time between `cons` and `append` is magnified when `append` is employed in a function like `reverse1`, where `cons` would suffice. This suggests that we should never use `append` where `cons` will suffice (see *Design Guideline 3: Efficient List Construction*). We rewrite `reverse1` using only `cons` and no `appends` in a later example. Before doing so, however, we make some instructional observations on this initial version of the `reverse1` function.

- The expression `(cons (car l) '())` in the previous definition of `append` can be replaced by `(list (car l))` without altering the semantics of the function:

```

(define reverse1
  (lambda (l)
    (cond
      ((null? l) '())
      (else (append (reverse1 (cdr l)) (list (car l)))))))

```

The `list` function accepts an arbitrary number of arguments and creates a list of those arguments. The `list` function is not the same as the `append` function:

```

> (list 'a 'b 'c)
(a b c)
> (append 'a 'b 'c)
ERROR
> (list '(a) '(b) '(c))
((a) (b) (c))
> (append '(a) '(b) '(c))
(a b c)

```

The function `append` accepts only arguments that are proper lists. In contrast, the function `list` accepts any values as arguments (atoms or lists). The `list` function is not to be confused with the built-in Scheme predicate `list?`, which returns true if its argument is a proper list and false otherwise:

```

> (list? '(a b c))
#t
> (list? '(a (b c)))
#t

```



```

> (list? 'a)
#f
> (list? 3)
#f
> (list? '(a . b))
#f

```

Furthermore, the `list?` predicate is not to be confused with the `pair?` predicate, which returns true if its argument is a `cons` cell, even if not a proper list, and false otherwise:

```

> (list? '(a . b))
#f
> (pair? '(a . b))
#t
> (pair? '(a b c))
#t

```

- Scheme uses the *pass-by-value* parameter-passing mechanism (sometimes called *pass-by-copy*). This is the same parameter-passing mechanism used in C, with which readers may be more familiar. The following session illustrates the use of pass-by-value in Scheme:

```

> (define a 'a)
> a
a
> (define bc '(b c))
> bc
(b c)
> (define abc (cons a bc))
> abc
(a b c)
> (define bc '(d e))
> bc
(d e)
> abc
(a b c)

```

A consequence of pass-by-value semantics for the `reverse!` function is that after the function returns, the original list remains unchanged; in other words, it has the same order it had before the function was called. Parameter-passing mechanisms are discussed in detail in Chapter 12.

- A consequence of the typeless nature of Lisp is that most functions are *polymorphic*, without explicit *operator overloading*. Therefore, not only can the `reverse!` function reverse a list of numbers or strings, but it can also reverse a list of employee records or pixels, or reverse a list involving a combination of all four types. It can even reverse a list of lists.

5.6.3 The Difference Lists Technique

If we examine the pattern of recursion used in the definition of our `reverse!` function, we notice that the function mirrors both the recursive specification of the problem and the recursive definition of a reversed list. We were able to follow

our guidelines for developing recursive algorithms in defining it. Improving the run-time complexity of `reverse1` involves obviating the use of `append` through a method called the *difference lists technique* (see *Design Guideline 7: Difference Lists Technique*). (We revisit the difference lists technique in Section 13.7, where we introduce the concept of *tail recursion*.) Using the difference lists technique compromises the natural correspondence between the recursive specification of a problem and the recursive solution to it. Compromising this correspondence and, typically, the readability of the function, which follows from this break in symmetry, for the purposes of efficiency of execution is a theme that recurs throughout this text. We address this trade-off in more detail in Chapter 13, where a reasonable solution to the problem is presented.

In the absence of side effects, which are contrary to the spirit of functional programming, the only ways for successive calls to a recursive function to share and communicate data is through return values (as is the case in the `reverse1` function) or parameters. The difference lists technique involves using an additional parameter that represents the solution (e.g., the reversed list) computed thus far. A solution to the problem of reversing a list using the difference lists technique is presented here:

```

1  (define reverse1
2    (lambda (l)
3      (cond
4        ((null? l) '())
5        (else (rev l '())))))
6
7  (define rev
8    (lambda (l rl)
9      (cond
10       ((null? l) rl)
11       (else (rev (cdr l) (cons (car l) rl))))))

```

Notice that this solution involves the use of a helper function `rev`, which ensures that the signature of the original function `reverse1` remains unchanged. The additional parameter is `rl`, which stands for reversed list. When `rev` is first called on line 5, the reversed list is empty. On line 11, we grow that reversed list by consing each element of the original list into `rl` until the original list `l` is empty (i.e., the base case on line 10), at which point we simply return `rl` because it is the completely reversed list at that point. Thus, the reversed list is built as the original list is traversed. Notice that `append` is no longer used.

Conducting a similar run-time analysis of this version of `reverse1` as we did with the prior version, we see:

```

(reverse1 '(a b c))
(rev '(a b c) '())
(rev '(b c) (cons (car '(a b c)) '()))
(rev '(b c) (cons 'a '()))
(rev '(b c) '(a))
(rev '(c) (cons (car '(b c)) '(a)))
(rev '(c) (cons 'b '(a)))
(rev '(c) '(b a))
(rev '() (cons (car '(c)) '(b a)))

```

```
(rev '()) (cons 'c '(b a)))
;; base case
(rev '()) '(c b a))
(c b a)
```

Now the running time of the function is linear [i.e., $O(n)$] in the size of the list to be reversed. Notice also that, unlike in the original function, when the expansion is rotated 90 degrees left, a rectangle is formed, rather than a parabola. Thus, the improved version of `reverse1` is more efficient not only in time, but also in space. An unbounded amount of memory (i.e., stack) is required for the first version of `reverse1`. Specifically, we require as many frames on the run-time stack as there are elements in the list to be reversed. Unbounded memory is required for the first version because each function call in the first version must wait (on the stack) for the recursive call it invokes to return so that it can complete the computation by appending `(cons (car l) '())` to the intermediate result that is returned:

```
;; append is waiting for reverse1 to return
;; so it can complete the computation
(append (reverse1 (cdr l)) (cons (car l) '()))
```

The same is not true for the second version. The second version only requires a constant memory size because no pending computations are waiting for the recursive call to return:

```
;; no computations are waiting for rev to return
(else (rev (cdr l) (cons (car l) rl)))
```

Formally, this is because the recursive call to `rev` is in *tail position* or is a *tail call*, and the difference lists version of `reverse1` is said to use *tail recursion* (Section 13.7).

While working through these examples in the Racket interpreter, notice that the functions can be easily tested in isolation (i.e., independently of the rest of the program) with the read-eval-print loop. For instance, we can test `rev` independently of `reverse1`. This fosters a convenient environment for debugging, and facilitates a process known as interactive or incremental testing. Compiled languages, such as C, in contrast, require test drivers in `main` (which clutter the program) to achieve the same.

Programming Exercises for Section 5.6

Exercise 5.6.1 Define a Scheme function `member1?` that accepts only an atom `a` and a list of atoms `lat`, in that order, and returns `#t` if the atom is an element of the list and `#f` otherwise.

Exercise 5.6.2 Define a Scheme function `remove` that accepts only a list and an integer `i` as arguments and returns another list that is the same as the input list, but with the `i`th element of the input list removed. If the length of the input list is

less than i , return the same list. Assume that $i = 1$ refers to the first element of the list.

Examples:

```
> (remove 1 '(9 10 11 12))
'(10 11 12)
> (remove 2 '(9 10 11 12))
'(9 11 12)
> (remove 3 '(9 10 11 12))
'(9 10 12)
> (remove 4 '(9 10 11 12))
'(9 10 11)
> (remove 5 '(9 10 11 12))
'(9 10 11 12)
```

Exercise 5.6.3 Define a Scheme function called `makeset` that accepts only a list of integers as input and returns the list with any repeating elements removed. The order in which the elements appear in the returned list does not matter, as long as there are no duplicate elements. Do not use any user-defined auxiliary functions, except the built-in Scheme `member` function.

Examples:

```
> (makeset '(1 3 4 1 3 9))
'(4 1 3 9)
> (makeset '(1 3 4 9))
'(1 3 4 9)
> (makeset '("apple" "orange" "apple"))
'("orange" "apple")
```

Exercise 5.6.4 Define a Scheme function `cycle` that accepts only a list and an integer i as arguments and cycles the list i times. Do not use any user-defined auxiliary functions and do not use the difference lists technique (i.e., you may use `append`).

Examples:

```
> (cycle 0 '(1 4 5 2))
'(1 4 5 2)
> (cycle 1 '(1 4 5 2))
'(4 5 2 1)
> (cycle 2 '(1 4 5 2))
'(5 2 1 4)
> (cycle 4 '(1 4 5 2))
'(1 4 5 2)
> (cycle 6 '(1 4 5 2))
'(5 2 1 4)
> (cycle 10 '(1))
'(1)
> (cycle 9 '(1 4))
'(4 1)
```

Exercise 5.6.5 Redefine the Scheme function `cycle` from Programming Exercise 5.6.4 using the *difference lists technique*. You may use `append`, but only in a base so that it is only ever applied once.

Exercise 5.6.6 Define a Scheme function `transpose` that accepts a list of atoms as its only argument and returns that list with adjacent elements transposed. Specifically, `transpose` accepts an input list of the form $(e_1 e_2 e_3 e_4 e_5 e_6 \cdots e_{n-1} e_n)$ and returns a list of the form $(e_2 e_1 e_4 e_3 e_6 e_5 \cdots e_n e_{n-1})$ as output. If n is odd, e_n will continue to be the last element of the list. Do not use any user-defined auxiliary functions and do not use `append`.

Examples:

```
> (transpose '())
()
> (transpose '(a))
(a)
> (transpose '(a b))
(b a)
> (transpose '(a b c d))
(b a d c)
> (transpose '(a b c d e))
(b a d c e)
```

Exercise 5.6.7 Define a Scheme function `oddevensum` that accepts only a list of integers as an argument and returns a pair consisting of the sum of the odd and even positions of the list. Do not use any user-defined auxiliary functions.

Examples:

```
> (oddevensum '())
'(0 . 0)
> (oddevensum '(6))
'(6 . 0)
> (oddevensum '(6 3))
'(6 . 3)
> (oddevensum '(6 3 8))
'(14 . 3)
> (oddevensum '(1 2 3 4))
'(4 . 6)
> (oddevensum '(1 2 3 4 5 6))
'(9 . 12)
> (oddevensum '(1 2 3))
'(4 . 2)
```

Exercise 5.6.8 Define a Scheme function `intersect` that returns the set intersection of two sets represented as lists. Do not use any built-in Scheme functions or syntactic forms other than `cons`, `car`, `cdr`, `or`, `null?`, and `member`.

Examples:

```
> (intersect '() '())
()
> (intersect '(a b) '())
()
> (intersect '() '(a b))
()
> (intersect '(a) '(a))
(a)
```

```

(a)
> (intersect '(a b) '(a b))
(a b)
> (intersect '(a b) '(c d))
()
> (intersect '(a b c) '(e d c))
(c)
> (intersect '(a b c) '(b d c))
(b c)
> (intersect '(a c b d e f) '(c e d))
(c d e)
> (intersect '(a b c d e f) '(a b c d e f))
(a b c d e f)

```

Exercise 5.6.9 Consider the following description of a function `mystery`. This function accepts a non-empty list of numbers in which no number is greater than its own index (first element is at index 1), and returns a list of numbers of the same length. Each number in the argument is treated as a backward index starting from its own position to a point earlier in the list of numbers. The result at each position is found by counting backward from the current position according to the index.

Examples:

```

> (mystery '(1 1 1 3 4 2 1 1 9 2))
(1 1 1 1 1 4 1 1 1 9)
> (mystery '(1 2 3 4 5 6 7 8 9))
(1 1 1 1 1 1 1 1 1)
> (mystery '(1 2 3 1 2 3 4 1 8 2 10))
(1 1 1 1 1 1 1 1 2 8 2)

```

Define the `mystery` function in Scheme.

Exercise 5.6.10 Define a Scheme function `reverse*` that accepts only an *S-list* as an argument and returns not only that S-list reversed, but also all sublists of that S-list reversed as well, and sublists of sublists, reversed, and so on.

Examples:

```

> (reverse* '())
()
> (reverse* '(((Nothing))) ((will) (()) ()))
((come ()) (of nothing)))
'(((nothing of) (()) come) ((()) (will)) (((Nothing))))
> (reverse* '(((1 2 3) (4 5)) ((6)) (7 8) (9 10)
              ((11 12 (13 14 (15 16))))))
'((((16 15) 14 13) 12 11)) (10 9) (8 7) ((6)) ((5 4) (3 2 1)))

```

5.7 Constructing Additional Data Structures

Sophisticated, dynamic memory data structures, such as trees, are built from lists, which are just `cons` cells.

5.7.1 A Binary Tree Abstraction

Consider the following BNF specification of a binary tree:

```

<bintree> ::= number
<bintree> ::= (<symbol> <bintree> <bintree>)

```

The following sentences in the language defined by this grammar represent binary trees:

```

111
32
(opus 111 32)
(sonata 1820 (opus 111 32))
(Beethoven (sonata 32 (opus 110 31)) (sonata 33 (opus 111 32)))

```

The following function accepts a binary tree as an argument and returns the number of internal and leaf nodes in the tree:

```

1 (define bintree-size
2   (lambda (s)
3     (cond
4       ((number? s) 1)
5       (else (+ (bintree-size (car (cdr s)))
6                (bintree-size (car (cdr (cdr s))))
7                1)))) ; count self

```

In this function, and in others we have seen in this chapter, we do not include provisions for handling errors (e.g., passing a string to the function). “Programs such as this that fail to check that their input is properly formed are *fragile*. (Users think a program is broken if it behaves badly, even when it is being used improperly.) It is generally better to write *robust* programs that thoroughly check their arguments, but robust programs are often much more complicated” (Friedman, Wand, and Haynes 2001, p. 16). Therefore, to focus on the particular concept at hand, we try as much as possible to shield the reader’s attention from all details superfluous to that concept and present fragile programs for ease and simplicity.

Note also that line 6 contains two consecutive `cdrs` followed by a `car`. Often when manipulating data structures represented as lists, we want to access a particular element of a list. This typically involves calling `car` and `cdr` in a variety of orders. Scheme provides syntactic sugar through some built-in functions to help the programmer avoid these long-winded series of calls to `car` and `cdr`. Specifically, the programmer can call `cxr`, where `x` represents a string of up to four `as` or `ds`. Table 5.1 presents some examples. Thus, we can rewrite `bintree-size` as follows:

```

(define bintree-size
  (lambda (s)
    (cond
      ((number? s) 1)
      (else (+ (bintree-size (cadr s))
               (bintree-size (caddr s))
               1))))

```

<code>(car (cdr (cdr (cdr ' (a b c d e f)))))</code>	<code>= (caddr ' (a b c d e f))</code>	<code>= d</code>
<code>(car (car (car ' (((a b)))))</code>	<code>= (caaar ' (((a b)))))</code>	<code>= a</code>
<code>(car (cdr (car (cdr ' (a (b c) d e)))))</code>	<code>= (cadadr ' (a (b c) d e))</code>	<code>= c</code>
<code>(cdr (car (cdr (car ' ((a (b c d)) e f)))))</code>	<code>= (cdadar ' ((a (b c d)) e f))</code>	<code>= (c d)</code>

Table 5.1 Examples of Shortening `car-cdr` Call Chains with Syntactic Sugar

Moreover, with a similar pattern of recursion, and the help of these abbreviated call chains, we can define a variety of binary tree traversals:

```
(define preorder
  (lambda (bintree)
    (cond
      ((number? bintree) (cons bintree '()))
      (else
       (cons (car bintree)
              (append (preorder (cadr bintree))
                      (preorder (caddr bintree)))))))

;;; if inorder returns a sorted list,
;;; then its parameter is a binary search tree
(define inorder
  (lambda (bintree)
    (cond
      ((number? bintree) (cons bintree '()))
      (else (append (inorder (cadr bintree))
                    (cons (car bintree) (inorder (caddr bintree)))))))
```

Using the definitions of the following three functions, we can make the definitions of the traversals more readable (see the definition of `preorder` on lines 13–19):

```
1 (define root
2   (lambda (bintree)
3     (car bintree)))
4
5 (define left
6   (lambda (bintree)
7     (cadr bintree)))
8
9 (define right
10  (lambda (bintree)
11    (caddr bintree)))
12
13 (define preorder
14   (lambda (bintree)
15     (cond
16       ((number? bintree) (cons bintree '()))
17       (else (cons (root bintree)
18                   (append (preorder (left bintree))
                           (preorder (right bintree)))))))
19
```

5.7.2 A Binary Search Tree Abstraction

As a final example of the use of `cons` cells as primitives in the construction of a data structure, consider the following BNF definition of a binary search tree:


```

<bst> ::= empty
<bst> ::= <key> <bst> <bst>

```

This *context-free* grammar does not define the semantic property of a binary search tree (i.e., that the nodes are arranged in an order rendering the tree amenable to an efficient search), which is an example of *context*.

Programming Exercises for Section 5.7

Exercise 5.7.1 Define `postorder` traversal in Scheme.

Exercise 5.7.2 (Friedman, Wand, and Haynes 2001, Exercise 1.17.1, p. 27) Consider the following BNF specification of a binary search tree.

```

<binsearchtree> ::= ()
<binsearchtree> ::= (<integer> <binsearchtree> <binsearchtree>)

```

Define a Scheme function `path` that accepts only an integer `n` and a list `bst` representing a binary search tree, in that order, and returns a list of `lefts` and `rights` indicating how to locate the vertex containing `n`. You may assume that the integer is always found in the tree.

Examples:

```

> (path 31 '(31 (15 () ()) (42 () ())))
'()
> (path 42 '(52 (24 (14 (8 (2 () ()) ()) (17 () ()))
                (32 (26 () ()) (42 () (51 () ())))
                (78 (61 () ()) (101 () ())))))
'(left right right)

```

Exercise 5.7.3 Complete Programming Exercise 5.7.2, but this time do not assume that the integer is always found in the tree. If the integer is not found, return the atom `'notfound`.

Examples:

```

> (path 17 '(14 (7 () (12 () ()))
               (26 (20 (17 () ()) (31 () ())))))
'(right left left)
> (path 32 '(14 (7 () (12 () ()))
               (26 (20 (17 () ())
                     (31 () ())))))
'notfound
> (path 17 '(17 () ()))
'()
> (path 17 '(18 () ()))
'notfound
> (path 2 '(31 (15 () ()) (42 () ())))
'notfound

```

```
> (path 17 '(52 (24 (14 (8 (2 () ()) ()) (17 () ())))
      (32 (26 () ()) (42 () (51 () ())))
      (78 (61 () ()) (101 () ())))
'(left left right)
```

Exercise 5.7.4 Complete Programming Exercise 5.7.3, but this time do not assume that the binary tree is a binary search tree.

Examples:

```
> (path 26 '(52 (24 (14 (8 (2 () ()) ()) (17 () ())))
      (32 (26 () ()) (42 () (51 () ())))
      (78 (61 () ()) (101 () ())))
'(left right left)
> (path 'Morisot
      '(Monet
        (Matisse
         (Degas (Manet (Renoir () ()) ()) (vanGogh () ()))
         (Cezanne
          (Pissarro () ())
          (Morisot () (Picasso () ())))
         (Rembrandt (Sisley () ()) (Bazille () ())))
      '(left right right)
```

5.8 Scheme Predicates as Recursive-Descent Parsers

Recall from Chapter 3 that the hallmark of a recursive-descent parser is that the program code implementing it naturally reflects the grammar. That is, there is a one-to-one correspondence between each non-terminal in the grammar and each function in the parser, where each function is responsible for recognizing a subsentence in the language starting from that non-terminal. Often Scheme predicates can be viewed in the same way.

5.8.1 `atom?`, `list-of-atoms?`, and `list-of-numbers?`

Consider the following predicate for determining whether an argument is an atom (Friedman and Felleisen 1996a, Preface, p. xii):

```
(define atom?
  (lambda (x)
    (and (not (pair? x)) (not (null? x)))))
```

We can extend this idea by trying to recognize a list of atoms—in other words, by trying to determine whether a list is composed only of atoms:

```
<list-of-atoms> ::= ()
<list-of-atoms> ::= (<atom> . <list-of-atoms>)
```

Notice that we use right-recursion in defining this language because left-recursion throws a recursive-descent parser into an infinite loop:

```
(define list-of-atoms?
  (lambda (lst)
    (or (null? lst)
        (and (pair? lst)
              (atom? (car lst))
              (list-of-atoms? (cdr lst))))))
```

Notice also that the definition of this function is a reflection of the two production rules given previously. The pattern used to recognize the list of atoms can be manually reused to recognize a list of numbers:

```
(define list-of-numbers?
  (lambda (lst)
    (or (null? lst)
        (and (pair? lst)
              (number? (car lst))
              (list-of-numbers? (cdr lst))))))
```

Notice that this is nearly a complete repeat of the `list-of-atoms?` function. Next, we see how to eliminate such redundancy in a functional program.

5.8.2 Factoring out the `list-of` Pattern

Since functions are first-class entities in Scheme, we can define a function that accepts a function as an argument. Thus, we can factor out the `number?` predicate used in the definition of the `list-of-numbers?` function so it can be passed in as an argument. Abstracting away the predicate as an additional argument generalizes the `list-of-numbers?` function. In other words, it now becomes a `list-of` function that accepts a predicate and a list as arguments and calls the predicate on the elements of the list to determine whether all of the items in the list are of some particular type:

```
(define list-of
  (lambda (predicate lst)
    (or (null? lst)
        (and (pair? lst)
              (predicate (car lst))
              (list-of predicate (cdr lst))))))
```

In this way, the `list-of` function abstracts the details of the predicate from the pattern of recursion used in the original definition of `list-of-numbers?`:

```
> (list-of atom? '(a b c d))
#t
> (list-of atom? '(1 2 3 4))
#t
> (list-of atom? '((a b) c d))
#f
> (list-of atom? 'abcd)
#f
```

```

> (list-of number? '(1 2 3 4))
#t
> (list-of number? '(a b c d))
#f
> (list-of number? '((1 2) 3 4))
#f

```

Recall that the first-class nature of functions also supports the definition of a function that returns a function as a value. Thus, we can refine the `list-of` function further by also abstracting away the list to be parsed, which further generalizes the pattern of recursion. Specifically, we can redefine the `list-of` function to accept a predicate as its only argument and to return a predicate that calls this input predicate on the elements of a list to determine whether all elements are of the given type (Friedman, Wand, and Haynes 2001, p. 45):

```

(define list-of
  (lambda (predicate)
    (lambda (lst)
      (or (null? lst)
          (and (pair? lst)
                (predicate (car lst))
                ((list-of predicate) (cdr lst)))))))

```

This revised `list-of` function returns a specific type of anonymous function called a *closure*—a function that *remembers* the lexical environment in which was created even after the function which in that environment is defined is popped off the stack. (We discuss closures in more detail in Chapter 6.) Incidentally, the language concept called *function currying* supports the automatic conception of the last definition of the `list-of` function from the penultimate definition of it. (We study function currying in Chapter 8.) Our revised `list-of` function—which accepts a function and returns a function—is now a powerful construct for generating a variety of helpful functions:

```

(define list-of-atoms? (list-of atom?))
(define list-of-symbols? (list-of symbol?))
(define list-of-numbers? (list-of number?))
(define list-of-strings? (list-of string?))
(define list-of-pairs? (list-of pair?))

```

Functions that either accept a function as an argument or return a function as a return value, or both, are called *higher-order functions* (HOFs). Higher-order functions encapsulate common, reusable patterns of recursion in a function. Higher-order and anonymous functions are often used in concert, such that the higher-order function either receives an anonymous function as an argument or returns one as a return value, or both. Higher-order functions, as we see throughout this text, especially in Chapter 8, are building blocks that can be creatively composed and combined, like LEGO® bricks, at a programmer’s discretion to construct powerful and reusable programming abstractions. Mastering the use of higher-order functions moves the imperative or object-oriented programmer closer to fully embracing the spirit and unleashing the power of functional programming. For instance, we used the higher-order function

`list-of` to create the `list-of-atoms?` and `list-of-numbers?` functions. Such functions also empower the programmer to define multiple functions that encapsulate the same pattern of recursion without repeating code. Indeed, it has been suggested that “one line of Lisp can replace 20 lines of C” (Muehlbauer 2002).

Using a language with support for functional programming to simply define a series of recursive functions is imperative programming without side effects (see the first layer of functional/Lisp programming in Figure 5.10 later in this chapter). Thus, it neither makes full use of the abstraction mechanisms of functional programming nor fully leverages the power resulting from their use. We need to cultivate the skill of programming with higher-order abstractions if we are to unleash the power of functional programming.

Programming Exercise for Section 5.8

Exercise 5.8.1 Complete Programming Exercise 3.4 (part a only) in Scheme using the grammar from Programming Exercise 3.5. Name your top-level function `parse` and invoke it as shown below.

Examples:

```
> (parse " 2 + 3")
" 2 + 3" is an expression.

> (parse "-45 + -45")
"-45 + -45" is an expression.

> (parse " -45 + -45+ --452 +2*3 ")
" -45 + -45+ --452 +2*3 " is an expression.

> (parse " -45 + -45+ --452 +2*a")
" -45 + -45+ --452 +2*a" contains lexical units
which are not lexemes and, thus, is not an expression.
```

Hint: Investigate the following built-in Scheme functions as they apply to this problem: `char-numeric?`, `display`, `integer?`, `list->string`, `string`, `string-append`, `string-length`, `string->list`, `string->number`, and `string->symbol`.

5.9 Local Binding: `let`, `let*`, and `letrec`

5.9.1 The `let` and `let*` Expressions

Local binding is introduced in a Scheme program through the `let` construct:

```
> (let ((a 1) (b 2))
>   (+ a b))
3
```

The semantics of a `let` expression are as follows. Bindings are created in the list of lists immediately following `let` [e.g., `((a 1) (b 2))`] and are only bound during the evaluation of the second S-expression [e.g., `(+ a b)`]. Use of `let` does not violate the spirit of functional programming for two reasons: (1) `let` creates bindings, not assignments, and (2) `let` is syntactic sugar used to improve the readability of a program; any `let` expression can be rewritten as an equivalent `lambda` expression. To make the leap from a `let` expression to a `lambda` expression, we must recognize that functional application is the only mechanism through which to create a binding in λ -calculus; that is, the argument to the function is bound to the formal parameter. Moreover, once an identifier is bound to a value, it cannot be rebound to a different value within the same scope:

```
> ((lambda (a b) (+ a b)) 1 2)
3
```

Thus, when the function `(lambda (a b) (+ a b))` is called with the arguments 1 and 2, `a` and `b` are bound to 1 and 2, respectively. The bindings in a `let` expression [e.g., `((a 1) (b 2))`] are evaluated in *parallel*, not in sequence. Thus, the evaluation of the following expression results in an error:

```
> (let ((a 1) (b (+ a 1)))
>      (+ a b))
ERROR: a not bound in the expression (b (+ a 1))
```

We can produce *sequential* evaluation of the bindings by nesting lets:

```
> (let ((a 1))
>      (let ((b (+ a 1)))
>          (+ a b)))
3
```

Scheme provides syntactic sugar for this style of nesting with a `let*` expression, in which bindings are evaluated in sequence (Table 5.2):

```
> (let* ((a 1) (b (+ a 1)))
>        (+ a b))
3
```

Thus, just as `let` is syntactic sugar for `lambda`, `let*` is syntactic sugar for `let`. Therefore, any `let*` expression can be reduced to a `lambda` expression as well:

```
> ((lambda (a)
>      ((lambda (b) (+ a b)) (+ a 1)))
>      1)
3
```

<code>let</code>	bindings are added to the environment in <i>parallel</i> .
<code>let*</code>	bindings are added to the environment in <i>sequence</i> .

Table 5.2 Binding Approaches Used in `let` and `let*` Expressions

Never use `let*` when there are no dependencies in the list of bindings [e.g., `((a 1) (b 2) (c 3))`].

5.9.2 The `letrec` Expression

Since the bindings specified in the first list of a `let` expression are not placed in the environment until the evaluation of the second list begins, recursion is a challenge. For instance, consider the following `let` expression:

```
1 > (let ((length1 (lambda (l)
2 >   (cond
3 >     ((null? l) 0)
4 >     (else (+ 1 (length1 (cdr l)))))))
5 >   (length1 '(a b c d)))
6 ERROR
```

Evaluation of this expression results in an error because `length1` is not yet bound on line 4—it is not bound until line 5. Notice the issue here is not one of parallel vis-à-vis sequential bindings since there is only one binding (i.e., `length1`). Rather, the issue is that a binding cannot refer to itself until it is bound. Scheme has the `letrec` expression to make bindings visible *while* they are being created:

```
> (letrec ((length1 (lambda (l)
>   (cond
>     ((null? l) 0)
>     (else (+ 1 (length1 (cdr l)))))))
>   (length1 '(a b c d)))
4
```

5.9.3 Using `let` and `letrec` to Define a Local Function

Armed with `letrec`, we can consolidate our example `reverse1` and `rev` functions to ensure that only `reverse1` can invoke `rev`. In other words, we want to restrict the *scope* of `rev` to the block of code containing the `reverse1` function (*Design Guideline 5: Nest Local Functions*):

```
(define reverse1
  (letrec ((rev
    (lambda (lst rl)
      (cond
        ((null? lst) rl)
        (else (rev (cdr lst) (cons (car lst) rl))))))
    (lambda (l)
      (cond
        ((null? l) '())
        (else (rev l '())))))
```

Just as `let*` is syntactic sugar for `let`, `letrec` is also syntactic sugar for `let` (and, therefore, both are syntactic sugar for `lambda` through `let`). In demonstrating how a `letrec` expression can be reduced to a `lambda` expression, we witness the power of first-class functions and λ -calculus supporting the use of mathematical techniques such as recursion, even in a language with no native

support for recursion. We start by reducing the preceding `letrec` expression for `length1` to a `let` expression. Functions only know about what is passed to them, and what is in their local environment. Here, we need the `length1` function to know about itself—so it can call itself recursively. Thus, we pass `length1` to `length1` itself!

```
> (let ((length1 (lambda (fun_length) l)
>      (cond
>        ((null? l) 0)
>        (else (+ 1 (fun_length fun_length (cdr l)))))))
>      (length1 length1 '(a b c d)))
4
```

Reducing this `let` expression to a `lambda` expression involves the same idea and technique used in Section 5.9.1—bind a function to an identifier `length1` by passing a literal function to another function that accepts `length1` as a parameter:

```
> ((lambda (length1) (length1 length1 '(a b c d)))
>   (lambda (fun_length) l)
>   (cond
>     ((null? l) 0)
>     (else (+ 1 (fun_length fun_length (cdr l))))))
4
```

From here, we simply need to make one more transformation to the code so that it conforms to λ -calculus, where only unary functions can be defined:

```
> ((lambda (length1) ((length1 length1) '(a b c d)))
>   (lambda (fun_length)
>     (lambda (l)
>       (cond
>        ((null? l) 0)
>        (else (+ (car l)
>                  ((fun_length fun_length) (cdr l)))))))
4
```

We have just demonstrated how to define a recursive function from first principles (i.e., assuming the programming language being used to define the function does not support recursion). The pattern used to define the `length1` function recursively is integrated (i.e., tightly woven) into the `length1` function itself. If we want to implement additional functions recursively (e.g., `reverse1`), without using the `define` syntactic form (i.e., the built-in support for recursion in Scheme), we would have to embed the pattern of code used in the definition of the function `length1` into the definitions of any other functions we desire to define recursively. Just as with the `list-of-atoms?` function, it is helpful to abstract the approach to recursion presented previously from the actual function we desire to define recursively. This is done with a λ -expression called the (*normal-order*) *Y combinator*, which expresses the essence of recursion in a non-recursive way in the λ -calculus:

$$\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

The *Y combinator* expression in the λ -calculus was invented by Haskell Curry. Some have hypothesized a connection between the *Y combinator* and the double

helix structure in human DNA, which consists of two copies of the same strand adjacent to each other and is the key to the self-replication of DNA. Similarly, the structure of the Y combinator λ -expression consists of two copies of the same subexpression [i.e., $(\lambda x.f(x\ x))$] adjacent to each other and is the key to recursion—a kind of self-replication—in the λ -calculus or a programming language. Programming Exercise 6.10.15 explores the Y combinator.

These transformations demonstrate that Scheme is an attractive language through which to explore and implement concepts of programming languages. We continue to use Scheme in this capacity in this text. For instance, we explore binding, and implement *lazy evaluation*—an alternative parameter-passing mechanism—and a variety of control abstractions, including *coroutines*, in Scheme in Chapters 6, 12, and 13, respectively.

Since `lambda` is primitive, any `let`, `let*`, and `letrec` expression can be reduced to a `lambda` expression (Figure 5.9). Thus, λ -calculus is sufficient to create programming abstractions.

Again, the grammar rules for λ -calculus, given in Section 5.2.2, have no provision for defining a function accepting more than one argument. However, here, we have defined multiple functions accepting more than one argument. Any function accepting more than one argument can be rewritten as an expression in λ -calculus by nesting λ -expressions. For instance, the function definition and invocation

```
> (lambda (a b)
    (+ a b))
#<procedure>

> ((lambda (a b)
    (+ a b)) 1 2)
3
```

can be rewritten as follows:

```
> (lambda (a)
    (lambda (b)
      (+ a b)))
#<procedure>

> ((lambda (a)
    ((lambda (b)
      (+ a b)) 2)) 1)
3
```

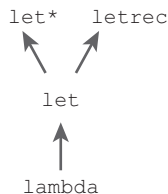


Figure 5.9 Graphical depiction of the foundational nature of `lambda`.

General Pattern	Instance of Pattern
<code>(let ((sym1 val1) (sym2 val2) ... (symn valn)) body)</code>	<code>(let ((a 1) (b 2)) (+ a b))</code>
<code>((lambda (sym1 sym2 ... symn) body) val1 val2 ... valn)</code>	<code>((lambda (a b) (+ a b)) 1 2)</code>
<code>(let ((sym1 val1)) (let ((sym2 val2)) ... (let ((symn valn)) body)))</code>	<code>(let ((a 1) (let ((b 2)) (+ a b))))</code>
<code>((lambda (sym1) ((lambda (sym2) ((lambda (...) ((lambda (symn) body) valn)) ...)) val2)) val1)</code>	<code>((lambda (a) ((lambda (b) (+ a b)) 2)) 1)</code>

Table 5.3 Reducing let to lambda (All rows of each column are semantically equivalent.)

General Pattern	Instance of Pattern
<code>(let* ((sym1 val1) (sym2 val2) ... (symn valn)) body)</code>	<code>(let* ((a 1) (b (+ a 1))) (+ a b))</code>
<code>(let ((sym1 val1)) (let ((sym2 val2)) ... (let ((symn valn)) body)))</code>	<code>(let ((a 1)) (let (b (+ a 1)) (+ a b)))</code>
<code>((lambda (sym1) ((lambda (sym2) ((lambda (...) ((lambda (symn) body) valn)) ...)) val2)) val1)</code>	<code>((lambda (a) ((lambda (b) (+ a b)) (+ a 1))) 1)</code>

Table 5.4 Reducing let* to lambda (All rows of each column are semantically equivalent.)

Tables 5.3, 5.4, and 5.5 summarize the reductions from let, let*, and letrec, respectively, into λ -calculus. Table 5.6 provides a summary of all three syntactic forms.

5.9.4 Other Languages Supporting Functional Programming: ML and Haskell

With an understanding of both λ -calculus—the foundation and theoretical basis of functional programming—and the building blocks of functional programs (e.g., functions and cons cells), learning new languages supporting function programming is a matter of orienting oneself to a new syntax. ML and Haskell are languages supporting functional programming that we use in this text, especially in our discussion of concepts related to types and data abstraction in Part II, particularly for the ease and efficacy with which concepts related to types can

General Pattern	Instance of Pattern
<pre>(let ((f (lambda (sym1 sym2 ... symn) ... (f val1 val2 ... valn) ...))) (f val1 val2 ... valn))</pre>	<pre>(let ((length1 (lambda (l) (cond ((null? l) 0) (else (+ 1 (length1 (cdr l))))))) (length1 '(a b c d)))</pre>
<pre>(letrec ((f (lambda (sym1 sym2 ... symn) ... (f val11 val12 ... valnm) ...))) (f val1 val2 ... valn))</pre>	<pre>(letrec ((length1 (lambda (l) (cond ((null? l) 0) (else (+ 1 (length1 (cdr l))))))) (length1 '(a b c d)))</pre>
<pre>(let ((f (lambda (copy_of_f copy_of_f sym1 sym2 ... symn) ... (copy_of_f copy_of_f val1 val2 ... valn) ...))) (f val1 val2 ... valn))</pre>	<pre>(let ((length1 (lambda (copy_of_length copy_of_length l) (cond ((null? l) 0) (else (+ 1 (copy_of_f copy_of_f (cdr l))))))) (length1 length1 '(a b c d)))</pre>
<pre>(let ((f (lambda (copy_of_f copy_of_f sym1 sym2 ... symn) ... (copy_of_f copy_of_f val1 val2 ... valn) ...))) (f val1 val2 ... valn))</pre>	<pre>(let ((length1 (lambda (copy_of_f copy_of_f l) (cond ((null? l) 0) (else (+ 1 (copy_of_f copy_of_f (cdr l))))))) (length1 length1 l))</pre>
<pre>((lambda (f) (f val1 val2 ... valn)) (lambda (copy_of_f copy_of_f sym1 sym2 ... symn) ... (copy_of_f copy_of_f val1 val2 ... valn) ...))</pre>	<pre>((lambda (length1) (length1 length1 l)) (lambda (copy_of_f copy_of_f l) (cond ((null? l) 0) (else (+ 1 (copy_of_f copy_of_f (cdr l)))))))</pre>

Table 5.5 Reducing letrec to lambda (All rows of each column are semantically equivalent.)

	General Pattern	Instance of Pattern
let (parallel)	<pre>(let ((sym1 val1) (sym2 val2) ... (symn valn)) sym1 and sym2 are only visible here in body)</pre>	<pre>(let ((a 1) (b 2)) ; a and b are only visible here (+ a b))</pre>
let* (sequential)	<pre>(let* ((sym1 val1) (sym2 sym1) ... (symn sym2)) sym1 sym2 ... symn are visible here in body)</pre>	<pre>(let* ((a 1) (b (+ a 1))) ; a is visible here and beyond ; a and b are visible here in body (+ a b))</pre>
letrec (recursive)	<pre>(letrec ((f (lambda (sym1 sym2 ... symn) ... (f val11 val12 ... val1nn) ...))) (f val1 val2 ... valn))</pre>	<pre>(letrec ((length1 (lambda (l) (cond ((null? l) 0) (else (+ 1 (length1 (cdr l))))))) (length1 '(a b c d)))</pre>

Table 5.6 Semantics of let, let*, and letrec

be demonstrated in these languages. We also encourage readers to explore and work through some of the programming exercises in online Appendices B and C, where we provide fundamental language and programming background in ML and Haskell, respectively, which is requisite for understanding some of the material and examples in Chapters 7–9. Doing so will also help you apply your understanding of functional programming to learning additional languages supporting that style of programming. (While Lisp is a typeless language, types—and reasoning about them—play a prominent role in programming in ML and Haskell.)

Conceptual Exercises for Section 5.9

Exercise 5.9.1 Explain the difference between *binding* and *assignment*.

Exercise 5.9.2 Read Paul Graham’s essay “Beating the Averages” from the book *Hackers and Painters* (2004a, Chapter 12), available at <http://www.paulgraham.com/avg.html>, and write a 250-word commentary on it.

Programming Exercises for Section 5.9

Exercise 5.9.3 Define and apply a recursive list length function in a single `let` expression (i.e., a `let` expression containing no nested `let` expressions). *Hint:* Use `set!`.

Exercise 5.9.4 Using `letrec`, define mutually recursive `odd?` and `even?` predicates to demonstrate that bindings are available for use within and before the blocks for definitions in the `letrec` are evaluated.

Exercise 5.9.5 Define a Scheme function `reverse1` that accepts only an *S-list* `s` as an argument and reverses the elements of `s` in *linear* time (i.e., time directly proportional to the size of `s`), $O(n)$. You may use *only* `define`, `lambda`, `let`, `cond`, `null?`, `cons`, `car`, and `cdr` in `reverse1`. Do *not* use `append` or `letrec` in your definition. Define only one function.

Examples:

```
> (reverse1 '(1 2 3 4 5))
(5 4 3 2 1)
> (reverse1 '(1))
(1)
> (reverse1 '(2 1))
(1 2)
> (reverse1 '(Twelfth Night and day))
(day and Night Twelfth)
> (reverse1 '(1 (2 (3)) (4 5)))
((4 5) (2 (3)) 1)
```

Exercise 5.9.6 Rewrite the following `let` expression as an equivalent `lambda` expression containing no nested `let` expressions while maintaining the bindings of `a` to 1 and `b` to `(+ a 1)`:

```
(let ((a 1))
  (let ((b (+ a 1)))
    (+ a b)))
```

Exercise 5.9.7 Rewrite the following `letrec` expression as an equivalent `let` expression while maintaining the binding of `sum` to the recursive function. However, do not use a *named* `let`. Do not use `define`:

```
(letrec ((sum (lambda (lon)
  (cond
    ((null? lon) 0)
    (else (+ (car lon) (sum (cdr lon)))))))
  (sum '(2 4 6 8 10)))
```

Exercise 5.9.8 Rewrite the following `let` expression as an equivalent `lambda` expression while maintaining the binding of `sum` to the recursive function. Do not use `define`:

```
(let ((sum (lambda (s l)
  (cond
    ((null? l) 0)
    (else (+ (car l) (sum s (cdr l)))))))
  (sum sum '(1 2 3 4 5)))
```

Exercise 5.9.9 Rewrite the following Scheme `member1?` function without a `let` expression (and without side effect) while maintaining the binding of `head` to `(car lat)` and `tail` to `(cdr lat)`. Only define one function. Do not use `let*`, `letrec`, `set!`, or any imperative features, and do not compute any single subexpression more than once.

```
(define member1?
  (lambda (a lat)
    (let ((head (car lat)) (tail (cdr lat)))
      (cond
        ((null? lat) #f)
        ((eqv? a head) #t)
        (else (member1? a tail))))))
```

Exercise 5.9.10 Complete Programming Exercise 5.9.9 without the use of `define`.

Exercise 5.9.11 Rewrite the following Scheme expression in λ -calculus:

```
((lambda (a b) (+ a b)) 1 2)
```

Exercise 5.9.12 Rewrite the following Scheme expression in λ -calculus:

```
(let* ((x 1) (y (+ x 1)))
  ((lambda (a b) (+ a b)) x y))
```

5.10 Advanced Techniques

Since `let`, `let*`, and `letrec` expressions can be reduced to `lambda` expressions, their use does not violate the spirit of functional programming. In turn, we use them for purposes of program readability. Moreover, their use can improve the efficiency (in time and space) of our programs, as we demonstrate in this section. We start by developing some list functions to be used later in our demonstrations.

5.10.1 More List Functions

The function `remove_first` removes the first occurrence of an atom `a` from a list of atoms `lat`:

```
1 (define remove_first
2   (lambda (a lat)
3     (cond
4       ((null? lat) '())
5       ((eqv? a (car lat)) (cdr lat))
6       (else (cons (car l) (remove_first a (cdr lat)))))))
```

Here the `eqv?` predicate returns true if its two arguments are equal and false otherwise. The function `remove_all` extends `remove_first` by removing *all* occurrences of an atom `a` from a list of atoms `lat` by simply returning `(remove_all (cdr lat))` in line 5 rather than `(cdr lat)`:

```
(define remove_all
  (lambda (a lat)
    (cond
      ((null? lat) '())
      ((eqv? a (car lat)) (remove_all a (cdr lat)))
      (else (cons (car lat) (remove_all a (cdr lat)))))))
```

We would like to extend `remove_all` so that it removes *all* occurrences of an atom `a` from *any* S-list, not just a list of atoms. Recall that recursive thought in functional programming involves learning and recognizing patterns (*Design Guideline 2: Specific Patterns of Recursion*). Using the third pattern in *Design Guideline 2* results in:¹¹

```
1 (define remove_all*
2   (lambda (a l)
3     (cond
4       ((null? l) '())
5       ((atom? (car l))
6        (cond
7          ((eqv? a (car l)) (remove_all* a (cdr l)))
8          (else (cons (car l) (remove_all* a (cdr l)))))
9       (else (cons (remove_all* a (car l))
10                  (remove_all* a (cdr l)))))))
```

11. A Scheme convention followed in this text is to use a `*` as the last character of any function name that recurses on an S-expression (e.g., `remove_all*`), whenever a corresponding function operating on a list of atoms is also defined (Friedman and Felleisen 1996a, Chapter 5).

Notice that in developing these functions, the pattern of recursion strictly follows *Design Guideline 2*.

5.10.2 Eliminating Expression Recomputation

Notice that in any single application of the function `remove_all*` with a non-empty list, the expression `(car l)` is computed twice—once on line 5, and once on either line 7, 8, or 9—with the same value of `l`. Note that `(cdr l)` is never computed more than once, because only one of lines 7, 8, and 10 can be evaluated at any one time through the function. Functional programs usually run more slowly than imperative programs because (1) languages supporting functional programming are typically interpreted; (2) recursion, the primary method for repetition in functional programs, is slower than iteration due to the overhead of the run-time stack; and (3) the pass-by-value parameter-passing mechanism is inefficient. However, barring interpretation and recursion, recomputing expressions only makes the program slower. We can bind the results of common expressions using a `let` expression to avoid recomputing the results of those expressions (*Design Guideline 4: Name Recomputed Subexpressions*):

```

1  (define remove_all*
2    (lambda (a l)
3      (cond
4        ((null? l) '())
5        (else (let ((head (car l)))
6                  (cond
7                    ((atom? head)
8                     (cond
9                      ((eqv? a head) (remove_all* a (cdr l)))
10                     (else (cons head (remove_all* a (cdr l))))))
11                  (else (cons (remove_all* a head)
12                              (remove_all* a (cdr l))))))))))

```

Notice that binding the result of the evaluation of the expression `(cdr l)` to the mnemonic `tail`, while improving readability, does not actually improve performance. While the expression `(cdr l)` appears more than once in this definition (lines 9, 10, and 12), it is computed only once per function invocation.

5.10.3 Avoiding Repassing Constant Arguments Across Recursive Calls

The last version of `remove_all*` still has a problem. Every time the function is called, it is passed the atom `a`, which never changes. Since Scheme uses pass-by-value semantics for parameter passing, passing an argument with the same value across multiple recursive calls is inefficient and unnecessary. We can factor out constant parameters using a `letrec` expression that accepts all but the constant parameter (*Design Guideline 6: Factor out Constant Parameters*). This gives us the final version of `remove_all*`:


```

1  (define remove_all*
2    (lambda (a l)
3      (letrec ((remove_all_helper*
4                (lambda (l)
5                  (cond
6                    ((null? l) '())
7                    (else (let ((head (car l)))
8                          (cond
9                            ((atom? head)
10                             (cond
11                               ((eqv? a head)
12                                (remove_all_helper* (cdr l)))
13                               (else
14                                (cons head
15                                   (remove_all_helper*
16                                     (cdr l)))))))
17                             (else
18                              (cons
19                               (remove_all_helper* head)
20                               (remove_all_helper*
21                                 (cdr l))))))))))
22      (remove_all_helper* l)))

> (remove_all* 'nothing
  '(((nothing)) ((will) (() ()) (come ()) (of nothing))))
  '((((())) ((will) (() ()) (come ()) (of))))

```

This version of `remove_all*` works because within the scope of `remove_all*` (lines 3–22), the parameter `a` is visible. We can think of it as global just within that block of code. Since it is visible in that range, it need not be passed to any function defined (either with a `let`, `let*`, or `letrec` expression) in that block, since any function defined within that scope already has access to it. Therefore, we defined a nested function `remove_all_helper*` that accepts only a list `l` as an argument. The parameter `a` is not passed to `remove_all_helper*` in the calls to it on lines 12, 15, and 18–20 (only a smaller list is passed), even though within the body of `remove_all_helper*` the parameter `a` (from the function `remove_all*`) is referenced. The concept of scope can be viewed as an instance of the more general concept of binding in programming languages, as discussed in Chapter 6. For instance, the *scope rules* of a language specify to which declaration of an identifier a reference to that identifier is *bound*. When improving functions using these techniques, remember to follow *Design Guideline 8: Correctness First, Simplification Second*.

Readers may have noticed a subtle, though important, difference in how we nest functions in the final definitions of `reverse1` and `remove_all*`. The `lambda` expression for the `reverse1` function is defined in the body of the `letrec` expression that binds the nested `rev` function. The opposite is the case with `remove_all*`: The `remove_all_helper*` nested function is bound within the definition of the `remove_all` function (i.e., the `lambda` expression for it). The following code fragments help highlight the difference in these two styles:

```

1  ;; style used to define remove_all*
2  (lambda (a)
3
4    ;; body of lambda expression

```

```

5      (letrec ((f (lambda (<parameter list>) ...)) ...)
6
7          ;; body of letrec expression
8          ;; parameter a is accessible here
9
10         ;; call to f
11         ... (f ...) ...))
12
13 ;; style used to define reverse1
14 (letrec ((f (lambda (<parameter list>)
15
16             ;; parameter a is not accessible here
17
18             ;; call to f
19             ... (f ...) ...)))
20
21 ;; body of letrec expression
22 (lambda (a)
23
24     ;; body of lambda expression
25     ;; parameter a is accessible here
26
27     ;; call to f
28     ... (f ...) ...))

```

This distinction is important. If the nested function `f` must access one or more of the parameters (i.e., *Design Guideline 6*), which is the case with `remove_all*`, then the style illustrated in lines 1–11 must be used. Conversely, if one or more of the parameters to the outer function should be hidden from the nested function, which is the case with `reverse1`, then the style used on lines 13–28 must be used. If we apply these guidelines to improve the last definition of `list-of`, we determine that while the nested function `list-of-helper` does need to know about the predicate argument to the outer function, `predicate` does not change—so it need not be passed through each successive recursive call. Therefore, we should nest the `letrec` within the `lambda`:

```

(define list-of
  (lambda (predicate)
    (letrec ((list-of-helper
              (lambda (lst)
                (or (null? lst)
                    (and (pair? lst)
                        (predicate (car lst))
                        (list-of-helper (cdr lst)))))))
      list-of-helper)))

```

While the choice of which of the two styles is most appropriate for a program depends on the context of the problem, in some cases in functional programming it is a matter of preference. Consider the following two `letrec` expressions, both of which yield the same result:

```

1 > (letrec ((length1 (lambda (l)
2 >                      (cond
3 >                        ((null? l) 0)
4 >                        (else (+ 1 (length1 (cdr l)))))))
5 >   (length1 '(a b c d e)))

```

```

6 5
7
8 > ((letrec ((length1 (lambda (l)
9 >                               (cond
10 >                                ((null? l) 0)
11 >                                (else (+ 1 (length1 (cdr l)))))))
12 >   length1) '(1 2 3 4 5))
13 5

```

While these two expressions are functionally equivalent (i.e., they have the same denotational semantics), they differ in operational semantics. The first expression (lines 1–5) calls the local function `length1` in the body of the `letrec` (line 5). The second expression (lines 8–12) first returns the local function `length1` in the body of the `letrec` (line 12) and then calls it—notice the double parentheses to the left of `letrec` on line 8. The former expression uses binding to invoke the function `length1`, while the latter uses binding to return the function `length1`.

Programming Exercises for Section 5.10

Exercise 5.10.1 Redefine the `applytoall` function from Programming Exercise 5.4.4 so that it follows *Design Guidelines 5* and *6*.

Exercise 5.10.2 Redefine the `member1?` function from Programming Exercise 5.6.1 so that it follows *Design Guidelines 5* and *6*.

Exercise 5.10.3 Define a Scheme function `member*?` that accepts only an atom and an *S-list* (i.e., a list possibly nested to an arbitrary depth), in that order, and returns `#t` if the atom is an element found anywhere in the *S-list* and `#f` otherwise.

Examples:

```

> (member*? 'a '())
#f
> (member*? 'a '(()))
#f
> (member*? 'a '(() ()))
#f
> (member*? 'c '(a b c d))
#t
> (member*? 'e '(a (b) () (c ()) () d ((e))))
#t
> (member*? 'd '(((a b)) (c) () d (((e) () ((f))))))
#t
> (member*? 'i '(a (b c) (((d)) (e)) (f g (h))))
#f

```

Exercise 5.10.4 Redefine the `member*?` function from Programming Exercise 5.10.3 so that it follows *Design Guidelines 4–6*.

Exercise 5.10.5 Redefine the `makeset` function from Programming Exercise 5.6.3 so that it follows *Design Guideline 4*.

Exercise 5.10.6 Redefine the `cycle` function from Programming Exercise 5.6.5 so that it follows *Design Guideline 5*.

Exercise 5.10.7 Redefine the `transpose` function from Programming Exercise 5.6.6 so that it follows *Design Guideline 4*.

Exercise 5.10.8 Redefine the `oddevensum` function from Programming Exercise 5.6.7 so that it follows *Design Guideline 4*.

Exercise 5.10.9 Define a Scheme function `count-atoms` that accepts only an *S-list* as an argument and returns the number of atoms that occur in that *S-list* at all levels. You may use the `atom?` function given in Section 5.8.1. Follow *Design Guideline 4*.

Examples:

```
> (count-atoms '(a b c))
3
> (count-atoms '(a (b c) d (e f)))
6
> (count-atoms '(((a 1.2) (b (c d) 3.14) (e))))
7
> (count-atoms '(nil nil (nil nil) nil))
5
```

Exercise 5.10.10 Define a Scheme function `flatten1` that accepts only an *S-list* as an argument and returns it flattened as a list of atoms.

Examples:

```
> (flatten1 '())
()
> (flatten1 '(()))
()
> (flatten1 '(() ()))
()
> (flatten1 '(a b c d))
(a b c d)
> (flatten1 '(a (b) () (c ()) () d ((e))))
(a b c d e)
> (flatten1 '(((a b)) (c) () d ((e) () ((f)))))
(a b c d e f)
> (flatten1 '(a (b c) ((d)) (e)) (f g (h))))
(a b c d e f g h)
```

Exercise 5.10.11 Redefine the `flatten1` function from Programming Exercise 5.10.10 so that it follows *Design Guideline 4*.

Exercise 5.10.12 Define a function `samefringe` that accepts an integer `n` and two *S-expressions*, and returns `#t` if the first non-null `n` atoms in each *S-expression* are equal and in the same order and `#f` otherwise.

Examples:

```
> (samefringe 2 '(1 2 3) '(1 2 3))
#t
> (samefringe 2 '(1 1 2) '(1 2 3))
#f
```

```

> (samefringe 5 '(1 2 3 (4 5)) '(1 2 (3 4) 5))
#t
> (samefringe 5 '(1 ((2) 3) (4 5)) '(1 2 (3 4) 5))
#t
> (samefringe 5 '(1 6 3 (7 5)) '(1 2 (3 4) 5))
#f
> (samefringe 3 '(((1)) 2 (((3)))) '(1) (((((2)))) 3))
#t
> (samefringe 3 '(((1)) 2 (((3)))) '(1) (((((2)))) 4))
#f
> (samefringe 2 '((((a)) c))) '((a b)))
#f

```

Exercise 5.10.13 Redefine your solution to Programming Exercise 5.6.9 so that it follows *Design Guidelines 4* and *5*.

Exercise 5.10.14 Define a Scheme function `permutations` that accepts only a list representing a set as an argument and returns a list of all permutations of that list as a list of lists. You will need to define some nested auxiliary functions. Pass a λ -function to `map` where applicable in the bodies of the functions to simplify their definitions. Follow *Design Guideline 5*. *Hint*: This solution requires approximately 20 lines of code.

Examples:

```

> (permutations '())
'()
> (permutations '(1))
'((1))
> (permutations '(1 2))
'((1 2) (2 1))
> (permutations '(1 2 3))
'((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
> (permutations '(1 2 3 4))
'((1 2 3 4) (1 2 4 3) (1 3 2 4) (1 3 4 2) (1 4 2 3) (1 4 3 2)
  (2 1 3 4) (2 1 4 3) (2 3 1 4) (2 3 4 1) (2 4 1 3) (2 4 3 1)
  (3 1 2 4) (3 1 4 2) (3 2 1 4) (3 2 4 1) (3 4 1 2) (3 4 2 1)
  (4 1 2 3) (4 1 3 2) (4 2 1 3) (4 2 3 1) (4 3 1 2) (4 3 2 1))
> (permutations '("oranges" "and" "tangerines"))
'(("oranges" "and" "tangerines") ("oranges" "tangerines" "and")
  ("and" "oranges" "tangerines") ("and" "tangerines" "oranges")
  ("tangerines" "oranges" "and") ("tangerines" "and" "oranges"))

```

Exercise 5.10.15 Define a function `sort1` that accepts only a list of numbers as an argument and returns the list of numbers sorted in increasing order. Follow *Design Guidelines 4*, *5*, and *6* completely.

Examples:

```

> (sort1 '())
()
> (sort1 '(3))
(3)
> (sort1 '(3 2))
(2 3)
> (sort1 '(3 2 1))
(1 2 3)

```

```
(1 2 3)
> (sort1 '(9 8 7 6 5 4 3 2 1))
(1 2 3 4 5 6 7 8 9)
> (sort1 '(1 4 6 3 2))
(1 2 3 4 6)
```

Exercise 5.10.16 Use the *mergesort* sorting algorithm in your solution to Programming Exercise 5.10.15. Name your top-level function *mergesort*.

Exercise 5.10.17 Define a function *sort1* that accepts only a numeric comparison predicate and a list of numbers as arguments, in that order, and returns the list of numbers sorted by the predicate. Follow *Design Guidelines 4, 5, and 6* completely.

Examples:

```
> (sort1 < '())
()
> (sort1 < '(3))
(3)
> (sort1 < '(3 2))
(2 3)
> (sort1 < '(3 2 1))
(1 2 3)
> (sort1 < '(9 8 7 6 5 4 3 2 1))
(1 2 3 4 5 6 7 8 9)
> (sort1 > '())
()
> (sort1 > '(1))
(1)
> (sort1 > '(1 2))
(2 1)
> (sort1 > '(1 2 3))
(3 2 1)
> (sort1 > '(1 2 3 4 5 6 7 8 9))
(9 8 7 6 5 4 3 2 1)
```

Exercise 5.10.18 Use *mergesort* in your solution to Programming Exercise 5.10.17. Name your top-level function *mergesort*.

Exercise 5.10.19 Rewrite the final version of the *remove_all** function presented in this section without the use of any *letrec* or *let* expressions, without the use of *define*, and without the use of any function accepting more than one argument, while maintaining the bindings to the identifiers *remove_all**, *remove_all_helper**, and *head*. In other words, redefine the final version of the *remove_all** function in λ -calculus.

Exercise 5.10.20 A mind-bending exercise is to build an interpreter for Lisp in Lisp (i.e., a *metacircular interpreter*) in about a page of code. In this exercise, you are going to do so.

Start by reading *The Roots of Lisp* by P. Graham (2002), available at <http://www.paulgraham.com/rootsoflisp.html>. The article and the entire code are available at <https://lib.store.yahoo.net/lib/paulgraham/jmc.ps>. Sections 1–3 (pp. 1–7) should be a review of Lisp for you. Section 4 (p. 8) is the “surprise.”

Get the metacircular interpreter in Section 4 running; it is available at <http://ep.yimg.com/ty/cdn/paulgraham/jmc.lisp>. While it is written in Common Lisp, it does not take much work to convert it to Scheme or Racket. For instance, replace `defun` with `define`, and `label` with `letrec`. Most of the predicate functions in Common Lisp do not end with a `?` as they do in Racket. Thus, you must rewrite `null`, `atom`, and `eq` as `null?`, `atom?`, and `eqv?`, respectively. Also, in the `cond` expression replace the `'t`, which often appears in the final case with `else`. You might also name the main function `eval1` so not to override `eval` in Scheme or Racket. Refer to Graham (1993, Figure 20.1, p. 259), available at <http://www.paulgraham.com/onlisp.txt>, for a succinct list of key differences between Scheme and Common Lisp. Test the interpreter thoroughly. Verify it interprets the sample expressions on pp. 9–10 properly. It has been said that “C is a programming language for writing UNIX; Lisp is a language for writing Lisp.”

5.11 Languages and Software Engineering

Programming languages that support

- the construction of abstractions, and
- ease of program modification

also support

- ongoing development of a malleable program design, and
- the evolution of a prototype into product.

Let us unpack these aspects of software development.

5.11.1 Building Blocks as Abstractions

An objective of this chapter is to demonstrate the ease with which data structures (e.g., binary trees) and reusable programming abstractions (e.g., higher-order functions) are constructed in a functional style of programming. While Lisp is a simple (e.g., only two types: atom and S-list) and small language with a consistent and uniform syntax, its capacity for power and flexibility is vast, and these properties have compelling implications for software development. Previously, we built data structures and programming abstractions with only the three grammar rules of λ -calculus. Functional programming is much more an activity of discovering, creating, and then using and specializing the appropriate abstractions (like LEGO® bricks) for a set of related programming tasks than imperative programming is. As we progress through this book, we will build additional programming abstractions without inflating the language through which we express those abstractions—we mostly remain with the three grammar rules of λ -calculus. “[T]he key to flexibility, I think, is to make the language very

abstract. The easiest program to change is one that's very short" (Graham 2004b, p. 27). [While Lisp is a programming language, it pioneered the idea of language support for abstractions (Sinclair and Moon 1991).]

5.11.2 Language Flexibility Supports Program Modification

Another theme running through this chapter is that a functional style of programming in a flexible language supports ease of program modification. We not only organically constructed the functions and programs presented in this chapter, but also refined them repeatedly with ease. A programming language should support these micro-level activities. "It helps to have a medium that makes change easy" (Graham 2004b, p. 141). Paul Graham (1996, pp. 5–6) has made the observation that before the widespread use of oil paint in the fifteenth century, painters used *tempera*, which could not be mixed or painted over. Tempera made painters less ambitious because mistakes were costly. The advent of oil paint made painters' lives easier on a practical level. Similarly, a programming language should make it easy to modify a program. The interactive *read-eval-print* loop used in interpreted languages fosters rapid program development, modification, testing, and debugging. In contrast, programming in a compiled language such as C++ involves the use of a *program-compile-debug-recompile* loop.

5.11.3 Malleable Program Design

The ability to make more global changes to a program easily is especially important in the world of software development, where evolving specifications are a reality. A language not only should support (low-level) program modification, but also, more broadly, should support more global program design and redesign. A programming language should facilitate, and not handicap, an (inevitable) evolving design and redesign. In other words, a programming language should be an algorithm for program design and development, not just a tool to implement a design: "a language itself is a problem-solving tool" (Felleisen et al. 2018, p. 64).

5.11.4 From Prototype to Product

The logical extension of easily modifiable, malleable, and redesignable programs is the evolution of prototypes into products. The more important effect of the use of oil paint was that it empowered painters with the liberty to change their mind *in situ* (Murray and Murray 1963), and in doing so, removed the barriers to ambition and increased creativity and, as a result, ushered in a new style of painting (Graham 1996, pp. 5–6). In short, oil paint not only enabled micro changes, but also supported more macro-level changes in the painting and, thus, was the key ingredient that fostered the evolution of the prototype into the final work of art (Graham 2004b, pp. 220–221).

Like painting, programming is an art of exploration and discovery, and a programming language, like oil, should not only be a medium to accommodate changes in the software requirements and changes in the design thoughts of the

programmer (Graham 1996, p. 27), but should also support those higher-order activities.

In programming, an original design or prototype is typically sketched and used primarily for generating thoughts and discovering the parameters of the design space. For this reason, it is sometimes called a *throwaway prototype*. However, “[a] prototype doesn’t have to be just a model; you can refine it into the finished product. . . . It lets you take advantage of new insights you have along the way” (Graham 2004b, p. 221). Program design can then be informed by an invaluable source of practical insight: “the experience of implementing it.” (Graham 1996, p. 5). Like the use of oil in painting, we would like to discover a medium (in this case, a language and its associated tools) that reduces the cost of mistakes, not only tolerates, but even encourages second (and third and so on) thoughts, and, thus, favors exploration rather than planning.

Thus, a programming language and the tools available for use with it should not only dampen the effects of the constraints of the environment in which a programmer must work (e.g., changing specifications, incremental testing, routine maintenance, and major redesigns) rather than amplify them, but also foster design exploration, creativity, and discovery without the (typical) associated fear of risk.

The tenets of functional programming combined with a language supporting abstractions and dynamic bindings support these aspects of software development and empower programmers to embark on more ambitious projects (Graham 1996, p. 6). The organic, improvised style of functional programming demonstrated in this chapter is a natural fit. We did little to no design of the programs we developed here. As we journey deeper into functional programming, we encounter more general and, thus, powerful patterns, techniques, and abstractions.

5.12 Layers of Functional Programming

At the beginning of this chapter, we introduced functional programming using recursive-control behavior (see the bottommost layer of Figure 5.10). We then identified some inefficiencies in program execution resulting from that style of programming and embraced a more efficient style of functional programming (see the second layer from the bottom of Figure 5.10). We continue to evolve our programming style throughout this text as we go deeper into our study of programming languages. We discuss further the use of HOFs in Chapter 8 and move toward more efficient functional programming in Chapter 13. Each layer depicted in Figure 5.10 represents a shift in thinking about how to craft a solution to a problem and progressively refine it.

The bottom three layers apply to functional programming in general; the top two layers apply primarily to Lisp. Since Lisp is a homoiconic language—Lisp programs are Lisp lists—Lisp programs can generate Lisp code. Lisp programmers typically exploit the homoiconic nature of Lisp “by defining a kind of operator called a *macro*. Mastering macros is one of the most important steps in moving from writing correct Lisp programs to writing beautiful ones” (Graham 1993, p. vi). “As well as writing their programs down toward the language [(the

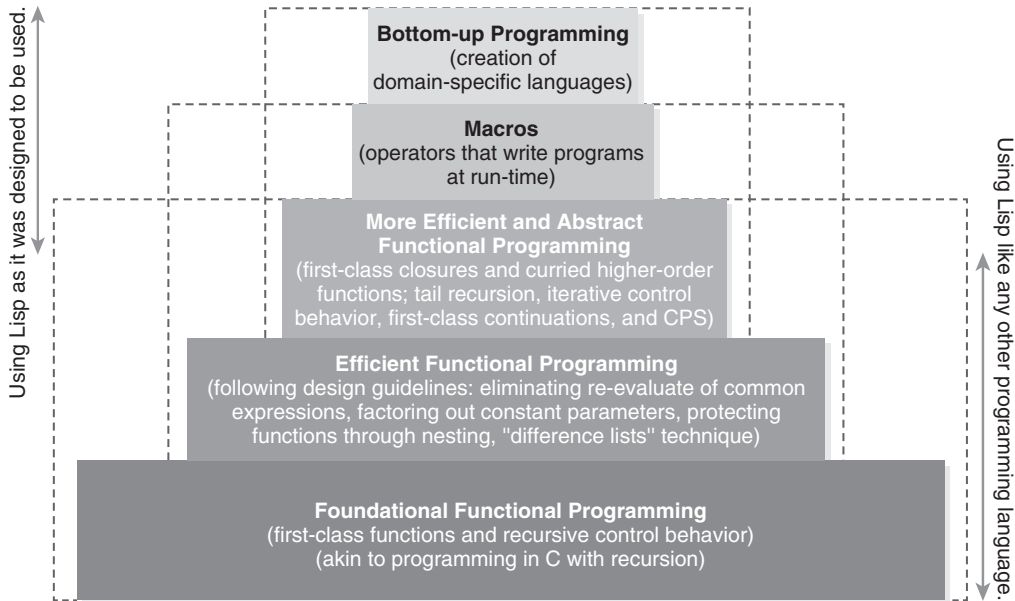


Figure 5.10 Layers of functional programming.

bottom three layers)), experienced Lisp programmers build the language up toward their programs [(the top two layers)]" (Graham 1993, p. v). Macros support the layer above them—leading to *bottom-up programming*. While the bottom three layers involve writing a target program (in Lisp), a bottom-up style of programming entails writing a target language (in Lisp) and then writing the target program in that language (Graham 1993, p. vi). "Not only can you program *in* Lisp (that makes it a programming language) but you can program the language itself" (Foderaro 1991, p. 27). The most natural way to use Lisp is for bottom-up programming (Graham 1993). "[A]ugmenting the language plays a proportionately larger role in Lisp style—so much so that Lisp is not just a different language, but a whole different way of programming" (Graham 1993, p. 4). This is not intended to convey the message that you cannot write top-down programs in Lisp—it is just that doing so does not unleash the full power of Lisp. We briefly return to bottom-up program design in Section 15.4. For more information on how to program using a bottom-up style, we refer readers to Graham (1993, 1996) and Krishnamurthi (2003).

5.13 Concurrency

As we conclude this chapter, we leave readers with a thought to ponder. We know from the study of operating systems that when two or more concurrent threads share a resource, we must synchronize their activities to ensure that the integrity of the resource is maintained and the system is never left in an inconsistent state—we must synchronize to avoid *data races*. Therefore, in the absence of side effects

and, thus, any shared state and/or mutable data, functional programs are natural candidates for parallelization:

You can't change the state of anything, and no function can have side effects, which is the reason why [functional programming] is ideal for distributing algorithms over multiple cores. You never have to worry about some other thread modifying a memory location where you've stored some value. You don't have to bother with locks and deadlocks and race conditions and all that mess. (Swaine 2009, p. 14)

There are now multiple functional concurrent programming languages, including Erlang, Elixir, Concurrent Haskell, and pH—a parallel Haskell from MIT. Joe Armstrong, who was one of the designers of Erlang, has claimed—with data to justify—that an Erlang application written to run on a single-core processor will run four times faster on a processor with four cores without requiring any modifications to the application (Swaine 2009, p. 15).

5.14 Programming Project for Chapter 5

Define a function `evaluate-expression` that accepts only a list argument, which represents a logical expression; applies the logical operators in the input expression; and returns a list of all intermediate results, including the final return value of the expression, which can be either `#t` or `#f`.

The expressions are represented as a parenthesized combination of `#t` (representing true), `#f` (representing false), `~` (representing not), `∨` (representing or), and `&` (representing and). In the absence of parentheses, normal precedence rules hold: `~` has the highest precedence, `&` has the second highest, and `∨` has the lowest. Assume left-to-right associativity. For instance, the expression `(#f ∨ #t & #f ∨ (~ #t))` is equivalent to `((#f ∨ (t & #f)) ∨ (~ #t))`. No two operators can appear in succession and the `~` will always be enclosed in parentheses. All input expressions will be valid.

Examples:

```

1 > (evaluate-expression '(#t))
2 '#t)
3 > (evaluate-expression '(#t & #f))
4 '#f)
5 > (evaluate-expression '(((((((#t))))))))
6 '#t)
7 > (evaluate-expression '(((((((#t)))))) & #f))
8 '#t #f)
9 > (evaluate-expression '(#f ∨ (#t & #f) & (#t ∨ #f)))
10 '#f #t #f #f)
11 > (evaluate-expression '(#f ∨ (#t & #f) ∨ #t))
12 '#f #f #t)
13 > (evaluate-expression '(#f ∨ (~ #t)))
14 '#f #f)
15 > (evaluate-expression
16 '(((~ #t) ∨ #t & (#f & (~ #f))) & #t & (~ (#t ∨ #f))))
17 '#f #t #f #f #f #f #t #f #f)

```

```

18 > (evaluate-expression '(#f V #t & #t & #f))
19 '#t
20 > (evaluate-expression '((~ #t) V (~#f) & #t))
21 '#f
22 > (evaluate-expression '((#f) & (#t) V (#f) & (~ #t)))
23 '#f
24 > (evaluate-expression '((((~ (((#t V #f)))) & ((~ #t))))))
25 '#t
26 > (evaluate-expression
27   '(((~ #t) V #t V (#f & (~ #f))) & #t & (~ (#t V #f))))
28 '#f
29 > (evaluate-expression '(#t & #t))
30 '#t
31 > (evaluate-expression '(#t & #t & (#t & #f)))
32 '#t
33 > (evaluate-expression '(#t & (#t & #f)))
34 '#f
35 > (evaluate-expression '(#t & (#t & #f) & (#f & #t)))
36 '#f
37 > (evaluate-expression '((#t V #f) & #t))
38 '#t
39 > (evaluate-expression '((#t V #f) & #t & #f))
40 '#t
41 > (evaluate-expression '((#t V #f) & (#t V #t)))
42 '#t
43 > (evaluate-expression '((#t V #f) & (#t V #t) V #t))
44 '#t
45 > (evaluate-expression '(#t V #t))
46 '#t
47 > (evaluate-expression '(#t V #t & (#t & #f)))
48 '#f
49 > (evaluate-expression '(#t V (#t & #f)))
50 '#f
51 > (evaluate-expression '(#t V (#t & #f) & (#f & #t)))
52 '#f
53 > (evaluate-expression '((#t V #f) V #t))
54 '#t
55 > (evaluate-expression '((#t V #f) V #t & #f))
56 '#t
57 > (evaluate-expression '((#t V #f) V (#t V #t)))
58 '#t
59 > (evaluate-expression '((#t V #f) V (#t V #t) V #t))
60 '#t

```

You may define one or more helper functions. Keep your program to approximately 120 lines of code. Use of the pattern-matching facility in Racket will significantly reduce the size of the evaluator to approximately 30 lines of code. See <https://docs.racket-lang.org/guide/match.html> for the details of pattern matching in Racket. (Try building a graphical user interface for this expression evaluator in Racket; see <https://docs.racket-lang.org/gui/>.)

5.15 Thematic Takeaways

- Functional programming unites beauty with utility.
- The λ -calculus, and the three grammar rules that constitute it, are sufficiently powerful. (Notice that we did not discuss much syntax in this chapter.)

- An important theme in a course on data structures and algorithms is that data structures and algorithms are natural reflections of each other. Therefore, “when defining a program based on structural induction, the structure of the program should be patterned after the structure of the data” (Friedman, Wand, and Haynes 2001, p. 12).
- Powerful programming abstractions can be constructed in a few lines of Scheme code.
- Recursion can be built into any programming language with support for first-class anonymous functions.
- “[L]earning Lisp will teach you more than just a new language—it will teach you new and more powerful ways of thinking about programs” (Graham 1996, p. 2).
- Improvements in software development methodologies have not kept pace with the improvements in computer hardware (e.g., multicore processors in smartphones) over the past 30 years. Such improvements in hardware have reduced the importance of speed of execution as a primary program design criterion. As a result, speed of development is now a more important criterion in the creation of software than it has been historically.

5.16 Chapter Summary

This chapter introduced readers to functional programming through the Scheme programming language. We established that a recursive thought process toward function conception and implementation is an essential tenet of functional programming. We studied λ -expressions; the definition of recursive functions; and `cons` cells, lists, and S-expressions. We studied the use of the `cons` cell as a primitive for building data structures, which we defined using BNF. Data structures and the functions that manipulate them are natural reflections of each other—the BNF grammar for the data structure provides a pattern for the function definition. We also explored improved program readability and local binding through `let`, `let*`, and `letrec` expressions, and demonstrated that such expressions can be reduced to λ -calculus and, therefore, are syntactic sugar. We saw how to implement recursion from first principles—by passing a recursive function to itself. We incrementally developed and followed a set of functional programming guidelines (Table 5.7).

In a study of Lisp, we are naturally confronted with some fundamental language principles. Although perhaps unbeknownst to the reader, we have introduced multiple concepts of programming languages in this chapter, such as binding (e.g., through the binding of arguments to parameters), scope (e.g., through nested `lambda` or `let` expressions), and parameter passing (e.g., pass-by-value). Binding is a universal concept in the study of programming languages because other language concepts (e.g., scope and parameter passing) involve binding. Any student who has completed an introductory course on computer programming in some high-level language has experienced these

1. **General Pattern of Recursion.** Solve the problem for the smallest instance of the problem (called the *base case*; e.g., $n = 0$ for $n!$, which is $n^0 = 1$). Assume the penultimate [i.e., $(n - 1)$ th, e.g., $(n - 1)!$] instance of the problem is solved and demonstrate how you can extend that solution to the n th instance of the problem [e.g., multiply it by n ; i.e., $n * (n - 1)!$].
2. **Specific Patterns of Recursion.** When recurring on a list of atoms, `lat`, the base case is an empty list [i.e., `(null? lat)`] and the recursive step is handled in the `else` clause. Similarly, when recurring on a number, `n`, the base case is, typically, $n = 0$ [i.e., `(zero? n)`] and the recursive step is handled in the `else` clause.

When recurring on a list of S-expressions, `l`, the base case is an empty list [i.e., `(null? l)`] and the recursive step involves two cases: (1) where the `car` of the list is an atom [i.e., `(atom? (car l))`] and (2) where the `car` of the list is itself a list (handled in the `else` clause, or vice versa).

3. **Efficient List Construction.** Use `cons` to build lists.
4. **Name Recomputed Subexpressions.** Use `(let (···) ···)` to name the values of repeated expressions in a function definition if they may be evaluated more than once for one and the same use of the function. Moreover, use `(let (···) ···)` to name the values of the expressions in the body of the `let` that are reevaluated every time a function is used.
5. **Nest Local Functions.** Use `(letrec (···) ···)` to hide and protect recursive functions and `(let (···) ···)` or `(let* (···) ···)` to hide and protect non-recursive functions. Nest a `lambda` expression within a `letrec` (or `let` or `let*`) expression:

```
(define f
  (letrec ((g (lambda (···) ···))) ; or let or let*
    (lambda (···) ···)))
```

6. **Factor out Constant Parameters.** Use `letrec` to factor out parameters whose arguments are constant (i.e., never change) across successive recursive applications. Nest a `letrec` (or `let` or `let*`) expression within a `lambda` expression:

```
(define member1
  (lambda (a lat)
    (letrec ((M (lambda (lat) ...)))
      (M lat))))
```

7. **Difference Lists Technique.** Use an additional argument representing the return value of the function that is built up across the successive recursive applications of the function when that information would otherwise be lost across successive recursive calls.
8. **Correctness First, Simplification Second.** Simplify a function or program, by nesting functions, naming recomputed values, and factoring out constant arguments, only after the function or program is thoroughly tested and correct.

Table 5.7 Functional Programming Design Guidelines

concepts though they may not have been aware of it. Binding is the topic of Chapter 6.

We also demonstrated how, within a small language (we focused on the λ -calculus as the substrate of Scheme), lies the core of computation through which powerful programming abstractions can be created and leveraged. We introduced the compelling implications of the properties of functional programming (and Lisp) for software development, such as prototypes evolving into deployable software, speed of program development vis-à-vis speed of program execution, bottom-up programming, and concurrency. While Lisp has a simple and uniform syntax, it is a powerful language that can be used to create advanced data structures and sophisticated abstractions in a few lines of code. Ultimately, we demonstrated that functional programming unites beauty with utility.

5.17 Notes and Further Reading

John McCarthy, the original designer of Lisp, received the ACM A. M. Turing Award in 1971 for contributions to artificial intelligence, including the creation of Lisp. For a detailed account of the history of Lisp we refer readers to McCarthy (1981). For a concise introduction to Lisp, we refer readers to Sussman, Steele, and Gabriel (1993).

In his 1978 Turing Award paper, John Backus described how the style of functional programming embraced by a language called FP is different from languages based on the λ -calculus:

An FP system is based on the use of a fixed set of combining forms called functional forms. These, plus simple definitions, are the only means of building new functions from existing ones; they use no variables or substitutions rules, and they become the operations of an associated algebra of programs. All the functions of an FP system are of one type: they map objects onto objects and always take a single argument. (Backus 1978, p. 619)

While FP was never fully embraced in the industrial programming community, it galvanized debate and interest in functional programming and subsequently influenced multiple languages supporting a functional style of programming (*Interview with Simon Peyton-Jones* 2017).

Design Guidelines 2–8 in Table 5.7 correspond to the *First*, *Second*, *Fifteenth*, *Thirteenth*, *Twelfth*, *Eleventh*, and *Sixth Commandments*, respectively, from Friedman and Felleisen (1996a, 1996b). The function `mystery` from Programming Exercise 5.6.9 is the function `scramble` from Friedman and Felleisen (1996b, pp. 11–15, 35, and 76). The functions `remove_first`, `remove_all`, `remove_all*` in Section 5.10.1 are from Friedman and Felleisen (1996a, Chapters 3 and 5), where they are called `rember`, `multirember`, and `rember*`, respectively.

For a derivation of the *Y combinator*, we refer readers to Gabriel (2001). For more information on *bottom-up programming*, we refer readers to Graham (1993, 1996) and Krishnamurthi (2003).

Scheme was the first Lisp dialect to use *lexical scoping*, which is discussed in Chapter 6. The language also required implementations of it to perform *tail-call optimization*, which is discussed in Chapter 13. Scheme was also the first language to support *first-class continuations*, which are an important ingredient for the creation of user-defined control structures and are also discussed Chapter 13.

Chapter 6

Binding and Scope

A rose by any other name would smell as sweet.

— William Shakespeare, *Romeo and Juliet*

B*inding*, as discussed in Chapter 1, is an association from one entity to another in a programming language or program (e.g., the variable `a` is bound to the data type `int`). Bindings were further discussed in Chapter 5 through and within the context of the Scheme programming language. Binding is one of the most foundational concepts in programming languages because other language concepts are examples of bindings. The main topic of this chapter, *scope*, is one such concept.

6.1 Chapter Objectives

- Describe *first-class closures*.
- Understand the meaning of the adjectives *static* and *dynamic* in the context of programming languages.
- Discuss *scope* as a type of binding from variable reference to declaration.
- Differentiate between *static* and *dynamic scoping*.
- Discuss the relationship between the lexical layout of a program and the representation and structure of a referencing environment for that program.
- Define *lexical addressing* and consider how it obviates the need for identifiers in a program.
- Discuss program translation as a means of improving the efficiency of execution.
- Learn how to resolve references in functions to parts of the program not currently executing (i.e., *the FUNARG problem*).
- Understand the difference between *deep*, *shallow*, and *ad hoc binding* in passing first-class functions as arguments to procedures.¹

1. In this text we refer to subprograms and subroutines as *procedures* and to procedures that return a value as *functions*.

6.2 Preliminaries

6.2.1 What Is a Closure?

An understanding of lexical closures is fundamental not only to this chapter, but more broadly to the study of programming languages. A *closure* is a function that *remembers* the lexical environment in which it was created. A closure can be thought of as a pair of pointers: one to a block of code (defining the function) and one to an environment (in which function was created). The bindings in the environment are used to evaluate the expressions in the code. Thus, a closure encapsulates *data* and *operations* and thus, bears a resemblance to an *object* as used in object-oriented programming. Closures are powerful constructs in functional programming (as we see throughout this text), and an essential element in the study of binding and scope.

6.2.2 Static Vis-à-Vis Dynamic Properties

In the context of programming languages, the adjective *static* placed before a noun phrase describing a binding, concept, or property of a programming language or program indicates that the binding to which the noun phrase refers takes place *before run-time*; the adjective *dynamic* indicates that the binding takes place *at run-time* (Table 6.1). For instance, the binding of a variable to a data type (e.g., `int a;`) takes place before run-time—typically at compile-time. In contrast, the binding of a variable to a value takes place at run-time—typically when an assignment statement (e.g., `a = 1;`) is executed.

6.3 Introduction

Implicit in the study of `let`, `let*`, and `letrec` expressions is the concept of *scope*. Scope is a concept that programmers encounter in every language. Since scope is often so tightly woven into the semantics of a language, we unconsciously understand it and rarely ever give it a second thought. In this chapter, we examine the details more closely.

In a program, variables appear as either *references* or *declarations*—even in typeless languages like Lisp that use *manifest typing*. The value named by a variable is called its *denotation*. Consider the following Scheme expression:

```

1 > ((lambda (x)
2 >   (+ 7
3 >     ((lambda (a b)
4 >       (+ a b x)) 1 2))) 5)
5 15

```

Static	bindings are <i>fixed</i>	<i>before</i> run-time. Example: <code>int a;</code>
Dynamic	bindings are <i>changeable</i>	<i>during</i> run-time. Example: <code>a = 1;</code>

Table 6.1 Static Vis-à-Vis Dynamic Bindings

The denotations of x , a , and b are 5, 1, and 2, respectively. The x on line 1 and the a and b on line 3 are *declarations*, while the a , b , and x on line 4 are *references*. A reference to a variable (e.g., the a on line 4) is *bound* to a declaration of a variable (e.g., the a on line 3).

Declarations have *limited* scope. The *scope* of a variable declaration in a program is the region of that program (i.e., a range of lines of code) within which references to that variable refer to the declaration (Friedman, Wand, and Haynes 2001). For instance, the scope of the declaration of a in the preceding example is line 4—the same as for b . The scope of the declaration of x is lines 2–4. Thus, the same identifier can be used in different parts of a program for different purposes. For instance, the identifier i is often used as the loop control variable in a variety of different loops in a program, and multiple functions can have a parameter x . In each case, the scope of the declaration is limited to the body of the loop or function, respectively.

The *scope rules* of a programming language indicate to which declaration a reference is bound. Languages where that binding can be determined by examining the text of the program *before run-time* use *static scoping*. Languages where the determination of that binding requires information available *at run-time* use *dynamic scoping*. In the earlier example, we determined the declarations to which references are bound as well as the scope of declarations based on our knowledge of the Scheme programming language—in other words, without consulting any formal rules.

6.4 Static Scoping

Static scoping means that the declaration to which a reference is bound can be determined before run-time (i.e., statically) by examining the text of the program. Static scoping was introduced in ALGOL 60 and has been widely adopted by most programming languages. The most common instance of static scoping is *lexical scoping*, in which the scope of variable declaration is based on the program's lexical layout. Lexical scoping and static scoping are not synonymous (Table 6.2). Examining the lexical layout of a program is one way to determine the scope of a declaration before run-time, but other strategies are also possible. In lexically scoped languages, the scope of a variable reference is the code constituting its static ancestors.

6.4.1 Lexical Scoping

To determine the declaration associated with a reference in a lexically scoped language, we must know that language's scope rules. The scope rules of a language are semantic rules.

Scope Rule for λ -calculus: In $(\text{lambda } (<\text{identifier}>) <\text{expression}>)$, the occurrence of $<\text{identifier}>$ is a declaration that binds all occurrences of that variable in $<\text{expression}>$ unless some intervening declaration of the same variable occurs. (Friedman, Wand, and Haynes 2001, p. 29).

Static scoping	A reference is bound to a declaration <i>before</i> run-time, e.g., based on the spatial relationship of nested program blocks to each other, i.e., <i>lexical scoping</i> .
Dynamic scoping	A reference is bound to a declaration <i>during</i> run-time, e.g., based on the calling sequences of procedures on run-time call stack.

Table 6.2 Static Scoping Vis-à-Vis Dynamic Scoping

In discussing lexical scoping, to understand what *intervening* means in this rule, it is helpful to introduce the notion of a block. A *block* is a syntactic unit or group of cohesive lines of code for which the beginning and ending of the group are clearly demarcated—typically by lexemes such as curly braces (as in C). An example is `if (x > 1) { /* this is a block */ }`. In Scheme, `let` expressions and functions are blocks. Lines 3–4 in the example in Section 6.3 define a block. A programming language whose programs are structured as series of blocks is a *block-structured language*. Blocks can be nested, meaning that they can contain other blocks. For instance, consider the following Scheme expression:

```

1 > ((lambda (x)
2 >   (+ 7
3 >     ((lambda (a b)
4 >       (+ a
5 >         ((lambda (c a)
6 >           (+ a b x)) 3 4))) 1 2))) 5)
7 19

```

This entire expression (lines 1–6) is a block, which contains a nested block (lines 2–6), which itself contains another block (lines 3–6), and so on. Lines 5–6 are the *innermost* block and lines 1–6 constitute the *outermost* block; lines 3–6 make up an *intervening* block. The spatial nesting of the blocks of a program is depicted in a *lexical graph*:

$$\text{lambda}^{(x)} \rightarrow + \rightarrow \text{lambda}^{(a\ b)} \rightarrow + \rightarrow \text{lambda}^{(c\ a)} \rightarrow (+\ a\ b\ x)$$

Scheme, Python, Java, and C are block-structured languages; Prolog and Forth are not. Typically block-structured languages are primarily lexically scoped, as is the case for Scheme, Python, Java, and C.

A simple procedure can be used to determine the declaration to which a reference is bound. Start with the innermost block of the expression containing the reference and search within it for its declaration. If it is not found there, search the next block enclosing the one just searched. If the declaration is not found there, continue searching in this innermost-to-outermost fashion until a declaration is found. After searching the outermost block, if a declaration is not found, the variable reference is *free* (as opposed to *bound*).

Due to the scope rules of Scheme and the lexical layout of the program (i.e., the nesting of the expressions) that it relies upon, applying this procedure reveals that

the reference to `x` in line 6 of the example Scheme expression previously is bound to the declaration of `x` on line 1. Neither the scope rule nor the procedure yields the scope of a declaration. The scope of a declaration is the region of the program within which references refer to the declaration. In this example, the scope of the declaration of `x` is lines 2–6.

The scope of the declaration of `a` on line 3, by contrast, is lines 4–5 rather than lines 4–6, because the inner declaration of `a` on line 5 *shadows* the outer declaration of `a` on line 3. The inner declaration of `a` on line 5 creates a *scope hole* on line 6, so that the scope of the declaration of `a` on line 3 is lines 4–5 and not lines 4–6. Thus, a declaration may shadow another and create a scope hole. For this reason, we now make a distinction between the visibility and scope of a declaration—though the two concepts are often used interchangeably. The *visibility* of a declaration in a program constitutes the regions of that program where references are bound to that declaration—this is the definition of scope given and used previously. *Scope* refers to the entire block of the program where the declaration is applicable. Thus, the scope of a declaration includes scope holes since the bindings still exist, but are hidden. The visibility of a declaration is a subset of the scope of that declaration and, therefore, is bounded by the scope. The visibility of a declaration is not always the entire body of a `lambda` expression owing to the possibility of holes. As an example, the following figure graphically depicts the declarations to which the references to `a`, `b`, and `x` are bound. Nesting of blocks progresses from left to right. On line 2, the declaration of `a` on line 3 is not in scope:

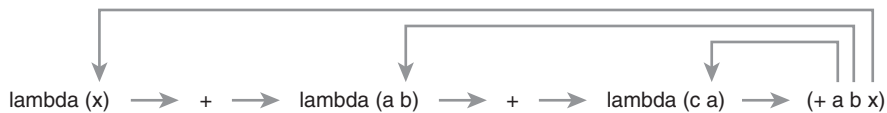


Figure 6.1 depicts the run-time stack at the time the expression `(+ a b x)` is evaluated.

Design Guideline 6: Factor out Constant Parameters in Table 5.7 indicates that we should nest a `letrec` within a `lambda` only when the body of the `letrec` must

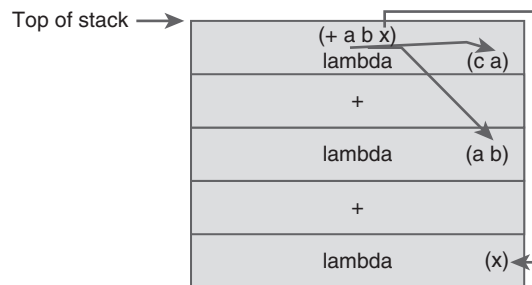


Figure 6.1 Run-time call stack at the time the expression `(+ a b x)` is evaluated. The arrows indicate to which declarations the references to `a`, `b`, and `x` are bound.

know about arguments to the outer function. For instance, as recursion progresses in the `reverse1` function, the list to be reversed changes (i.e., it gets smaller). In turn, in Section 5.9.3 we defined the `reverse1` function (i.e., the `lambda`) in the body block of the `letrec` expression. For purposes of illustrating a scope hole, we will do the opposite here; that is, we will nest the `letrec` within the `lambda`. (We are not implying that this is an improvement over the other definition.)

```

1  (define reverse1
2    (lambda (l)
3      (letrec ((rev
4                (lambda (lst rl)
5                  (cond
6                    ((null? lst) rl)
7                    (else (rev (cdr lst)
8                               (cons (car lst) rl)))))))
9        (cond
10         ((null? l) '())
11         (else (rev l '())))))

```

Based on our knowledge of shadowing and scope holes, we know there is no need to use two different parameter names (e.g., `l` and `lst`) because the inner `l` shadows the outer `l` and creates a scope hole in the body of the inner `lambda` expression (which is the desired behavior). Thus, the definition of `reverse1` can be written as follows, where all occurrences of the identifier `lst` in the prior definition are replaced with `l`:

```

(define reverse1
  (lambda (l)
    (letrec ((rev
              (lambda (l rl)
                (cond
                  ((null? l) rl)
                  (else (rev (cdr l) (cons (car l) rl))))))
      (cond
        ((null? l) '())
        (else (rev l '())))))

```

A reference can either be local or nonlocal. A *local reference* is bound to a declaration in the set of declarations (e.g., the formal parameter list) associated with the innermost block in which that reference is contained. Sometimes that block is called the *local* block. Note that not all blocks have a set of declarations associated with them; an example is `if (a == b) { c = a + b; d = c + 1; }` in Java. The reference to `a` on line 6 in the expression given at the beginning of this section is a local reference with respect to the `lambda` block on lines 5–6, while the references to `b` and `x` on line 6 are nonlocal references with respect to that block. All of the nested blocks enclosing the innermost block containing the reference are sometimes referred to as *ancestor* blocks of that block. In a lexically scoped language, we search both the local and ancestor blocks to find the declaration to which a reference is bound.

Since we implement interpreters for languages in this text, we must cultivate the habit of thinking in a language-implementation fashion. Thinking in an

implementation-oriented manner helps us understand how bindings can be hidden. *We must determine the declaration to which a reference is bound so that we can determine the value bound to the identifier at that reference so that we can evaluate the expression containing that reference.* This leads to the concept of an environment, which is a core element of any interpreter. Recall from Chapter 5 that a *referencing environment* is a set or mapping of name–value pairs that associates variable names (or symbols) with their current bindings at any point in a program in a programming language implementation. To summarize:

$$\begin{aligned}\text{scope}(\langle \text{declaration} \rangle) &= \langle \text{a set of program points} \rangle \\ \text{referencing environment}(\langle \text{a program point} \rangle) &= \langle \text{a set of variable bindings} \rangle\end{aligned}$$

The set of declarations associated with the innermost block in which a reference is contained differs from the referencing environment, which is typically much larger because it contains bindings for nonlocal references, at the program point where that reference is made. For instance, the referencing environment at line 6 in the expression given at the beginning of this section is $\{(a, 4), (b, 2), (c, 3), (x, 5)\}$ while the declarations associated with the innermost block containing line 6 is $((c\ 3)\ (a\ 4))$.

There are two perspectives from which we can study scope (i.e., the determination of the declaration to which a reference is bound): the programmer and the interpreter. The programmer, or a human, follows the innermost-to-outermost search process described previously. (Programmers typically do not think through the referencing environment.) Internally, that process is operationalized by the interpreter as a search of the environment. In turn, (static or dynamic) scoping (and the scope rules of a language) involves how and when the referencing environment is searched in the interpreter.

In a statically scoped language, that determination can be made before run-time (often by a human). In contrast, in a statically scoped, interpreted language, the interpreter makes that determination at run-time because that is the only time during which the interpreter is in operation. Thus, an interpreter progressively constructs a referencing environment for a computer program during execution.

While the specific structure of an environment is an implementation issue extraneous to the discussion at hand (though covered in Chapter 9), some cursory remarks are necessary. For now, we simply recognize that we want to represent and structure the environment in a manner that renders searching it efficient with respect to the scope rules of a language. Therefore, if the human process involves an innermost-to-outermost search, we would like to structure the environment so that bindings of the declarations of the innermost block are encountered before those in any ancestor block. One way to represent and structure an environment in this way is as a list of lists, where each list contains a list of name–value pairs representing bindings, and where the lists containing the bindings are ordered such that the bindings from the innermost block appear in the `car` position (the head) of the list and the declarations from the

ancestor blocks constitute the `cdr` (the tail) of the list organized in innermost-to-outermost order. Using this structure, the referencing environment at line 6 is represented as `(( (c 3) (a 4)) ((a 1) (b 2)) ((x 5)))`. These are the scoping semantics with which most of us are familiar. Representation options for the structure of an environment (e.g., flat list, nested list, tree) as well as how an environment is progressively constructed are the topic of Section 9.8.

Conceptual Exercises for Section 6.4

In each of the following two exercises, draw an arrow from each variable reference in the given λ -calculus expression to the declaration to which it is bound.

Exercise 6.4.1

```
((lambda (length1) ((length1 length1) ' (a b c d)))
  (lambda (fun_length)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (+ 1 ((fun_length fun_length) (cdr l))))))))
```

Exercise 6.4.2

```
(lambda (f)
  ((lambda (x)
    (f (lambda (y) ((x x) y))))
   (lambda (x)
    (f (lambda (y) ((x x) y))))))
```

Exercise 6.4.3 In programming languages that do not require the programmer to declare variables (e.g., Python), there is often no distinction between the declaration of a variable and the first reference to it without the use of qualifier. (Sometimes this concept is called *manifest typing* or *implicit typing*.) For instance, in the following Python program, is line 3 a reference to the declaration of an `x` on line 1 or a (new) declaration itself?

```
1 x = 10
2 def f():
3     x = 11
4     f()
```

(See Appendix A for an introduction to the Python programming language.) The following program suffers from a similar ambiguity. Is line 4 a reference bound to the declaration on line 2 or does it introduce a new declaration that shadows the declaration on line 2?

```
1 def f():
2     x = 10
3     def g():
4         x = 11
```

```

5     g()
6     return x
7     print(f())

```

Investigate the semantics of the keywords `global` and `nonlocal` in Python. How do they address the problem of discerning whether a line of code is a declaration or a reference? What are the semantics of `global x`? What are the semantics of `nonlocal x`?

6.5 Lexical Addressing

Identifiers are necessary for writing programs, but unnecessary for executing them. To see why, we annotate the environment from the expression given at the beginning of Section 6.4.1 with indices representing lexical depth and declaration position. Assume we number the innermost-to-outermost blocks of an expression from 0 to n . *Lexical depth* is an integer representing a block with respect to all of the nested blocks it contains. Further, assume that we number each formal parameter in the declaration list associated with each block from 0 to m . The *declaration position* of a particular identifier is an integer representing the position in the list of identifiers of a `lambda` expression of that identifier.

Table 6.3 illustrates the annotated environment for the expression given at the beginning of Section 6.4.1. We can think of this representation of the environment as a reduction of each block to the list of declarations with which it is associated. Those lists are then organized and numbered from innermost to outermost, and each element within each list represents a specific declaration, which are also numbered in each list. In this way, each reference in an expression can be reduced to a lexical depth and declaration position. For instance, the lexical depth and the declaration position of the reference to `a` on line 6 are 0 and 1, respectively. Given the representation and structure of this environment and this annotation style, identifying the lexical depth and declaration position is simple: Search the environment list shown in Table 6.3 from left to right; when an identifier is encountered that matches the reference, return the depth and position. This search is the interpreter analog of a manual search of the lexical layout of the program text conducted by the programmer.

We can associate each variable reference with a (lexical depth, declaration position) pair, such as $(v : d\ p)$:

```

;; partially converted to lexical addresses,
;; where references are replaced with
;; (identifier, depth, position) triples
> ((lambda (x)
>   (+ 7
>     ((lambda (a b)
>       (+ (a : 1 0)
>          ((lambda (c a)
>            (+ (a : 0 1) (b : 1 1) (x : 2 0))) 3 4))) 1 2))) 5)
19

```

depth:	0		1		2
position:	0	1	0	1	0
environment:	(((c 3) (a 4)) ((a 1) (b 2)) ((x 5)))				

Table 6.3 Lexical Depth and Position in a Referencing Environment

Given only a lexical address (i.e., lexical depth and declaration position), we can (efficiently) lookup the binding associated with the identifier in a reference—a step that is necessary to evaluate the expression containing that reference. Lexically scoped identifiers are useful for writing and understanding programs, but are superfluous and unnecessary for evaluating expressions and executing programs. Therefore, we can purge the identifiers from each lexical address:

```
;; fully converted to lexical addresses,
;; where identifiers are completely purged,
;; references are replaced with (depth, position) pairs.
> ((lambda (x)
>   (+ 7
>     ((lambda (a b)
>       (+ (1 0)
>          ((lambda (c a)
>            (+ (0 1) (1 1) (2 0))) 3 4))) 1 2))) 5)
19
```

With identifiers omitted from the lexical address, the formal parameter lists following each `lambda` are unnecessary and, therefore, can be replaced with their length:

```
;; fully converted to lexical addresses,
;; where identifiers are completely purged,
;; references are replaced with (depth, position) pairs, and
;; formal parameter lists are replaced by their length.
> ((lambda 1
>   (+ 7
>     ((lambda 2
>       (+ (1 0)
>          ((lambda 2
>            (+ (0 1) (1 1) (2 0))) 3 4))) 1 2))) 5)
19
```

Thus, lexical addressing renders variable names and formal parameter lists unnecessary. These progressive layers of translation constitute a mechanical process, which can be automated by a computer program called a *compiler*. A *symbol table* is an instance of an environment often used to associate variable names with lexical address information.

Conceptual Exercises for Section 6.5

Exercise 6.5.1 Consider the following λ -calculus expression:

```
1 (lambda (x y)
2   (lambda (z)
```

```

3      (x (y z)))
4      y))

```

This expression has two lexical depths: 0 and 1. Indicate at which lexical depth each of the four references in this expression resides. Refer to the references by line number.

Exercise 6.5.2 Purge each identifier from the following Scheme expression and replace it with its lexical address. Replace each parameter list with its length. Replace any free variable *v* with (*v* : free).

```

((lambda (x y)
  ((lambda (proc2)
    ((lambda (proc1)
      (cond
        ((zero? (read)) (proc1 5 20))
        (else (proc2))))
      (lambda (x y) (cons x (proc2))))))
  (lambda () (cons x (cons y (cons (+ x y) '()))))))
10 11)

```

Programming Exercise for Section 6.5

Exercise 6.5.3 (Friedman, Wand, and Haynes 2001, Exercise 1.31, p. 37) Consider the subset of Scheme specified by the following EBNF grammar:

```

<expression> ::= <identifier>
<expression> ::= (if <expression><expression><expression>)
<expression> ::= (lambda ({<identifier>}*) <expression>)
<expression> ::= ({<expression>}+)

```

Define a function `lexical-address` that accepts only an expression in this language and returns the expression with each variable reference *v* replaced by a list (*v* : *d p*). If the variable reference *v* is free, produce the list (*v* : free) instead.

Example:

```

1  > (lexical-address '(lambda (x y z)
2                        (if (eqv? y z)
3                            ((lambda (z)
4                              (cons x z))
5                               x)
6                               y)))
7  (lambda (x y z)
8    (if ((eqv? : free) (y : 0 1) (z : 0 2))
9        ((lambda (z)
10          ((cons : free) (x : 1 0) (z : 0 0)))
11         (x : 0 0))
12        (y : 0 1)))

```

6.6 Free or Bound Variables

A variable in an expression in any programming language can appear either (1) bound to a declaration and, therefore, a value, or (2) free, meaning unbound to a declaration and, thus, a denotation or value. The qualification of a variable as free or bound is defined as follows (Friedman, Wand, and Haynes 2001, Definition 1.3.2, p. 29):

- A variable v occurs *free* in an expression e if and only if there is a reference to v within e that is not bound by any declaration of v within e .
- A variable v occurs *bound* in an expression e if and only if there is a reference to v within e that is bound by some declaration of v in e .

For instance, in the expression `((lambda (x) x) y)`, the `x` in the body of the `lambda` expression occurs bound to the declaration of `x` in the formal parameter list, while the argument `y` occurs free because it is unbound by any declaration in this expression. A variable bound in the nearest enclosing λ -expression corresponds to a slot in the current activation record.

A variable may occur free in one context but bound in another enclosing context. For instance, in the expression

```
1 (lambda (y)
2   ((lambda (x) x) y))
```

the reference to `y` on line 2 occurs bound by the declaration of the formal parameter `y` on line 1.

The value of an expression e depends only on the values to which the free variables within the expression e are bound in an expression enclosing e . For instance, the value of the body (line 2) of the `lambda` expression in the preceding example depends only on the denotation of its single free variable `y` on line 1; therefore, the value of `y` comes from the argument to the function. The value of an expression e does not depend on the values bound to variables within the expression e . For instance, the value of the expression `((lambda (x) x) y)` is independent of the denotation of `x` at the time when the entire expression is evaluated. By the time the free occurrence of `x` in the body of `(lambda (x) x)` is evaluated, it is bound to the value associated with `y`.

The semantics of an expression without any free variables is fixed. Consider the *identity function*: `(lambda (x) x)`. It has no free variables and its meaning is always fixed as “return the value that is passed to it.” As another example, consider the following expression:

```
(lambda (x)
  (lambda (f)
    (f x)))
```

A variable v occurs *free* in a λ -calculus expression e if and only if:

- (*symbol*) e is a variable reference and e is the same as v ;
- (*function definition*) e is of the form $(\text{lambda } (y) e')$, where y is different from v and v occurs free in e' ; or
- (*function application*) e is of the form $(e_1 e_2)$ and v occurs free in e_1 or e_2 .

A variable v occurs *bound* in a λ -calculus expression e if and only if:

- (*function definition*) e is of the form $(\text{lambda } (y) e')$, where v occurs bound in e' or v and y are the same variable and y occurs free in e' or
- (*function application*) e is of the form $(e_1 e_2)$ and v occurs bound in e_1 or e_2 .

Table 6.4 Definitions of *Free* and *Bound* Variables in λ -Calculus (Friedman, Wand, and Haynes 2001, Definition 1.3.3, p. 31)

The semantics of this expression, which also has no free variables, is always “a function that accepts a value x and returns ‘a function that accepts a function f and returns the result of applying the function f to the value x .’” Expressions in λ -calculus not containing any free variables are referred to as *combinators*; they include the *identity function* $(\text{lambda } (x) x)$ and the *application combinator* $(\text{lambda } (f) (\text{lambda } (x) (f x)))$, which are helpful programming elements. We saw combinators in Chapter 5 and encounter combinators further in subsequent chapters.

The definitions of free and bound variables given here are general and formulated for any programming language. The definitions shown in Table 6.4 apply specifically to the language of λ -calculus expressions. Notice that the cases of each definition correspond to the three types of λ -calculus expressions, except there is no symbol case in the definition of a bound variable—a variable cannot occur bound in a λ -calculus expression consisting of just a single symbol.

Using these definitions, we can define recursive Scheme functions `occurs-free?` and `occurs-bound?` that each accept a variable `var` and a λ -calculus expression `expr` and return `#t` if `var` occurs free or bound, respectively, in `expr` and `#f` otherwise. These functions, which process expressions, are shown in Listing 6.1. The three cases of the `cond` expression in the definition of each function correspond to the three types of λ -calculus expressions.

The occurrence of the functions `caadr` and `caddr` make these `occurs-free?` and `occurs-bound?` functions unreadable because it is not salient that the

Listing 6.1 Definitions of Scheme functions `occurs-free?` and `occurs-bound?` (Friedman, Wand, and Haynes 2001, Figure 1.1, p. 32).

```
(define occurs-free?
  (lambda (var expr)
    (cond
      ((symbol? expr) (eqv? var expr))
      ((eqv? (car expr) 'lambda)
       (and (not (eqv? (caadr expr) var))
            (occurs-free? var (caddr expr))))
      (else (or (occurs-free? var (car expr))
                 (occurs-free? var (cadr expr)))))))

(define occurs-bound?
  (lambda (var expr)
    (cond
      ((symbol? expr) #f)
      ((eqv? (car expr) 'lambda)
       (or (occurs-bound? var (caddr expr))
           (and (eqv? (caadr expr) var)
                (occurs-free? var (caddr expr)))))
      (else (or (occurs-bound? var (car expr))
                 (occurs-bound? var (cadr expr)))))))
```

former refers to the declaration of a variable in a `lambda` expression or the latter refers to its body. Incorporating abstract data types into our discussion (Chapter 9) makes these functions more readable. Nonetheless, since Scheme is a homoiconic language (i.e., Scheme programs are Scheme lists), Scheme programs can be directly manipulated using standard language facilities (e.g., `car` and `cdr`).

Programming Exercises for Section 6.6

Exercise 6.6.1 (Friedman, Wand, and Haynes 2001, Exercise 1.19, p. 31) Define a function `free-symbols` in Scheme that accepts only a list representing a λ -calculus expression and returns a list representing a set (not a bag) of all the symbols that occur free in the expression.

Examples:

```
> (free-symbols '(a (b (c d))))
'a b c d
> (free-symbols '((lambda (x) x) y))
'y
> (free-symbols '((lambda (x) x) (y z)))
'y z
> (free-symbols '(lambda (f)
  (lambda (x)
    (f x))))
'()
> (free-symbols '(lambda (x)
  ((lambda (y)
```

```

      (lambda (z) (a y)))
    (b x))))
'(a b)
> (free-symbols '(lambda (x)
  ((lambda (y) (c d))
   (lambda (z) (z a)))))
'(c d a)
> (free-symbols '(lambda (x)
  ((lambda (y)
    (lambda (z)
      (a z)))
   (lambda (k)
     (lambda (j)
       (b k)))))))
'(a b)
> (free-symbols '(x x))
'(x)
> (free-symbols '((lambda (x) (x y)) x))
'(y x)
> (free-symbols '(lambda (y) (x x)))
'(x)

```

Exercise 6.6.2 (Friedman, Wand, and Haynes 2001, Exercise 1.19, p. 31) Define a function `bound-symbols` in Scheme that accepts only a list representing a λ -calculus expression and returns a list representing a set (not a bag) of all the symbols that occur bound in the expression.

Examples:

```

> (bound-symbols '(a (b (c d))))
'()
> (bound-symbols '((lambda (x) x) y))
'(x)
> (bound-symbols '((lambda (x) x) (y z)))
'(x)
> (bound-symbols '(lambda (f)
  (lambda (x)
    (f x))))
'(f x)
> (bound-symbols '(lambda (x)
  ((lambda (y)
    (lambda (z) (a y)))
   (b x)))))
'(y x)
> (bound-symbols '(lambda (x)
  ((lambda (y) (c d))
   (lambda (z) (z a)))))
'(z)
> (bound-symbols '(lambda (x)
  ((lambda (y)
    (lambda (z)
      (a z)))
   (lambda (k)
     (lambda (j)
       (b k)))))))
'(z k)

```


6.7 Dynamic Scoping

In a dynamically scoped language, the determination of the declaration to which a reference is bound requires run-time information. In a typical implementation of dynamic scoping, it is the calling sequence of procedures, and not their lexical relationship to each other, that is used to determine the declaration to which each reference is bound. While Scheme uses lexical scoping, for the purpose of demonstration, we use the following Scheme expression to demonstrate dynamic scoping:

```

1  ((lambda (x y)
2    (let ((proc2 (lambda () (cons x (cons y (cons (+ x y) '()))))))
3      (let ((proc1 (lambda (x y) (cons x (proc2)))))
4        (cond
5          ((zero? (read)) (proc1 5 20))
6          (else (proc2)))))
7    10 11)

```

In this expression we see nonlocal references to `x` and `y` in the definition of `proc2` on line 2, which does not provide declarations for `x` and `y`. Therefore, to resolve those references so that we can evaluate the `cons` expression, we must determine to which declarations the references to `x` and `y` are bound.

While static scoping involves a search of the program text, dynamic scoping involves a search of the run-time call stack. Specifically, in a lexically scoped language, determining the declaration to which a reference is bound involves an outward search of the nested blocks enclosing the block where the reference is made. In contrast, making such a determination in a dynamically scoped language involves a downward search from the top of the stack to the bottom.

Due to the invocation of the `read` function on line 5 (which reads and returns an integer from standard input), we are unable to determine the call chain of this program without running it. However, given any two procedures, we can statically determine which has access to the other (i.e., the ability to call) based on the program's lexical layout. Different languages have different rules specifying which procedures have access (i.e., permission to call) to other procedures in the program based on the program's lexical structure. By examining the program text from the preceding example we can determine the *static call graph*, which indicates which procedures have access to each other (Figure 6.2). The *call chain* (or *dynamic call graph*) of an expression depicts the series of functions called by the program as they

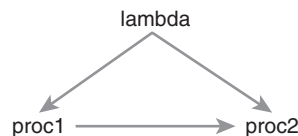


Figure 6.2 Static call graph of the program used to illustrate dynamic scoping in Section 6.7.

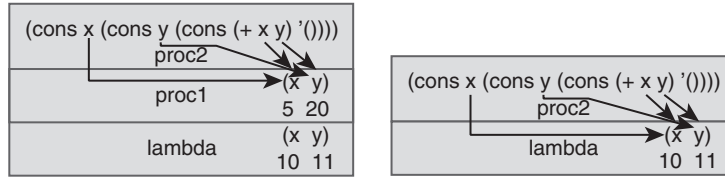


Figure 6.3 The two run-time call stacks possible from the program used to illustrate dynamic scoping in Section 6.7. The stack on the left corresponds to call chain $\text{lambda}^{(x\ y)} \rightarrow \text{proc1}^{(x\ y)} \rightarrow \text{proc2}$. The stack on the right corresponds to call chain $\text{lambda}^{(x\ y)} \rightarrow \text{proc2}$.

would appear on the run-time call stack. From the static call graph in Figure 6.2 we can derive three possible run-time call chains:

$\text{lambda}^{(x\ y)} \rightarrow \text{proc1}^{(x\ y)}$
 $\text{lambda}^{(x\ y)} \rightarrow \text{proc1}^{(x\ y)} \rightarrow \text{proc2}$
 $\text{lambda}^{(x\ y)} \rightarrow \text{proc2}$

Since `proc2` is the function containing the nonlocal references, we only need to consider the two call chains ending in `proc2`. Figure 6.3 depicts the two possible run-time stacks at the time the `cons` expression on line 2 is evaluated (corresponding to these two call chains). The left side of Figure 6.3 shows the stack that results when a 0 is given as run-time input, while the right side shows the stack resulting from a non-zero run-time input.

Since there is no declaration of `x` or `y` in the definition of `proc2`, we must search back through the call chain. When a 0 is input, a backward search of the call chain reveals that the first declarations to `x` and `y` appear in `proc1` (see the left side of Figure 6.3), so the output of the program is (5 5 20 25). When a non-zero integer is input, the same search reveals that the first declarations to `x` and `y` appear in the `lambda` expression (see the right side of Figure 6.3), so the output of the program is (10 11 21).

Shadowed declarations and, thus, scope holes can exist in dynamically scoped programs, too. However, with dynamic scoping, the hole is created not by an intervening declaration (in a block nested within the block containing the shadowed declaration), but rather by an intervening *activation record* (sometimes called a *stack frame* or *environment frame*) on the stack. For instance, when the run-time input to the example program is 0, the declarations of `x` and `y` in `proc1` on line 3 shadow the declarations of `x` and `y` in the `lambda` expression on line 1, creating a scope hole for those declarations in the body of `proc1` as well as any of the functions it or its descendants call.

The *lexical graph* of a program illustrates how the units or blocks of the program are spatially nested, while a *static call graph* indicates which procedures have access to each other. Both can be determined before run-time. The lexical graph is typically a tree, whereas the static call graph is often a non-tree graph. The call chain of a program depicts the series of functions called by the program as they would appear on the run-time call stack and is always linear—that is, a tree

structure where every vertex has exactly one parent and child except for the first vertex, which has no parent, and the last vertex, which has no child. While all possible call chains can be extracted from the static call graph, every process (i.e., program in execution) has only one call graph, but it cannot always be determined before run-time, especially if the execution of the program depends on run-time input.

Do not assume dynamic scoping when the only run-time call chain of a program matches the lexical structure of the nested blocks of that program. For instance, the run-time call chain of the program in Section 6.4.1 mirrors its lexical structure exactly, yet that program uses lexical scoping. When the call chain of a program matches its lexical structure, the declarations to which its references are bound are the same when using either lexical or dynamic scoping. Note that the lexical structure of the nested blocks of the `lambda` expression in the example program containing the call to `read` (i.e., $\text{lambda}^{(x\ y)} \rightarrow \text{proc2} \rightarrow \text{proc1}$) does not match any of its three possible run-time call chains; thus, the resolutions of the nonlocal references (and output of the program) are different using lexical and dynamic scoping.

Similarly, do not assume static scoping when you can determine the call chain and, therefore, resolve the nonlocal references before run-time. Consider the following Scheme expression:

```

1  ((lambda (x y)
2    (let ((proc2 (lambda () (cons x (cons y (cons (+ x y) '()))))))
3      (let ((proc1 (lambda (x y) (cons x (proc2)))))
4        (proc1 5 20))))
5  10 11)

```

The only possible run-time call chain of the preceding expression is $\text{lambda}^{(x\ y)} \rightarrow \text{proc1}^{(x\ y)} \rightarrow \text{proc2}$, even though the static call graph (Figure 6.2) permits more possibilities. Therefore, even if this program uses dynamic scoping, we know *before* run-time that the references to `x` and `y` in `proc2` on line 2 will be bound (at run-time) to the declarations of `x` and `y` in `proc1` on line 3. The program does not use lexical scoping because the nonlocal references on line 2 are bound to declarations nested deeper in the program, rather than being found from an inside-to-out search of its nested blocks from the point of the references. Dynamic scoping is a history-sensitive scoping method, such that the evaluation of nonlocal references depends on where you have been.

6.8 Comparison of Static and Dynamic Scoping

It is important to remember the meaning of *static* and *dynamic* scoping: The declarations to which references are bound are determinable *before* or *at* run-time, respectively. The specifics of how those associations are made before or during run-time (e.g., the lexical structure of the program vis-à-vis the run-time call chain) can vary.

Scoping	Advantages	Disadvantages
Static	improved readability; easier program comprehension; predictability; type checking/validation	larger scopes than necessary; can lead to several globals; can lead to all functions at the same level; harder to implement in languages with nested and first-class procedures
Dynamic	flexibility	reduced readability; reduced reliability; type checking/validation; can be less efficient to implement; difficult to debug; no locality of access; no way to protect local variables; easier to implement in languages with nested and first-class procedures

Table 6.5 Advantages and Disadvantages of Static and Dynamic Scoping

Lexical scoping is a more bounded method of resolving references to declarations than is dynamic scoping. The location of the declaration to which any reference to a lexically scoped variable is bound is limited to the nested blocks surrounding the block containing the reference. By comparison, the location of the declaration to which any reference to a dynamically scoped variable is bound is less restricted. Such a declaration can exist in any procedure in the program that has access to call the procedure containing the reference, and the procedures that have access to that one, and so on. This renders dynamic scoping more flexible—typical of any dynamic feature or concept—than static scoping. The rules governing which procedures a particular procedure can call are typically based on the program’s lexical layout. For instance, if a procedure g is nested within a procedure f , and a procedure y is nested within a procedure x , then f can call g and x can call y , but x cannot call g and f cannot call y . The process is more globally distributed through a program. Table 6.5 compares the advantages and disadvantages of static and dynamic scoping. We implement lexical and dynamic scoping in interpreters in Chapter 11.

Conceptual and Programming Exercises for Section 6.8

Exercise 6.8.1 Evaluate the following Scheme expression:

```
(let ((a 1))
  (let ((a (+ a 2)))
    a))
```

Exercise 6.8.2 Can the Scheme expression from Conceptual Exercise 6.8.1 be rewritten with only `let*`? Explain.

Exercise 6.8.3 Consider the following two C++ programs:

```

1  #include<iostream>
2  using namespace std;
3
4  int a = 10;
5
6  int main() {
7      int a = a + 2;
8      cout << a << endl;
9  }

1  #include<iostream>
2  using namespace std;
3
4  int main() {
5      int a = 10; {
6          int a = a + 2;
7          cout << a << endl;
8      }
9  }
```

Does the reference to `a` on the right-hand side of the assignment operator on line 7 of the first program bind to the declaration of the global variable `a` on line 4? Similarly, does the reference to `a` on line 6 of the second program bind to the local variable `a` declared on line 5? Run these programs. What can you infer about how C++ addresses scope based on the outputs?

Exercise 6.8.4 Consider the Java expression `int x = x + 1;`. Determine where the scope of `x` begins. In other words, is the `x` on the right-hand side of the assignment from another scope or does it refer to the `x` being declared on the left-hand side? Alternatively, is this expression even valid in Java? Explain.

Exercise 6.8.5 Consider the following C program:

```

1  int x;
2
3  void p (void) {
4      char x;
5      x = 'a'; /* assigns to char x */
6  }
7
8  int main() {
9      x = 2; /* assigns to global x */
10 }
```

Using *static scoping*, the declaration of `x` in `p` (line 4) takes precedence over the global declaration of `x` (line 1) in the body of `p`. Thus, the global integer `x` cannot be accessed from within the procedure `p`. The global declaration of `x` has a scope hole inside of `p`.

In C++, can you access the `x` declared in line 1 from the body of `p`? If so, how?

Exercise 6.8.6 Recall that Scheme uses *static* scoping.

(a) Consider the following Scheme code:

```

(define i 0)

(define f
  (lambda (lat)
    (cond
      ((null? lat) (cons i '()))
      (else (let ((i (+ i 1)))
              (let ((i (+ i 1)))
                (cons i (f (cdr lat))))))))))

```

What is the result of `(f ' (a b c))`?

(b) Consider the following Scheme code:

```

(define i 0)

(define g
  (let ((i (+ i 1)))
    (let ((i (+ i 1)))
      (lambda (lat)
        (cond
          ((null? lat) (cons i '()))
          (else (cons i (g (cdr lat))))))))))

```

What is the result of `(g ' (a b c))`?

Exercise 6.8.7 Consider the following skeletal program written in a block-structured programming language:

```

1  program main;
2    var x: integer;
3
4    procedure p1;
5
6      var x: real;
7
8      procedure p2; begin
9        ...
10     end;
11
12     begin
13       ...
14     end;
15
16     procedure p3; begin
17       write(x);
18     end;
19
20   begin
21     ...
22   end.

```

- (a) If this language uses *static scoping*, what is the type of the variable `x` printed on line 17 in procedure `p3`?
- (b) If this language uses *dynamic scoping*, what is the type of the variable `x` printed on line 17 in procedure `p3`?

Exercise 6.8.8 Consider the following Scheme program:

```

1  > (define f
2      (lambda (f)
3          (map
4              (lambda (f)
5                  (* f 2))
6                  f)))
7  > (f '(2 4 6))
8  (4 8 12)

```

- (a) Annotate lines 5–7 of this program with comments indicating to which declaration of `f` on lines 1, 2, and 4 the references to `f` on lines 5–7 are bound.
- (b) Annotate lines 1, 2, and 4 of this program with comments indicating, with line numbers, the scope of the declarations of `f` on lines 1, 2, and 4.

Exercise 6.8.9 In C++, is a variable declared in a `for` loop [e.g., `for (int i = 0; i < 10; i++)`] visible after the loop terminates when not declared in an outer scope? How about those in other program blocks, such as `if`, `while`, or a block in general (i.e., created with `{` and `}`)?

Exercise 6.8.10 Evaluate the Scheme expression in the last paragraph of Section 6.7 using lexical scoping.

Exercise 6.8.11 Explain the evaluation of the following Scheme code:

```

(define x 10)
(define y 11)

(define proc2
  (lambda ()
    (cons x (cons y '()))))

(define proc1
  (lambda (x y)
    (proc2)))

(cond
  ((zero? (read)) (proc1 5 20))
  (else (proc2)))

```

Exercise 6.8.12 Explain the evaluation of the following Scheme code:

```

(define x 10)
(define y 11)

(define proc2
  (lambda ()
    (cons x (cons y '()))))

(define proc1
  (lambda (x y)
    (proc2)))

```

```
(define main
  (lambda ()
    (cond
      ((zero? (read)) (proc1 5 20))
      (else (proc2))))
  (main))
```

Exercise 6.8.13 Can a programming language that uses *dynamic scoping* also use *static type checking*? Explain.

Exercise 6.8.14 Can a programming language that uses *dynamic type checking* also use *static scoping*? Explain.

Exercise 6.8.15 Consider the following Scheme expression:

```
;; mutually recursive iseven? and isodd? functions
(let*
  ((iseven? (lambda (x) (if (zero? x) #t (isodd? (- x 1)))))
   (isodd? (lambda (x) (if (zero? x) #f (iseven? (- x 1)))))
  (isodd? 15))
```

Will this expression return 1 when evaluated under *dynamic scoping*, even in the absence of a `letrec` expression? Explain.

Exercise 6.8.16 Write a Scheme program that outputs different results when run using lexical scoping and dynamic scoping.

6.9 Mixing Lexically and Dynamically Scoped Variables

Dynamic scoping was used in McCarthy's original version of Lisp (more on this in Section 6.10) as well as in APL and SNOBOL4. Scheme, a dialect of Lisp, adopted static scoping.² Some languages, including Perl and Common Lisp, leave the scoping method up to the programmer. However, Perl gives the programmer a finer level of control over scope. Control of scope is done on the variable level, rather than the program level. Instead of setting the method of scoping for the entire program, Perl enables the programmer to fine-tune which variables are statically scoped and which are dynamically scoped. The provisions for scope in Common Lisp are similar. Qualifiers, including `private`, `public`, and `friend`, and operators, including the scope resolution operator `::`, give the programmer a finer control over the scope of a declaration in C++.

We use the Perl program in Listing 6.2 to demonstrate this mixture. The main program (i.e., the program code before the definitions of procedures `proc1` and `proc2`) prints the values of two variables (`l` and `d`) to standard output, calls `proc1`, increments `l` and `d`, and again prints the values of two variables (`l` and `d`) to standard output. In the definition of `proc1`, the `my` qualifier on the declaration

2. This is an example of mutation in the evolution of programming languages.

Listing 6.2 A Perl program demonstrating dynamic scoping.

```

1  $l = 10;
2  $d = 11;
3
4  print "Before the call to proc1 --- l: $l, d: $d\n";
5
6  &proc1(); # call to proc1
7
8  $l++;
9  $d++;
10
11 print "After the call to proc1 --- l: $l, d: $d\n";
12
13 exit(0);
14
15 sub proc1 {
16
17     # keyword "my" makes l a lexically scoped variable,
18     # meaning it is accessible only in this block
19     # and any nested blocks herein
20     my $l;
21
22     # keyword "local" makes d a dynamically scoped variable,
23     # meaning it is accessible to its descendants in the call chain
24     local $d;
25
26     $l = 5;
27     $d = 20;
28
29     print "Inside the call to proc1 --- l: $l, d: $d\n";
30
31     &proc2(); # call to proc2
32
33     print "After the call to proc2 --- l: $l, d: $d\n";
34 }
35
36 sub proc2 {
37     print "Inside the call to proc2 --- l: $l, d: $d\n";
38 }

```

of the variable `l` specifies that `l` is a lexically scoped variable. This means that any reference to `l` in `proc1` or any blocks nested therein follow the lexical scoping rule given previously, unless there is an intervening declaration of `l`. The `local` qualifier on the declaration of the variable `d` specifies that `d` is a dynamically scoped variable. This means that any reference to `d` in `proc1` or any procedure called from `proc1` or called from that procedure, and so on, is bound to this declaration of `d` unless there is an intervening declaration of `d`. Thus, the first two lines of program output are

```

Before the call to proc1 --- l: 10, d: 11
Inside the call to proc1 --- l: 5, d: 20

```

We see a reference to `l` and `d` on line 37 in the definition of `proc2`, which does not provide declarations for `l` and `d`. To resolve those references so that we

procedure names	activation records	variables
proc2		
proc1	20	d
	5	l
main	11	d
	10	l

Figure 6.4 Depiction of run-time stack at call to `print` on line 37 of Listing 6.2.

can evaluate the `print` statement, we must determine to which declarations the references to `l` and `d` are bound.

Examining this program, we see that the only possible run-time call sequence of procedures is `main` $\rightarrow$ `proc1` $\rightarrow$ `proc2`. Figure 6.4 depicts the run-time stack at the time of the `print` statement on line 37. While static scoping involves a search of the program text, dynamic scoping involves a search of the run-time stack. Specifically, while determining the declaration to which a reference is bound in a lexically scoped language involves an outward search of the nested blocks enclosing the block where the reference is made, doing the same in a dynamically scoped language involves a downward search from the top of the stack to the bottom. Using the approach outlined in Section 6.4.1 for determining the declaration associated with a reference to a lexically scoped variable, we discover that the reference to `l` on line 37 is bound to the declaration of `l` on line 1. Since `d` is a dynamically scoped variable and `d` is not declared in the definition of `proc2`, we must search back through the call chain. Examining the definition of the procedure that called `proc2` (i.e., `proc1`), we find a declaration for `d`. Thus, our search is complete and we use the denotation of `d` in `proc1` at the time `proc2` is called: 20. Therefore, `proc2` prints

```
Inside the call to proc2 --- l: 10, d: 20
```

The output of the entire program is

```
Before the call to proc1 --- l: 10, d: 11
Inside the call to proc1 --- l: 5, d: 20
Inside the call to proc2 --- l: 10, d: 20
After the call to proc2 --- l: 5, d: 20
After the call to proc1 --- l: 11, d: 12
```

Dynamic scoping means that the declaration to which a reference is bound cannot be determined until run-time. However, in Listing 6.2, we need not run the program to determine to which declaration the reference to `d` on line 37 is bound. Even though the variable `d` is dynamically scoped, we can determine the call chain of the procedures, before run-time, by examining the text of the program. However, in most programs we cannot determine the call chain of procedure before run-time—primarily due to the presence of run-time input.

Listing 6.3 A Perl program, whose run-time call chain depends on its input, demonstrating dynamic scoping.

```

1  $l = 10;
2  $d = 11;
3
4  # reads an integer from standard input
5  $input = <STDIN>;
6
7  print "Before the call to proc1 --- l: $l, d: $d\n";
8
9  if ($input == 5) {
10     &proc1(); # call to proc1
11  } else {
12     &proc2(); # call to proc2
13  }
14
15  $l++;
16  $d++;
17
18  print "After the call to proc1 --- l: $l, d: $d\n";
19
20  exit(0);
21
22  sub proc1 {
23
24     # keyword "my" makes l a lexically scoped variable,
25     # meaning it is accessible only in this block
26     # and any nested blocks herein
27     my $l;
28
29     # keyword "local" makes d a dynamically scoped variable,
30     # meaning it is accessible to its descendants in the call chain
31     local $d;
32
33     $l = 5;
34     $d = 20;
35
36     print "Inside the call to proc1 --- l: $l, d: $d\n";
37
38     &proc2(); # call to proc2
39
40     print "After the call to proc2 --- l: $l, d: $d\n";
41  }
42
43  sub proc2 {
44     print "Inside the call to proc2 --- l: $l, d: $d\n";
45  }

```

Consider the Perl program in Listing 6.3, which is similar to Listing 6.2 except that the call chain depends on program input. If the input is 5, then the call chain is

main → proc1 → proc2

and the output is the same as the output for Listing 6.2. Otherwise, the call chain is

main → proc2

and the output is

```
Before the call to proc1 --- l: 10, d: 11
Inside the call to proc2 --- l: 10, d: 11
After the call to proc1 --- l: 11, d: 11
```

As we can see, just because we can determine the declaration to which a reference is bound before run-time in a particular program, that does not mean that the language in which the program is written uses static scoping.

Listings 6.2 and 6.3 contain both shadowed lexical and dynamic declarations and, therefore, lexical and dynamic scope holes, respectively. For instance, the declaration of `l` on line 20 in Listing 6.2 shadows the declaration of `l` on line 1. Furthermore, the declaration of `d` on line 24 in Listing 6.2 shadows the declaration of `d` on line 2, creating a scope hole in the definition of `proc1` as well as any of the functions it or its descendants (on the stack) call. In other words, the shadow is cast into `proc2`. In contrast, the declaration of `l` on line 20 in Listing 6.2 does not create scope holes in any descendant procedures.

Conceptual and Programming Exercises for Section 6.9

Exercise 6.9.1 Identify all of the scope holes on lines 4, 11, 29, 33, and 37 of Listing 6.2. For each of those lines, state which declarations create shadows and indicate the declarations they obscure.

Exercise 6.9.2 Identify all of the scope holes on lines 7, 18, 36, 40, 44 of Listing 6.3. For each of those lines, state which declarations create shadows and indicate the declarations they obscure.

Exercise 6.9.3 Sketch a graph depicting the lexical structure of the procedures (i.e., a lexical graph), including the body of the main program, in Listing 6.2.

Exercise 6.9.4 Sketch the static call graph for Listing 6.2. Is the static call graph for Listing 6.3 the same? If not, give the static call graph for Listing 6.3 as well.

Exercise 6.9.5 Consider the following Scheme, C, and Perl programs, which are analogs of each other:

```
1  ((lambda (x y)
2    (let ((proc1 (lambda (x) (+ x 1)))
3          (proc2 (lambda (y) (* y 2))))
4      (proc2 (* y (proc1 x))))) 2 3)

1  int main() {
2    int x=2;
3    int y=3;
4
5    int proc1 (int x) {
6      return x+1;
7    }
8    int proc2 (int y) {
9      return y*2;
10 }
```

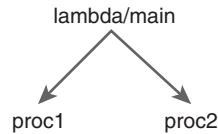
```

11
12     return proc2(proc1(x)*y);
13 }

1  sub main() {
2      $x=2;
3      $y=3;
4
5      sub proc1 {
6          return $_[0]+1;
7      }
8
9      sub proc2 {
10         return $_[0]*2;
11     }
12     print proc2(proc1($x)*$y);
13 }
14 main;

```

The following graph depicts the lexical structure of these three programs (i.e., a lexical graph):



The rules in Scheme, C, and Perl that specify which procedures have access to call other procedures are different. Therefore, while each program has the same lexical structure, they may not have the same static call graph.

Sketch the static call graph for each of these programs.

Exercise 6.9.6 Consider the following Scheme, C, and Perl programs, which are analogs of each other:

```

1  (define x 2)
2  (define y 3)
3  (define proc1 (lambda (x) (+ x 1)))
4  (define proc2 (lambda (y) (* y 2)))
5  (proc2 (* y (proc1 x)))

1  int proc1 (int x) {
2      return x+1;
3  }
4
5  int proc2 (int y) {
6      return y*2;
7  }
8
9  int main() {
10     int x=2;
11     int y=3;
12
13     return proc2(proc1(x)*y);
14 }

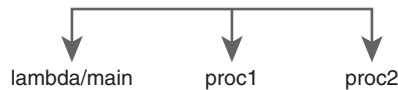
```

```

1  $x=2;
2  $y=3;
3
4  print proc2(proc1($x)*$y);
5
6  sub proc1 {
7      return $_[0]+1;
8  }
9
10 sub proc2 {
11     return $_[0]*2;
12 }

```

The following graph depicts the lexical structure of these three programs (i.e., a lexical graph):



The rules in Scheme, C, and Perl that specify which procedures have access to call other procedures are different. Therefore, while each program has the same lexical structure, they may not have the same static call graph.

Sketch the static call graph for each of these programs.

Exercise 6.9.7 Does line 4 [i.e., `print proc2 (proc1($x)*$y);`] of the Perl program in Exercise 6.9.6 demonstrate that Perl supports first-class functions? Explain why or why not.

Exercise 6.9.8 Common Lisp, like Perl, allows the programmer to declare statically or dynamically scoped variables. Figure out how to set the scoping method of a variable in a Common Lisp program and write a Common Lisp program that illustrates the difference between static and dynamic scoping, similarly to the Perl programs in this section. (Do not replicate that program in Common Lisp.) Use GNU CLISP implementation of Common Lisp, available at <https://clisp.sourceforge.io/>. Writing a program that only demonstrates how to set the scoping method in Common Lisp is insufficient.

6.10 The FUNARG Problem

The concept of scope is only relevant in the presence of nonlocal references. Dynamic scoping is easier to implement than lexical scoping since it simply requires a downward search of the run-time call stack. Lexical scoping, by contrast, requires a search of the lexical graph of the program, which is typically tree structured. *In either case, the activation record containing the declaration to which a nonlocal reference is bound will always be on the run-time stack, even though it may not be found in the record immediately beneath the record for the procedure*

containing the nonlocal reference. McCarthy's first version of Lisp used dynamic scoping, though this was unintentional. This is an instance of a programming language being historically designed based on ease of implementation rather than the abilities of programmers.

When we include first-class procedures in the discussion of scope, the issue of resolving nonlocal references suddenly becomes more complex, particularly with respect to implementation. The issue of determining the declaration to which a reference is bound is more interesting in languages with first-class procedures implemented using a run-time stack. The question is: To which declaration does a reference in the body of a passed or returned function bind? The difficulty of implementing first-class procedures in a stack-based programming language is dubbed the FUNARG (FUNCTIONAL ARGUMENT) problem. The FUNARG problem helps to illustrate the relationship between scope and closures and ties together multiple concepts related to scope (within the context of broader themes and history). Moreover, this discussion provides background for more implementation-oriented issues addressed elsewhere in this text.

The difficulty arises when a nested function makes a nonlocal reference (i.e., a reference to an identifier not representing a parameter) to an identifier in the environment in which the function is defined, but not invoked. In such a case, we must determine the environment in which to resolve that reference so that we can evaluate the body of the function. *The problem is that the environment in which the function is created may not be on the stack.* In other words, what do we do when a function refers to something that may not be currently executing (i.e., not on the run-time stack)? There are two instances of the FUNARG problem: the downward FUNARG problem and the upward FUNARG problem.

6.10.1 The Downward FUNARG Problem

The *downward FUNARG problem* involves passing a function (called a *downward FUNARG*) to another function. It is generally regarded as the easier of the two to solve primarily because the environment in which the passed function is created can be stored (in activation records) on the stack. Due to the lexical nesting of a program, the activation records are structured as a tree rather than a stack. However, this has the disadvantage of rendering human reasoning about the state of a program more complex. Consider the following Scheme expression:

```

1  ((lambda (x y)
2    ((lambda (proc2)
3      ((lambda (proc1) (proc1 5 20))
4        (lambda (x y) (cons x (proc2))))))
5    (lambda () (cons x (cons y (cons (+ x y) '())))))
6  10 11)
```

The functions passed on lines 4 and 5, and accessed through the parameters `proc1` and `proc2`, respectively, are downward FUNARGS.

6.10.2 The Upward FUNARG Problem

The *upward FUNARG problem* involves returning a function (called an *upward FUNARG*) from a function, rather than passing functions to a function. It is more difficult to solve than the downward version of the problem. Consider the following classical example of an upward FUNARG in Scheme:

```

1  (define add_x
2    (lambda (x)
3      (lambda (y)
4        (+ x y))))
5
6  (define main
7    (lambda ()
8      (let ((add5 (add_x 5)))
9        (add6 (add_x 6))))
10     (cons (add5 2) (cons (add6 2) '()))))
11  (main)

```

The function `add_x` returns a closure (lines 3–4), which adds its argument (i.e., `y`) to the argument to `add_x` (i.e., `x`) and returns the result. The `add_x` function provides the simplest nontrivial example of a closure. The `add_x` function creates (and returns) a closure around the inner function.

The left side of Figure 6.5 illustrates the upward FUNARG problem by depicting the run-time stack after `add_x` is called, but before it returns to `main` (line 8). The right side of Figure 6.5 depicts the run-time stack after `add_x` returns to `main` (line 9). As seen in this figure, the function returned by the `(add_x 5)` call to `add_x` is `(lambda (y) (+ x y))`, which appears to have no free variables. The reference to `y` is bound to the declaration of `y` in the inner `lambda` expression; the reference to `x` is bound to the declaration of `x` in the outer `lambda` expression. However, once `add_x` returns the function `(lambda (y) (+ x y))`, its activation record is popped off the stack and destroyed. Therefore, the binding of `x` to 5 no longer exists; moreover, the `x` itself no longer exists. In other words, a closure can outlive its lexical parent. A *closure*

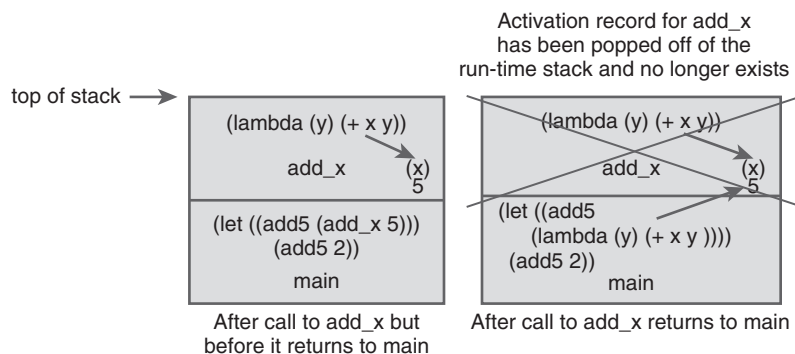


Figure 6.5 Illustration of the upward FUNARG problem highlighting a reference to a declaration in a nonexistent activation record.

Name of Closure	Closure	
	expression	environment
add5	(lambda (y) (+ x y))	(x 5)
add6	(lambda (y) (+ x y))	(x 6)

Table 6.6 Example Data Structure Representation of Closures

is a function created by a program at run-time that remembers the environment at the time it is created and uses it to provide the bindings for the free variables it contains. Thus, the returned function contains references to data which no longer exists on the stack. This is the essence of the FUNARG problem—how to implement first-class functions in a stack-based language.

The return value of `add_x` is a closure—that is, a function created by a program at run-time that refers to free variables in its lexical context. The free variable in this case is `x`. The question is: From where does `x` derive its value? We can think of a closure as a package of two pointers: one to an *expression* [e.g., `(lambda (y) (+ x y))`] and one to an *environment* [e.g., `(x 5)`]. When the expression is evaluated, bindings for the free variables within that expression come from the environment. Therefore, we can think of a closure as encapsulating both *behavior* (i.e., code) and local *state* (i.e., environment), much like an object in object-oriented languages. Since a closure can be passed to a function, returned from a function, or stored in a variable, closures are first-class values. Closures can be used as building blocks for creating language abstractions (e.g., currying and lazy evaluation; see Chapters 8 and 12, respectively). They are often used as arguments to higher-order functions (e.g., `map`), to hide state and/or to delay computation (i.e., lazy evaluation; see Section 12.5). Closures are a functional analog of objects in object-oriented languages.

Table 6.6 presents one data structure that an interpreter might potentially use to internally represent these closures. As can be seen, each closure stores a copy of the value for `x` *at the time of their creation*, since `x` was in the environment of the lexically nested outer scope in which each was defined—a scope that is no longer available at the time each is invoked. Even though only one instance of `x` is found in the environment of the outer scope, these two closures never confuse that `x` because each has its own copy. In that sense, closures are like objects: both encapsulate code (known as *methods* in object-oriented nomenclature) with local state (known as *instance variables*). The preceding example works in Scheme and returns `(7 8)` because Scheme addresses the FUNARG problem.

Replicating this example in a language that does not support first-class closures with *indefinite extent* or *unlimited extent* (i.e., indefinite lifetime), including C, makes the FUNARG problem more salient because in such languages it is unaddressed and, therefore, unsolved. Consider the following C program, which is the analog of the example Scheme program:

```

1  #include<stdio.h>
2
3  /* f is a function that accepts an int x
4     as an argument and returns a pointer
5     to a function that accepts an int as an
6     argument and returns an int as a return value. */
7  int (*f(int x)) (int) {
8
9      int g(int y) {
10         return (x+y);
11     }
12
13     /* return a pointer to the function g */
14     return &g;
15 }
16
17 int main() {
18     /* add5 is a pointer to a function that
19        accepts an int as an argument and returns
20        an int as a return value. */
21     int (*add5) (int) = f(5);
22     int (*add6) (int) = f(6);
23
24     printf("%d\n", add5(2));
25     printf("%d\n", add6(2));
26 }

```

Although C does not support first-class procedures directly, we can simulate such procedures by assigning, passing, or returning a pointer to a function. (Essentially the effect is the same as that seen in Scheme since everything is a pointer in Scheme and Scheme uses implicit pointer dereferencing.) The preceding C program contains a function `f` that accepts an `int x` as an argument and returns a pointer to a function that accepts an `int` as an argument and returns an `int` as a return value. Rather than using a `void*` generic pointer (i.e., an untyped pointer—a pointer to a memory address of data of any type), the declarations of the pointers (lines 7, 21, and 22) explicitly indicate to which type of data they point. For example, `int* x` indicates that `x` is declared to be a pointer to an integer type. In declaring these pointer types, we are declaring the type of a function.

Declaring the type of a function may be unfamiliar to readers with a background in imperative, or other, languages that do not support first-class procedures. Generally, we declare the type of data using primitive (e.g., `int x`, `float rate`) or user-defined types (e.g., `stack items`). However, in languages where functions are first-class entities, those functions, just like any other data object, have types. Declaring types of functions is common practice in ML and Haskell, where types are a primary focus. The idea that functions also have types is discussed further in Chapters 7 and 8. We can describe the type of a function with an expression in the form $a \rightarrow b$, where a is the type of the domain of the function and b is the type of the range of the function. While a discussion of the syntactic details of declaration of pointer types in C is beyond the scope of this chapter, in the example C program the identifier `add6` is declared to be a pointer to a function with type `int  $\rightarrow$  int` (line 21). The function `f` is declared to have type `int  $\rightarrow$  (int  $\rightarrow$  int)` (line 7).

C, which is a stack-based programming language, does not address the FUNARG problem. Thus, the example C program produces erroneous output:³

```
$ gcc makeadder.c
$ ./a.out
8
8
```

The program is trying to access a stack frame that is no longer guaranteed to exist.

Another way of saying that C does not address the FUNARG problem is to say that it does not support first-class closures. Given the presence of function pointers in C, it is more accurate to say that C does have first-class procedures, but does not have first-class closures and, therefore, does not solve the FUNARG problem. Trying to simulate first-class closures in a language without direct support for them is an arduous task.

Python supports both first-class procedures and first-class closures. The following program is the Python analog of the Scheme program given at the beginning of this subsection:

```
>>> def f(x):
...     return lambda y: x+y
...
>>> add5 = f(5)
>>> add6 = f(6)
>>>
>>> add5
<function f.<locals>.<lambda> at 0x10df9e700>
>>> add6
<function f.<locals>.<lambda> at 0x10df9e790>
>>>
>>> add5(2)
7
>>> add6(2)
8
```

We use the following classical example of closures in Scheme as another demonstration of both the upward FUNARG problem and the analogy between closures and objects:

```
1 (define new_counter
2   (lambda ()
3     (let ((current 0))
4       (lambda ()
5         (set! current (+ current 1))
6         current))))
```

Lines 4–6 represent a closure. The `set!` function is the assignment operator in Scheme. Although there is only one declaration of the variable `current` in the

3. The output of C programs like this are highly compiler- and system-dependent and may generate a fatal run-time error rather than producing erroneous output. Moreover, you may need to compile such programs with the `-fnested-functions` option to `gcc` (e.g., `gcc -fnested-functions makeadder.c`).

program (line 3), the two counter closures each have their own copy of it and, therefore, are independent:

```

1  > (define counter1 (new_counter))
2  > (define counter2 (new_counter))
3  >
4  > (counter1)
5  1
6  > (counter1)
7  2
8  > (counter2)
9  1
10 > (counter2)
11 2
12 > (counter1)
13 3
14 > (counter1)
15 4
16 > (counter2)
17 3

```

As a result, the counters never get mixed up (lines 4–17). The binding to data popped off the stack (e.g., `current`) still exists.

The `new_counter` function resembles a constructor—it constructs new counters (i.e., objects). Often constructors are parameterized so that the constructed objects are created with a user-specified state rather than a default state. For example, they might set the maximum number of items in a queue object to be 11 rather than the default of 10. Here, we can parameterize the constructor so that the counter created is initialized to a user-specified value rather than 0:

```

(define new_counter
  (lambda (initial)
    (let ((current initial))
      (lambda ()
        (set! current (+ current 1))
        current)))))

> (define counter1 (new_counter 1))
> (define counter2 (new_counter 100))
>
> (counter1)
2
> (counter1)
3
> (counter2)
101
> (counter2)
102
> (counter1)
4
> (counter1)
5
> (counter2)
103

```

This example makes the analogy between closures and objects stronger: In addition to packaging behavior and state, these closures hide and protect

the counter value `current`, as is done by prefacing an instance variable with a qualifier (e.g., `private`) in some languages supporting object-oriented programming. This behavior demonstrates information hiding—another concept from software engineering adopted by object-oriented programming. Closures and objects share the following similarities, as highlighted by this example:

- encapsulation of behavior and state
- information hiding
- arbitrary construction (i.e., creation) at the programmer's discretion (e.g., `new_counter`)
- existence of each in a separate memory space

Again, the analogous C program does not work because once the `new_counter` function returns and is popped off the stack, the local variable `current` no longer exists:

```
#include<stdio.h>

int (*new_counter(int initial)) () {
    int current = initial;

    int increment () {
        current++;
        return current;
    }

    return &increment;
}

int main() {
    int (*counter1) () = new_counter (1);
    int (*counter2) () = new_counter (100);

    printf ("%d\n", counter1());
    printf ("%d\n", counter2());
    printf ("%d\n", counter1());
    printf ("%d\n", counter1());
}

$ gcc makecounter.c
$ ./a.out
101
102
103
104
```

Counters in different lexical spaces are helpful constructs in a variety of programming tasks. Achieving the same outcome in a language without first-class closures (e.g., C) is an arduous task.

In the absence of first-class closures in C, the keyword `static`, when used to give a variable local to a function static (or global) storage, is a way (albeit ad hoc) to impart context or state to a function after it has popped off the stack. In other words, it allows us to save state between multiple calls to the same function. This approach is ad hoc in the sense that the environment attached to the function is a global (not local) environment that does not foster a true closure:

```

#include<stdio.h>

int increment1() {
    static int current = 1;
    return ++current;
}

int increment100() {
    static int current = 100;
    return ++current;
}

int main() {
    printf ("%d\n", increment1());
    printf ("%d\n", increment100());
    printf ("%d\n", increment1());
    printf ("%d\n", increment100());
    printf ("%d\n", increment1());
    printf ("%d\n", increment100());
    printf ("%d\n", increment1());
    printf ("%d\n", increment100());
}

$ gcc increment.c
$ ./a.out
2
101
3
102
4
103
5
104

```

Notice that even though the variable `current` has static storage, each function has its own global variable `current`, with the same name. Thus, the functions maintain separate counters. This approach is not much different from the following:

```

#include<stdio.h>

int counter1 = 1;
int counter100 = 100;

int increment1() {
    return ++counter1;
}

int increment100() {
    return ++counter100;
}

int main() {
    printf ("%d\n", increment1());
    printf ("%d\n", increment100());
    printf ("%d\n", increment1());
    printf ("%d\n", increment100());
    printf ("%d\n", increment1());
    printf ("%d\n", increment100());
    printf ("%d\n", increment1());
    printf ("%d\n", increment100());
}

```

```

$ gcc increment2.c
$ ./a.out
2
101
3
102
4
103
5
104

```

Since Python supports first-class functions and first-class closures, as well as implicit typing, we can use it to study these concepts:

```

1 >>> def new_counter(initial):
2 ...     current = initial
3 ...     def increment():
4 ...         current = current + 1
5 ...         return current
6 ...     return increment
7 ...
8 >>> counter1 = new_counter(1)
9 >>> counter2 = new_counter(100)
10 >>>
11 >>> counter1
12 <function new_counter.<locals>.increment at 0x1038d5700>
13 >>> counter2
14 <function new_counter.<locals>.increment at 0x1038d5790>
15 >>>
16 >>> counter1()
17 Traceback (most recent call last):
18   File "<stdin>", line 1, in <module>
19   File "<stdin>", line 4, in increment
20 UnboundLocalError: local variable 'current' referenced before
21 assignment

```

The reason the call to `counter1()` on line 16 does not run is not related to the FUNARG problem because Python addresses the FUNARG problem. Instead, the interference arises from implicit typing. In this example, we want `current` on the left-hand side of the assignment operator in line 4 to be interpreted as a reference bound to the declaration of `current` in line 2, rather than as a new declaration of a variable with the same name. While `current` on the right-hand side of the assignment operator in line 4 is a reference, the Python interpreter thinks it is a reference to a variable that has yet to be assigned a value. In this case, as with other languages that use implicit typing, it is unclear whether `current` on the left-hand side of the assignment operator in line 4 is intended as a reference or a declaration.

To force the semantics we want upon this program, so that the `current` in the definition of `increment` refers to the declaration of `current` in the enclosing `new_counter` function, we wrap `current` in a list:

```

1 >>> def new_counter(initial):
2 ...     # makes current a list
3 ...     current = [initial]
4 ...     def increment():
5 ...         # makes it unambiguous that the current we are referencing

```

```

6 ...      # is bound to the declaration of current because the
7 ...      # brackets are used for referencing, not declaration
8 ...      current[0] = current[0] + 1
9 ...      return current[0]
10 ...     return increment
11 ...
12 >>> counter1 = new_counter(1)
13 >>> counter2 = new_counter(100)
14 >>>
15 >>> counter1
16 <function new_counter.<locals>.increment at 0x10bc74700>
17 >>> counter2
18 <function new_counter.<locals>.increment at 0x10bc74790>
19 >>>
20 >>> counter1()
21 2
22 >>> counter1()
23 3
24 >>> counter2()
25 101
26 >>> counter2()
27 102
28 >>> counter1()
29 4
30 >>> counter1()
31 5
32 >>> counter2()
33 103

```

Wrapping the initial value in a list of one element named `current` has the convenient side effect of making the intended semantics unambiguous. The occurrence of `current` using list bracket notation on the left-hand side of the assignment operator (line 8) is a reference bound to the list `current` declared in the enclosed scope (i.e., the definition of `new_counter` function) rather than a new intervening declaration. (We return to the concept of implicit/manifest typing in Chapter 7.) Also notice here that we use a named (i.e., `def`) rather than anonymous (i.e., `lambda`) function.

The first-class function returned in this program (`increment`) is bound to the environment in which it is created. In object-oriented programming, an object encapsulates *multiple* functions (called methods) and *one* environment. In other words, an object binds *multiple* functions to the *same* environment. The same effect can be achieved with first-class closures by returning a list of closures:

```

>>> def new_counter(initial):
...     current = [initial]
...     def initialize(initial):
...         current[0] = initial
...     def increment():
...         current[0] = current[0] + 1
...     def decrement():
...         current[0] = current[0] - 1
...     def get():
...         return current[0]
...     def write():
...         print (current[0])
...     return [initialize, increment, decrement, get, write]
...

```



```

>>> counter1 = new_counter(1)
>>> initialize1 = counter1[0]
>>> increment1 = counter1[1]
>>> decrement1 = counter1[2]
>>> get1 = counter1[3]
>>> write1 = counter1[4]
>>>
>>> counter2 = new_counter(100)
>>> initialize2 = counter2[0]
>>> increment2 = counter2[1]
>>> decrement2 = counter2[2]
>>> get2 = counter2[3]
>>> write2 = counter2[4]
>>>
>>> increment1()
>>> increment2()
>>> increment1()
>>> increment2()
>>> increment1()
>>> increment2()
>>> write1()
4
>>> write2()
103
>>> decrement1()
>>> decrement2()
>>> decrement1()
>>> decrement2()
>>> decrement1()
>>> decrement2()
>>> write1()
1
>>> write2()
100

```

6.10.3 Relationship Between Closures and Scope

Programming language terms and concepts have evolved with programming languages. The term *closure*⁴ is an example. A closure is a function with free or open variables that are bound to declarations determinable before run-time. In other words, the declarations to which the *open* variables are bound are *closed* before run-time (i.e., static scoping) rather than left *open* until run-time (i.e., dynamic scoping). (Note that *closures*—functions with free variables—and *combinators*—functions without free variables—are opposites of each other.) “The reason it is called a ‘closure’ is that an expression containing free variables is called an ‘open’ expression, and by associating to it the bindings of its free variables, you close it” (Wikström 1987, p. 125). Since dynamic scoping predates static scoping, initially languages did not have closed functions. For example, the first version of Lisp used dynamic scoping. With the advent of static scoping, some languages had both open and closed functions and needed a way to distinguish between the two. Ultimately, the term *closure* was adopted to refer to the latter.

4. Another use of the term *closure* in computer science is for the *Kleene closure* or *Kleene star* operator (discussed in Chapter 2) used in regular expressions and EBNF grammars to match zero or more of the preceding expression (e.g., the regular expression `aa*` matches the strings `a`, `aa`, `aaa` and so on).

All modern languages relevant to this discussion use static scoping and, thus, all functions are closed; no longer do functions exist containing free variables whose declarations are unknown until run-time. The term *closure* has persisted, but assumed a new meaning. Instead of referring a function whose free variables are all bound to a declaration before run-time, it now means a function containing free variables bound to declarations before run-time that may not exist at run-time (e.g., the function returned by `add_x` that references the environment of `add_x` even after `add_x` has returned). Of course, this mutated sense is difficult to implement and creates the upward FUNARG problem discussed in Section 6.10.2). In turn, the term *closure* has persisted to distinguish between two different types of closed functions rather than between open and closed functions as originally conceived. Some people refer to a closure as a function that “remembers” the lexical environment in which it is created, because its defining environment is packaged within it. This is why we define a closure as an abstract data type with only two pointers: one to an expression and one to an environment (in which to evaluate that expression).

The terms *closure* and *anonymous function* are often mistakenly used interchangeably. Most languages that support anonymous functions allow them to be nested inside another function or scope and returned—thus creating a closure. While a closure “remembers” the environment in which it is created, an anonymous function—which is simply an unnamed function—may not. Multiple languages support closures and anonymous functions (e.g., Python, C#).

6.10.4 Uses of Closures

Closures delay evaluation because they do not perform any computations until called. Programmers can capitalize on this behavior by using them to define control structures. Primitive control (e.g., `if`) and repetition (e.g., `while`) structures in Smalltalk are defined using objects whose methods accept closures (called *blocks* in Smalltalk). Closures can be used to implement user-defined control structures as well. For these reasons, first-class closures are a fundamental primitive in programming languages from which to construct and conceive powerful abstractions (e.g., control structures; see Chapter 13) and concepts (e.g., parameter-passing mechanisms including as lazy evaluation; see Chapter 12).

6.10.5 The Upward and Downward FUNARG Problem in a Single Function

Some functions, including the following, accept one or more functions as arguments and return a function as a value. As a result, they involve both downward and upward FUNARG problems.

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x))))))
```

```

(define list-of
  (lambda (pred)
    (lambda (lst)
      (cond
        ((null? lst) #t)
        ((pred (car lst)) ((list-of pred) (cdr lst)))
        (else #f)))))

```

The following function presents an upward FUNARG problem in the context of a downward FUNARG problem. Specifically, the upward FUNARG returned on line 8 is immediately passed as an argument to the function that is defined on lines 2–5:

```

1  ((lambda (x y)
2    ((lambda (proc2)
3      ((lambda (proc1) (proc1 5 20))
4        ;; this function gets bound to proc1
5        (lambda (x y) (cons x (proc2))))))
6    ((lambda (x y)
7      ;; this function (closure) gets bound to proc2
8      (lambda () (cons x (cons y (cons (+ x y) '())))))
9      100 101)))
10 10 11)

```

6.10.6 Addressing the FUNARG Problem

Since functions are first-class entities in Scheme, any implementation of Scheme must address both the upward and downward FUNARG problems. For this reason, Scheme is sometimes called a *full* FUNARG programming language. Any language with first-class functions must address this problem.

The technique of *λ -lifting* involves converting a closure (i.e., a λ -expression with free variables) into a *pure function* (i.e., a λ -expression with no free variables) by passing values for those free variables as arguments to the λ -expression containing the free variables itself. In a sense, this technique *lifts* the λ in the closure to higher lexical scopes until it has no free variables, at which point it is a top-level function (i.e., in the outermost lexical block). While *λ -lifting* is a simple solution, it does not work in general.

Another approach is to build a closure and pass it to the FUNARG as a argument when the FUNARG is invoked (Programming Exercise 6.10.10). In this way, the closure—a package containing a λ -expression and bindings for its free variables—helps the FUNARG (the λ -expression in the closure) to “remember” the environment in which it was created (the values in the closure).

Some languages address the FUNARG problem by representing function values as closures allocated from the heap,⁵ rather than the stack. Since a lexical closure can and often does outlive the activation record of its parent, the garbage collector must not reclaim activation frames until they are no longer required to provide the bindings for the free variables in any closures created in their scope. Activation

5. The word *heap* when used in the context of dynamic memory allocation does not refer to the heap data structure. Rather, it simply means a heap (or pile) of memory.

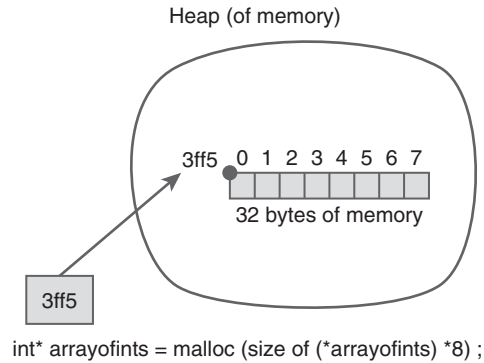


Figure 6.6 The heap (of memory) in a process from which dynamic memory is allocated.

records in Scheme are stored on the heap, so closures in Scheme have *indefinite extent*.

It is important to note which attribute the adjectives *static* and *dynamic* modify in this context. All memory is allocated at run-time when a program becomes a process—that is, a program in execution. For instance, a variable local to a function (e.g., `int x`) is not allocated until the function is called and its activation record is pushed onto the stack at run-time. Thus, the adjectives *static* and *dynamic* cannot be referring to the *time* at which memory is allocated, but instead refer to the *size* of the memory. The size of static data is fixed before run-time (e.g., `int x` is 4 bytes) even though it is not allocated until run-time. Conversely, the size of dynamic memory can grow or shrink at run-time.

Figure 6.6 illustrates the allocation of dynamic memory from the heap using the C function `malloc` (i.e., memory allocation). The function `malloc` accepts the number of bytes the programmer wants to allocate from the heap and returns a pointer to the memory. We use the `sizeof` function to make the allocation portable. On many architectures, `ints` are 4 bytes, so we are allocating 32 bytes of memory from the heap, or an array of 8 integers. However, this allocation is from the heap. If we declared the array as `int arrayofints[8]`, the allocation would come from the stack or static data region.

Although the C programming language supports function pointers, and functions are first-class entities because they can be passed and returned, historically functions could not be nested in a C program. In consequence, the language was not required to address the upward FUNARG problem. The prevention of function nesting also mitigated the downward FUNARG problem. Since functions could not nest, the environment of each function could be entirely specified as the local environment of the function plus the statically allocated global variables and the top-level functions.

Since the allocation of a closure is automatic, happening implicitly when a function is called, languages that allocate closures from the heap (called heap-allocated closures) typically use garbage collection as opposed to manual

memory management, such as through the use of the `free()` function in C. Recall that a closure must remember the activation record of its parent closure, and that the memory occupied by this activation record must not be reclaimed until it is no longer required (i.e., until there are no more remaining references to the closure). Garbage collection is an ideal solution for dealing with closures with unlimited extent. Scheme uses garbage collection, and its use was later adopted by other languages, including Java. We return to the idea of allocating first-class entities from the heap in subsequent chapters, particularly in Chapter 13, where we discuss control.

Conceptual Exercises for Section 6.10

Exercise 6.10.1 Discuss how the *lifetime* of a variable can exceed its *scope*. Sketch a block-structured program to illustrate how this can happen.

Exercise 6.10.2 Hypothesize why the output of the second C program in Section 6.10.2 with the following replacement `main` function differs from the output of the original program. Explain.

```
int main() {
    int (*counter1) () = new_counter (1);
    int (*counter2) () = new_counter (100);

    printf ("%d %d %d %d\n", counter1(),
           counter2(), counter1(), counter1());
}

$ gcc makecounter.c
$ ./a.out
104 103 102 101
```

Exercise 6.10.3 Under which conditions will λ -lifting not work to convert a *closure* (i.e., a λ -expression with free variables) into a *pure function* (i.e., a λ -expression with no free variables)?

Exercise 6.10.4 Give a Scheme expression involving a *closure* (i.e., a λ -expression with free variables) that cannot be converted with λ -lifting into a *pure function* (i.e., a λ -expression with no free variables).

Programming Exercises for Section 6.10

Exercise 6.10.5 Modify the second definition of the `new_counter` function in Scheme in Section 6.10.2 so to incorporate a `step` on the increment into the closure.

Examples:

```
> (define counter1 (new_counter 0 1))
> (define counter2 (new_counter 1 2))
> (define counter50 (new_counter 100 50))
>
```

```

> (counter1)
1
> (counter1)
2
> (counter2)
3
> (counter2)
5
> (counter1)
3
> (counter1)
4
> (counter2)
7
> (counter50)
150
> (counter50)
200
> (counter50)
250
> (counter1)
5

```

Exercise 6.10.6 Investigate the Python qualifiers `nonlocal` and `global` as they relate to the Python closure example in this section. Rewrite the second `new_counter` closure Python program in Section 6.10.2 using one of these qualifiers to avoid the use of a list. In other words, prevent Python from interpreting the inner reference on the left-hand side of the assignment statement as a definition of a new binding rather than a rebinding to an existing definition.

Exercise 6.10.7 Investigate the use of first-class closures in the Go programming language. Define a function `Fibonacci` that returns a closure that, when called, returns progressive Fibonacci numbers. Specifically, fill in the missing lines of code (identified with ellipses) in the following skeletal program:

```

package main

import "fmt"

// Returns a function that returns
// a function that returns
// successive Fibonacci numbers.
func Fibonacci() func() int {
    ...
    return func() int {
        ...
        ...
    }
}

func main() {
    f := fib()
    // Function calls are evaluated left-to-right.
    // Prints: 1 1 2 3 5
    fmt.Println(f(), f(), f(), f(), f())
}

```

Exercise 6.10.8 Go, unlike C, does not have a `static` keyword: A function name or variable whose identifier starts with a lowercase letter has internal linkage, while one starting with an uppercase letter has external linkage. How can we simulate in Go a variable local to a function with static (i.e., global) storage? Write a program demonstrating a variable with both local scope to a function and static (i.e., global) storage. *Hint:* Use a closure.

Exercise 6.10.9 As discussed in Section 6.10.6, λ -lifting is a simple solution to the upward FUNARG problem, but it does not work in all contexts. The technique of λ -lifting involves passing the values of any free variables in a λ -expression as arguments to the function. Consider the following Scheme expression:

```
1 > ((lambda (article1 article2)
2   (let ((buildlist (lambda (l1 l2)
3     (cons (cons article1 l1) (cons article2 l2)))))
4     (append (buildlist '(pamplemousse) '(poire))
5       (buildlist '(raisin) '(pomme))))) 'le 'la)
6 '(le pamplemousse) la poire (le raisin) la pomme)
```

Apply λ -lifting to this expression so that values for the free variable `article1` and `article2` referenced in the λ -expression on lines 2–3 are passed to the λ -expression itself.

Exercise 6.10.10 Rather than using λ -lifting (which does not work in all cases), eliminate the free variables in the Scheme expression from Programming Exercise 6.10.9 by building a closure as a Scheme vector. The vector must contain the λ -expression and the values for the free variables in the λ -expression, in that order. Pass this constructed closure to the λ -expression as an argument when the function is invoked so it can be used to retrieve values for the free variables when they are accessed. The function `vector` is the constructor for a Scheme vector and accepts the ordered values of the vector as arguments—for example, `(define fruit (vector 'apple 'orange 'pear))`. The function `vector-ref` is the vector accessor; for example, `(vector-ref fruit 1)` returns `'orange`.

One way to simulate an object in a language supporting first-class closures is to conceive an object as a vector of member functions whose closure contains the member variables. The type of the vector serves as the interface for the class. The function that creates this vector is the constructor, and its definition resembles a class as demonstrated in this section. (This approach, of course, does not permit inheritance or public member variables—though they can be incorporated.) The next four exercises involve the use of this approach.

Exercise 6.10.11 Define a class `Circle` in Scheme with member variable `radius` and member functions `getRadius`, `getArea`, and `getCircumference`. Access these member functions in the vector representing an object of the class through accessor functions:

```
(define circle-get-setRadius (lambda (c) (vector-ref c 0)))
```

Use this class to run the following program:

```
> (let ((circleA (Circle)) (circleB (Circle)))
  (let
    ((AsetRadius (circle-get-setRadius circleA))
     (AgetRadius (circle-get-getRadius circleA))
     (AgetArea (circle-get-getArea circleA))
     (AgetCircumference (circle-get-getCircumference circleA))
     (BsetRadius (circle-get-setRadius circleB))
     (BgetRadius (circle-get-getRadius circleB))
     (BgetArea (circle-get-getArea circleB))
     (BgetCircumference (circle-get-getCircumference circleB)))
    (let
      ((ignoreA (AsetRadius 5))
       (Aradius (AgetRadius))
       (Aarea (AgetArea))
       (Acircumference (AgetCircumference))
       (ignoreB (BsetRadius 10))
       (Bradius (BgetRadius))
       (Barea (BgetArea))
       (Bcircumference (BgetCircumference)))
      (cons (list Aradius Aarea Acircumference)
             (cons (list Bradius Barea Bcircumference) '())))))
'((5 78.5 31.400000000000002) (10 314.0 62.800000000000004))
```

Exercise 6.10.12 Create a stack object in Scheme, where the stack is a vector of closures and the stack data structure is represented as a list. Specifically, define an argumentless function⁶ `new-stack` that returns a vector of closures—`reset-stack`, `empty-stack?`, `push`, `pop`, and `top`—that access the stack list. You may use the functions `vector`, `vector-ref`, and `set!`. The following client code must work with your stack:

```
> (let ((s1 (new-stack)) (s2 (new-stack)))
  (let
    ((s1reset (stack-get-reset-method s1))
     (s1empty? (stack-get-empty-method s1))
     (s1push (stack-get-push-method s1))
     (s1top (stack-get-top-method s1))
     (s1pop (stack-get-pop-method s1))
     (s2reset (stack-get-reset-method s2))
     (s2empty? (stack-get-empty-method s2))
     (s2push (stack-get-push-method s2))
     (s2top (stack-get-top-method s2))
     (s2pop (stack-get-pop-method s2)))
    (let
      ((d1 (s1push 15))
       (d2 (s2push (+ 1 (s1top))))
       (d3 (s2push (+ 1 (s1top))))
       (if (not (s2empty?)) (s2pop) (s2push "Au revoir")))))
```

16

Exercise 6.10.13 (Friedman, Wand, and Haynes 2001, Section 2.4, p. 66) Create a queue object in Scheme, where the queue is a vector of closures. Specifically, define an argumentless function `new-queue` that returns a vector of closures—

6. The *arity* of a function with zero arguments (i.e., 0-ary) is *nullary* (from *nūllus* in Latin) and *niladic* (from Greek).

queue-reset, enqueue, and dequeue—that access the queue. The dequeue function must contain a private local function queue-empty?. The queue data structure is represented as a list of only two lists to make accessing the queue efficient. You may use the functions vector, vector-ref, and set!. Consider the following examples:

method	argument	q before method	q after method	return value
enqueue	1	' (() ())	' ((1) ())	
enqueue	2	' ((1) ())	' ((2 1) ())	
enqueue	3	' ((2 1) ())	' ((3 2 1) ())	
enqueue	4	' ((3 2 1) ())	' ((4 3 2 1) ())	
dequeue		' ((4 3 2 1) ())	' (() (2 3 4))	1
dequeue		' (() (2 3 4))	' (() (3 4))	2
dequeue		' (() (3 4))	' (() (4))	3
enqueue	5	' (() (4))	' ((5) (4))	
enqueue	6	' ((5) (4))	' ((6 5) (4))	
dequeue		' ((6 5) (4))	' ((6 5) ())	4
dequeue		' ((6 5) ())	' (() (6))	5

The following client code must work with your queue:

```
> (let ((q1 (new-queue)) (q2 (new-queue)))
  (let ((qlresetq (get-qreset-method q1))
        (qlenqueue (get-enqueue-method q1))
        (qldequeue (get-dequeue-method q1))
        (q2resetq (get-qreset-method q2))
        (q2enqueue (get-enqueue-method q2))
        (q2dequeue (get-dequeue-method q2)))
    (let
      ((d1 (qlenqueue 15))
       (d2 (qlenqueue 16))
       (d3 (q2enqueue (+ 1 (qldequeue))))
       (d4 (q2enqueue "Au revoir")))
      (cond
        ((eqv? (q2dequeue) 16) (qldequeue))
        (else (q2dequeue))))))
```

16

Exercise 6.10.14 Consider the binary tree abstraction, and the suite of functions accessing it, created in Section 5.7.1. Specifically, consider the addition of the functions root, left, and right at the end of the example to make the definition of the preorder and inorder traversals more readable (by obviating the necessity of the car-cdr call chains). The inclusion of the root, left, and right helper functions creates a function protection problem. Specifically, because these helper functions are defined at the outermost block of the program, any other functions in that outermost block also have access to them—in addition to the preorder and inorder functions—even though they may not need access to them. To protect these root, left, and right helper functions from functions that do not use them, we can nest them within the preorder function with a letrec expression. That approach creates another problem: The definitions of

the `root`, `left`, and `right` functions need to be replicated in the `inorder` function and any other functions requiring access to them (e.g., `postorder`). Solve this function-protection-access problem in the binary tree program without duplicating any code by using first-class closures.

Exercise 6.10.15 Investigate the *applicative-order Y combinator*, which expresses the essence of recursion using only first-class functions (Section 5.9.3). A derivation of the applicative-order Y combinator is available at <https://www.dreamsongs.com/Files/WhyOfY.pdf> (Gabriel 2001). Since JavaScript supports first-class functions (and uses applicative-order evaluation of function arguments), implement the applicative-order Y combinator in JavaScript. Specifically, construct a webpage with text fields that accept the arguments to factorial, Fibonacci, and exponentiation functions implemented using the Y combinator. When complete, build a series of linearly linked webpages that walk the user through each step in the construction of the Y combinator using a factorial function.

6.11 Deep, Shallow, and Ad Hoc Binding

The presence of first-class procedures makes the determination of the declaration to which a nonlocal reference binds more complex than in languages without support for first-class procedures. The question is: Which environment should be used to supply the value of a nonlocal reference in the body of a passed or returned function? There are three options:

- *deep binding* uses the environment at the time the passed function was *created*
- *shallow binding* uses the environment of the expression that *invokes* the passed function
- *ad hoc binding* uses the environment of the invocation expression in which the procedure is *passed* as an argument

Consider the following Scheme expression:

```

1  (let ((y 3))
2    (let ((x 10)
3          ;; to which declaration of y is the reference to y bound?
4          (f (lambda (x) (* y (+ x x)))))
5
6      (let ((y 4))
7        (let ((y 5)
8              (x 6)
9
10             (g (lambda (x y) (* y (x y)))))
11          (let ((y 2))
12            (g f x))))))

```

The function `(lambda (x) (* y (+ x x)))` that is bound to `f` on line 4 contains a free variable `y`. This function is passed to the function `g` on line 13 in the expression `(g f x)` and invoked (as `x`) on line 10 in the expression `(x y)`.

The question is: To which declaration of `y` does the reference to `y` on line 4 bind? In other words, from which environment does the denotation of `y` on line 4 derive? There are multiple options:

- the `y` declared on line 1
- the `y` declared on line 6
- the `y` declared on line 7
- the `y` declared on line 11

6.11.1 Deep Binding

Scheme uses deep binding. The following Scheme expression is the preceding Scheme expression annotated with comments that indicate the denotations of the identifiers involved in the determination of the declaration to which the `y` on line 4 is bound:

```

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      ?      6 6
4          (f (lambda (x) (* y (+ x x))))))
5
6    (let ((y 4))
7      (let ((y 5)
8            (x 6)
9            ; f 6      6  f 6
10           (g (lambda (x y) (* y (x y))))))
11        (let ((y 2))
12              ; 6
13              (g f x))))))

```

Deep binding evaluates the body of the passed procedure in the environment in which it is *created*. The environment in which `f` is created is `((y 3))`. Therefore, when the argument `f` is invoked using the formal parameter `x` on line 10, which is passed the argument `y` bound to 6 (because the reference to `x` on line 13 is bound to the declaration of `x` on line 8; i.e., static scoping), the return value of `(x y)` on line 10 is `(* 3 (+ 6 6))`. This expression equals 36, so the return value of the call to `g` (on line 13) is `(* 6 36)`, which equals 216. The next three Scheme expressions are progressively annotated with comments to help illustrate the return value of 216 with deep binding:

```

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      3      12
4          (f (lambda (x) (* y (+ x x))))))
5
6    (let ((y 4))
7      (let ((y 5)
8            (x 6)
9            ; f 6      6
10           (g (lambda (x y) (* y (x y))))))
11        (let ((y 2))
12              ; 6
13              (g f x))))))

```

```

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      36
4          (f (lambda (x) (* y (+ x x))))))
5
6    (let ((y 4))
7      (let ((y 5)
8            (x 6)
9            ; f 6      6  36
10           (g (lambda (x y) (* y (x y))))))
11        (let ((y 2))
12          ; 6
13          (g f x))))))

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      36
4          (f (lambda (x) (* y (+ x x))))))
5
6    (let ((y 4))
7      (let ((y 5)
8            (x 6)
9            ; f 6      216
10           (g (lambda (x y) (* y (x y))))))
11        (let ((y 2))
12          ; 216
13          (g f x))))))

```

6.11.2 Shallow Binding

Evaluating this code using shallow binding yields a different result. Shallow binding evaluates the body of the passed procedure in the environment of the expression that *invokes* it. The expression that invokes the passed procedure in this expression is $(x\ y)$ on line 10, and the environment at line 10 is

```

((y 4))
((x 10)
 (f (lambda (x) (* (y (+ x x))))))
((y 3)))

```

Thus, the free variable y on line 4 is bound to 4 on line 6. Evaluating the body, $(* y (+ x x))$, of the passed procedure f in this environment results in $(* 4 (+ 6 6))$, which equals 48. Thus, the return value of the call to g (on line 13) is $(* 6 48)$, which equals 288. The next three Scheme expressions are progressively annotated with comments to help illustrate the return value of 288 with shallow binding:

```

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      4  12
4          (f (lambda (x) (* y (+ x x))))))
5
6    (let ((y 4))
7      (let ((y 5)
8            (x 6)

```

```

9          ; f 6      6
10      (g (lambda (x y) (* y (x y))))
11      (let ((y 2))
12          ; 6
13          (g f x))))))

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      48
4          (f (lambda (x) (* y (+ x x)))))
5
6      (let ((y 4))
7        (let ((y 5)
8              (x 6)
9              ; f 6      6 48
10             (g (lambda (x y) (* y (x y)))))
11          (let ((y 2))
12              ; 6
13              (g f x))))))

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      48
4          (f (lambda (x) (* y (+ x x)))))
5
6      (let ((y 4))
7        (let ((y 5)
8              (x 6)
9              ; f 6      288
10             (g (lambda (x y) (* y (x y)))))
11          (let ((y 2))
12              ; 288
13              (g f x))))))

```

6.11.3 Ad Hoc Binding

Evaluating this code using ad hoc binding yields yet another result. *Ad hoc binding* uses the environment of the invocation expression in which the procedure is *passed* as an argument to evaluate the body of the passed procedure. The invocation expression in which the procedure `f` is *passed* is `(g f x)` on line 13, and the environment at line 13 is

```

((y 2))
(y 5)
(x 6)
(g (lambda (x y) (* y (x y)))))
(y 4))
(x 10)
(f (lambda (x) (* (y (+ x x)))))
(y 3))

```

Thus, the free variable `y` on line 4 is bound to 2 on line 11. Evaluating the body, `(* y (+ x x))`, of the passed procedure `f` in this environment results in `(* 2 (+ 6 6))`, which equals 24. Thus, the return value of the call to `g` (on line 13) is `(* 6 24)`, which equals 144. The next three Scheme expressions are

progressively annotated with comments to help illustrate the return value of 144 with ad hoc binding:

```

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      2      12
4            (f (lambda (x) (* y (+ x x))))))
5
6    (let ((y 4))
7      (let ((y 5)
8            (x 6)
9            ; f 6      6
10              (g (lambda (x y) (* y (x y))))))
11        (let ((y 2))
12          ; 6
13            (g f x))))))

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      24
4            (f (lambda (x) (* y (+ x x))))))
5
6    (let ((y 4))
7      (let ((y 5)
8            (x 6)
9            ; f 6      6      24
10              (g (lambda (x y) (* y (x y))))))
11        (let ((y 2))
12          ; 6
13            (g f x))))))

1  (let ((y 3))
2    (let ((x 10)
3          ; 6      24
4            (f (lambda (x) (* y (+ x x))))))
5
6    (let ((y 4))
7      (let ((y 5)
8            (x 6)
9            ; f 6      144
10              (g (lambda (x y) (* y (x y))))))
11        (let ((y 2))
12          ; 144
13            (g f x))))))

```

The terms *shallow* and *deep* derive from the means used to search the run-time stack. Resolving nonlocal references with shallow binding often results in only searching a few activation records back in the stack (i.e., a shallow search). Resolving nonlocal references with deep binding (even though we do not think of searching the stack) often involves searching deeper into the stack—that is, going beyond the first few activation records on the top of the stack.

Deep binding most closely resembles lexical scoping not only because it can be done before run-time, but also because resolving nonlocal references depends on the nesting of blocks. Conversely, shallow binding most closely resembles dynamic scoping because we cannot determine the calling environment until run-time. Ad hoc binding lies somewhere in between the two. However, deep binding is not the same as static scoping, and shallow binding is not the same as dynamic scoping.

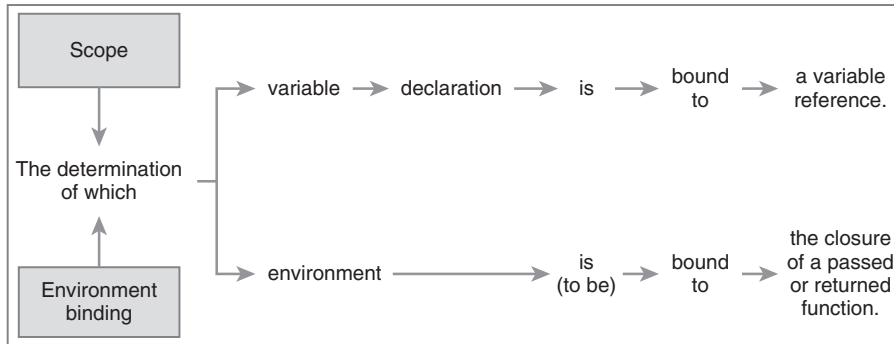


Table 6.7 Scoping Vis-à-Vis Environment Binding

A language that uses lexical scoping can also use shallow binding for passed procedures. Even though we cannot determine the calling environment until run-time (i.e., shallow binding), that environment can contain bindings as a result of static scoping. In other words, while we cannot determine the point in the program where the passed procedure is invoked until run-time (i.e., shallow binding), once it is determined, the environment at that point can be determined before run-time if the language uses static scoping. For instance, the expression that invokes the passed procedure f in our example Scheme expression is $(x\ y)$ on line 10, and we said the environment at line 10 is

```

((y 4))
((x 10)
 (f (lambda (x) (* (y (+ x x))))))
((y 3))

```

That environment, at that point, is based on lexical scoping. Thus, in general, scoping and environment binding are not the same concept even though the rules for each in a particular language indicate how nonlocal references are resolved. Both the type of scoping method used and the type of environment binding used have implications for how to organize an environment data structure most effectively to facilitate subsequent search of it in a language implementation. See Table 6.7; Section 9.8; Chapters 10–11; and Sections 12.2, 12.4, 12.6, and 12.7, where we implement languages.

When McCarthy and his students at MIT were developing the first version of Lisp, they really wanted static scoping, but implemented pure dynamic scoping by accident, and did not address the FUNARG problem. Their implementation of the second version of Lisp attempted to rectify this. However, what they implemented was ad hoc binding, which, while closer to static scoping than what they originally conceived, is not static scoping. Scheme was an early dialect of Lisp that sought to implement lexical scoping.

As stated at the beginning of this chapter, binding is a universal concept in programming languages, and we are by no means through with our treatment of it. This chapter covers the binding of references to declarations—otherwise known as scope. The universality of binding is a theme that recurs frequently in this text.

Conceptual Exercises for Section 6.11

Exercise 6.11.1 Consider the following Scheme program:

```
(define g
  (lambda (f)
    (f)))

(define e
  (lambda ()
    (cdr x)))

(define d
  (lambda (f x)
    (g f)))

(define c
  (lambda ()
    (d e '(m n o))))

(define b
  (lambda (x)
    (c)))

(define a
  (lambda ()
    (b '(c d e))))

(a)
```

- (a) Draw the sequence of procedures on the run-time stack (horizontally, where it grows from left to right) when `e` is invoked (including `e`). Clearly label local variables and parameters, where present, in each activation record on the stack.
- (b) Using *dynamic scoping* and *shallow binding*, what value is returned by `e`?
- (c) Using *dynamic scoping* and *ad hoc binding*, what value is returned by `e`?
- (d) Using *lexical scoping*, what value is returned by `e`?

Exercise 6.11.2 Give the value of the following JavaScript expression when executed using (a) *deep*, (b) *shallow*, and (c) *ad hoc* binding:

```
1  ((x, y) => (
2    ((proc2) => (
3      ((proc1) => proc1(5,20)) ((x, y) => [x, ...proc2()])
4    )
5  )) ((() => [x, y, x + y])
6  )
7  )(10, 11)
```

The `(args) => (body)` syntax in JavaScript, which defines an anonymous/ λ -function, is the same as the `(lambda (args) (body))` syntax in Scheme. The `...` on line 3, called the *spread* operator, is syntactic sugar for inserting the output of the following expression [e.g., `proc2()`] into the list in which it appears.

Exercise 6.11.3 Reconsider the last Scheme example in Section 6.10.5. In that example, an anonymous function is passed on line 8: `(lambda () (cons x (cons y (cons (+ x y) ' ())))`. Since that function is created in the same environment in which it is passed, the result using deep or ad hoc binding is the same: `(5 100 101 201)`. Will the evaluation of any program using deep or ad hoc binding always be the same when every function passed as argument in the program is an anonymous/literal function? If so, explain why. If not, give an example where the two binding strategies lead to different results.

Programming Exercises for Section 6.11

Exercise 6.11.4 ML, Haskell, Common Lisp, and Python all support first-class procedures. Convert the Scheme expression given at the beginning of Section 6.11 to each of these four languages, and state which type of binding each language uses (deep, shallow, or ad hoc).

Exercise 6.11.5 Give a Scheme program that outputs different results when run using deep, shallow, and ad hoc binding.

6.12 Thematic Takeaways

- Programming language concepts often have options, as with scoping (static or dynamic) and nonlocal reference binding (deep, shallow, or ad hoc).
- A *closure*—a function that *remembers* the lexical environment in which was created—is an essential element in the study of language concepts.
- The concept of *binding* is a universal and fundamental concept in programming languages. Languages have many different types of bindings; for example, scope refers to the binding of a reference to a declaration.
- Determining the scope in a programming language that uses manifest typing is challenging because manifest typing blurs the distinction between a variable declaration and a variable reference.
- Lexically scoped identifiers are useful for writing and understanding programs, but are superfluous and unnecessary for evaluating expressions and executing programs.
- The resolution of nonlocal references to the declarations to which they are bound is challenging in programming languages with support for first-class functions. These languages must address the FUNARG problem.

6.13 Chapter Summary

Binding is a relationship from one entity to another in a programming language or program (e.g., the variable `a` is bound to the data type `int`). The establishment of this relationship takes place either before run-time or during run-time. In the context of programming languages, the adjective *static* placed before a noun

phrase indicates that the binding takes place *before run-time*; the adjective *dynamic* indicates that the binding takes place *at run-time*. For instance, the binding of a variable to a data type (e.g., `int a;`) takes place before run-time—typically at compile time—while the binding of a variable to a value takes place at run-time—typically when an assignment statement (e.g., `a = 1;`) is executed. Binding is one of the most foundational concepts in programming languages because other language concepts involve binding. *Scope* is a language concept that can be studied as a type of binding.

Identifiers in a program appear as *declarations* [e.g., in the expressions `(lambda (tail) ...)` and `(let ((tail ...)) ...)` the occurrences of `tail` are as declarations] and as *references* [e.g., in the expression `(cons head tail)`, `cons`, `head`, and `tail` are references]. There is a binding relationship—defined by the programming language—between declarations of and references to identifiers in a program. Each reference is statically or dynamically bound to a declaration that has limited scope. The *scope* of a variable declaration in a program is the region of that program (a range of lines of code) within which references to that variable refer to the declaration (Friedman, Wand, and Haynes 2001). In programming languages that use *static scoping* (e.g., Scheme, Python, and Java), the relationship between a reference and its declaration is established before run-time. In a language using *dynamic scoping*, the determination of the declaration to which a reference is bound requires run-time information, such as the calling sequence of procedures.

Languages have scoping rules for determining to which declaration a particular reference is bound. *Lexical scoping* is a type of static scoping in which the scope of a declaration is determined by examining the lexical layout of the blocks of the program. The procedure for determining the declaration to which a reference is bound in a lexically scoped language is to search the blocks enclosing the reference in an inside-out fashion (i.e., from the innermost block to the outermost block) until a declaration is found. If a declaration is not found, the variable reference is *free* (as opposed to *bound*). Bound references to a declaration can be shadowed by inner declarations using the same identifier, creating a *scope hole*.

Lexically scoped identifiers are useful for writing and understanding programs, but are superfluous and unnecessary for evaluating expressions and executing programs. Thus, we can replace each reference to a lexically scoped identifier in a program with its *lexical depth* and *position*; this pair of non-negative integers serves to identify the declaration to which the reference is bound. Depth indicates the block in which the declaration is found, and position indicates precisely where in the declaration list of that block the declaration is found; they use zero-based indexing from inside-out relative to the reference and left-to-right in the declaration list, respectively. The functions `occurs-free?` and `occurs-bound?` each accept a λ -expression and an identifier and determine whether the identifier occurs free or bound, respectively, in the expression. These functions are examples of programs that process other programs, which we increasingly encounter and develop as we progress toward the interpreter-implementation part of this text (i.e., Chapters 10–12).

The concept of scope is only relevant in the presence of nonlocal references. Resolving nonlocal references in the presence of first-class functions creates a challenge called the FUNARG problem: Which environment should be used to supply the value of a nonlocal reference in the body of a passed or returned function? There are three options: *deep binding* (uses the environment at the time the passed function was *created*), *shallow binding* (uses the environment of the expression that *invokes* the passed function), and *ad hoc binding* (uses the environment of the invocation expression in which the procedure is *passed* as an argument). The FUNARG problem illustrates the relationship between *scope* and *closures*—functions that remember the lexical environment in which they were created. *Closures* and *combinators*— λ -expressions with and without free variables, respectively—are useful programming constructs that we will continue to encounter.

6.14 Notes and Further Reading

Peter J. Landin coined the term *closure* in 1964, and the concept of the closure was first implemented in 1970 in the PAL programming language. Scheme was the first Lisp dialect to use *lexical scoping*. For a derivation of the *Y combinator*, we refer readers to Friedman and Felleisen (1996a, Chapter 9). For the details of dynamic memory allocation and the declaration of pointers to functions in C, we refer readers to Harbison and Steele (1995).

PART II

TYPES

Prerequisite: An understanding of fundamental language and programming background in ML and Haskell, provided in online Appendices B and C, respectively, is requisite for our study of type concepts explored through ML and Haskell in Chapters 7–9.

Chapter 7

Type Systems

Clumsy type systems drive people to dynamically typed languages.

— Robert Griesemer

[A] proof is a program; the formula it proves is a type for the program.

— Haskell Curry and his intellectual descendants

WE study programming language concepts related to types—particularly, *type systems* and *type inference*—in this chapter.

7.1 Chapter Objectives

- Compare the two varieties of *type systems* for *type checking* in programming languages: *statically typed* and *dynamically typed*.
- Describe *type conversions* (e.g., *type coercion* and *type casting*), *parametric polymorphism*, and *type inference*.
- Differentiate between *parametric polymorphism* and *function overloading*.
- Differentiate between *function overloading* and *function overriding*.

7.2 Introduction

The *type system* in a programming language broadly refers to the language's approach to type checking. In a *static type system*, types are checked and almost all type errors are detected before run-time. In a *dynamic type system*, types are checked and most type errors are detected at run-time. Languages with static type systems are said to be *statically typed* or to use *static typing*. Languages with dynamic type systems are said to be *dynamically typed* or to use *dynamic typing*. Reliability, predictability, safety, and ease of debugging are advantages of a statically typed

language. Flexibility and efficiency are benefits of using a dynamically typed language.

The past 20 years have seen the dominance of statically typed languages like Java, C#, Scala, ML, and Haskell. In recent years, however, dynamically typed languages like Scheme, Smalltalk, Ruby, JavaScript, Lua, Perl, and Python have gained in popularity for their ease of extending programs at runtime by adding new code, new data, or even manipulating the type system at runtime. (Wright 2010, p. 16)

There are a variety of methods for achieving a degree of flexibility within the confines of the type safety afforded by some statically typed languages: *parametric* and *ad hoc polymorphism*, and *type inference*.

The type concepts we study in this chapter were pioneered and/or made accessible to programmers in the research projects that led to the development of the languages ML and Haskell. For this reason as well as because of the elegant and concise syntax employed in ML/Haskell for expressing types, we use ML/Haskell as vehicles through which to experience and explore most type concepts in Chapters 7–9.¹ Bear in mind that our objective is not to study how a particular language addresses type concepts, but rather to learn type concepts so that we can understand and evaluate how a variety of languages address type concepts. The interpreted nature, interactive REPL, and terse syntax in ML/Haskell render them appropriate languages through which concepts related to types can be demonstrated with ease and efficacy and, therefore, support this objective.

7.3 Type Checking

A *type* is a set of values (e.g., `int` in $C = \{-2^{15} .. 2^{15} - 1\}$)² and the permissible operations on those values (e.g., `+` and `*`). *Type checking* verifies that the values of types and (new) operations on them—and the values they return—abide by these constraints. For instance, consider the following C program:

```
#include<stdio.h>

void f(int x) {
    printf ("f accepts a value of type int.\n");
}

int main() {
    f(1.7);
}
```

1. The value and utility of ML (Harper, n.d.a, n.d.b) and Haskell (Thompson 2007, p. 6) as teaching languages have been well established.

2. Note that `ints` in C are not guaranteed to be 16-bits; an `int` is only guaranteed to be at least 16-bits. Commonly, on 32-bit and 64-bit processors, an `int` is 32-bits. Programmers can use `int8_t`, `int16_t`, and `int32_t` to avoid any ambiguity.

```
$ gcc notypechecking.c
$
$ ./a.out
f accepts a value of type int.
```

Data types for function parameters in C are not required in function definitions or function declarations (i.e., prototypes):

```
#include<stdio.h>

void f(x) {
    printf ("f accepts a value of any type.\n");
}

int main() {
    f(1.7);
}

$ gcc notypechecking.c
$
$ ./a.out
f accepts a value of any type.
```

A warning is issued if data types for function parameters are not used in function declarations (line 3):

```
1  #include<stdio.h>
2
3  void f(x);
4
5  int main() {
6      f(1.7);
7  }
8
9  void f(x) {
10     printf ("f accepts a value of any type.\n");
11 }

$ gcc notypechecking.c
notypechecking.c:3: warning: parameter names (without types)
in function declaration
$
$ ./a.out
f accepts a value of any type.
```

Languages that permit programmers to deliberately violate the integrity constraints of types (e.g., by granting them access to low-level machine primitives and operations) have *unsound* or *unsafe type systems*. While Fortran, C, and C++ are statically typed languages, they permit the programmer to violate integrity constraints on types and, thus, are sometimes referred to as *weakly typed* languages. For instance, most values in C can be cast to another type of the same storage size. Similarly, Prolog does not try to constrain types. (Lisp does not so much have an unsafe type system as much as it has no type system.) In contrast, Java, ML, and Haskell all have a *sound* or *safe type system*—one that does not permit programmers to circumvent type constraints. Thus, they are sometimes referred to as *strongly typed* or *type safe* languages (Table 7.1). Consider the following Java program:


```

class TypeChecking {
    static void f(int x) {
        System.out.println ("f accepts a value of any type.\n");
    }

    public static void main(String[] args) {
        f(1.7);
    }
}

$ javac TypeChecking.java
TypeChecking.java:8: error: incompatible types: possible
    lossy conversion from double to int
        f(1.7);
          ^
1 error

```

The terms *strongly* and *weakly typed* do not have universally agreed upon definitions in reference to languages or type systems. Generally, a *weakly* or *strongly typed language* is one that does or does not, respectively, permit the programmer to violate the integrity constraints on types. The terms *strong* and *weak* typing are often used to incorrectly mean *static* and *dynamic* typing, respectively, but the two pairs of terms should not be conflated. The nature of a type system (e.g., static or dynamic) and type safety are orthogonal concepts. For instance, C is a *statically* typed language that has a *unsafe* type system, whereas Python is a *dynamically* typed language that has a *safe* type system.

There are a variety of methods for providing programmers with a degree of flexibility within the confines of the type safety afforded by some statically typed languages, thereby mitigating the rigidity enforced by a sound type system. These methods, which include conversions of various sorts, parametric and ad hoc polymorphism, and type inference, are discussed in the following sections.

Concept	Definition	Example(s)
<i>Static type system</i>	Types are checked and almost all type errors are detected before run-time.	C/C++
<i>Dynamic type system</i>	Types are checked and most type errors are detected at run-time.	Python
<i>Safe type system</i>	Does not permit the integrity constraints of types to be deliberately violated.	C#, ML
<i>Unsafe type system</i>	Permits the integrity constraints of types to be deliberately violated.	C/C++
<i>Explicit typing</i>	Requires the type of each variable to be explicitly declared.	C/C++
<i>Implicit typing</i>	Does not require the type of each variable to be explicitly declared.	Python

Table 7.1 Features of Type Systems Used in Programming Languages

7.4 Type Conversion, Coercion, and Casting

Type conversion is the most general of these concepts, in that the other two concepts (i.e., casting and coercion) are instances of conversion. *Conversion* refers to either implicitly or explicitly changing a value from one data type to another. For instance, converting an integer into a floating-point number is an example of conversion. The storage requirements (e.g., from 32 bits to 64 bits) of a value may change as a result of conversion. Type conversions can be either implicit or explicit.

7.4.1 Type Coercion: Implicit Conversion

Coercion is an implicit conversion in which values can deviate from the type required by an operator or function without warning or error because the appropriate conversions are made automatically before or at run-time and are transparent to the programmer. The following C program demonstrates coercion:

```

1  #include<stdio.h>
2
3  int main() {
4
5      int y;
6
7      /* 3.7 is coerced into an int (3) by truncation */
8      y = 3.7;
9
10     printf ("y as an int: %d\n", y);
11     printf ("y as a float: %f\n", y);
12
13     /* 4.1 is coerced into an int (4) by truncation */
14     y = 4.1;
15
16     printf ("y as an int: %d\n", y);
17     printf ("y as a float: %f\n", y);
18
19     /* 1 is coerced into
20        a double (the default floating-point type) */
21     printf ("3.1+1=%f\n", 3.1+1);
22
23     /* 1 is coerced into a double and then the result of
24        the addition is coerced into an int by truncation */
25     y = 3.1+1;
26
27     printf ("y as an int: %d\n", y);
28     printf ("y as a float: %f\n", y);
29 }

```

```

30 $ gcc coercion.c
31 $
32 $ ./a.out
33 y as an int: 3
34 y as a float: -0.000000
35 y as an int: 4
36 y as a float: -0.000000
37 3.1+1=4.100000
38 y as an int: 4
39 y as a float: 4.099998

```

There are five coercions in this program: one each on lines 8, 14, and 21, and two on line 25. Notice also that coercion happens automatically without any intervention from the programmer.

While the details of how coercions happen can be complex and vary from language to language, when integers and floating-point numbers are operands to an arithmetic operator, the integers are usually coerced into floating-point numbers. For example, a coercion is made from an integer to a floating-point number when mixing an integer and a floating-point number with the addition operator; likewise, a coercion is made from a floating-point number to an integer when mixing an integer and a floating-point number with the division operator. In the program just given, when adding an integer and a floating-point number on line 21, the integer (1) is coerced into a floating-point number (1.0) and the result is a floating-point number (line 37).

Such implicit conversions are generally a language implementation issue and dependent on the targeted hardware platform and operating system (because of storage implications). Consequently, language specifications and standards might be general or silent on how coercions happen and leave such decisions to the language implementer. In some cases, the results are predictable:

```

1  #include<stdio.h>
2
3  int main() {
4
5      int fourbyteint = 4;
6      double eightbytedouble = 8.22;
7
8      printf ("The storage required for an int: %d.\n", sizeof(int));
9      printf ("The storage required for a double: %d.\n\n",
10             sizeof(double));
11
12     printf ("fourbyteint: %d.\n", fourbyteint);
13     printf ("eightbytedouble: %f.\n\n", eightbytedouble);
14
15     /* int coerced into a double; */
16     /* smaller type coerced into a larger type; */
17     /* no loss of data */
18     eightbytedouble = fourbyteint;
19
20     printf ("eightbytedouble: %f.\n", eightbytedouble);
21
22     eightbytedouble = 8.0;
23
24     /* double coerced into an int; */
25     /* larger type coerced into a smaller type; */
26     /* truncation results in loss of data */
27     fourbyteint = eightbytedouble;
28
29     printf ("fourbyteint: %d.\n", fourbyteint);
30 }

```

```

31 $ gcc storage.c
32 $
33 $ ./a.out
34 The storage required for an int: 4.
35 The storage required for a double: 8.
36

```

```

37  fourbyteint: 4.
38  eightbytedouble: 8.220000.
39
40  eightbytedouble: 4.000000.
41  fourbyteint: 8.

```

In this program, a value of a type requiring less storage can be generally coerced (or cast) into one requiring more storage without loss of data (lines 18 and 40). However, a value of a type requiring more storage cannot generally be coerced (or cast) into one requiring less storage without loss of data (lines 27 and 41).

In the program `coercion.C`, when the floating-point result of adding an integer and a floating-point number is assigned to a variable of type `int` (line 25), unlike the results of the expressions on lines 8 and 14 (lines 34 and 36, respectively), it remains a floating-point number (line 39). Thus, there are no guarantees with coercion. The programmer forfeits a level of control depending on the language implementation, hardware platform, and OS being used. As a result, coercion, while offering flexibility and relieving the programmer of the burden of using explicit conversions when deviating from the types required by an operator or function, is generally unpredictable, rendering a program using coercion less safe. Moreover, while coercions between values of differing types add flexibility to a program and can be convenient from the programmer's perspective when intended, they also happen automatically—and so can be a source of difficult-to-detect bugs (because of the lack of warnings or errors before run-time) when unintended. Java does not perform coercion, as seen in this program:

```

1  public class NoCoercion {
2      public static void main(String[] args) {
3
4          int x = 2 + 3.2;
5
6          if (false && (1/0))
7              System.out.println("type mismatch");
8      }
9  }

$ javac NoCoercion.java
NoCoercion.java:4: error: incompatible types:
    possible lossy conversion from double to int
    int x = 2 + 3.2;
                ^
NoCoercion.java:6: error: bad operand types for
    binary operator '&&'
    if (false && (1/0))
                ^
    first type:  boolean
    second type:  int
2 errors

```

Java performs no coercion, even between floats and doubles:

```

0  $ cat NoCoercion2.java
1  public class NoCoercion2 {
2      public static void main(String[] args) {
3

```

```

4      F f = new F();
5
6      f.f(1.1);
7  }
8  }
9
10 class F {
11     void f (float x) {
12         System.out.println("f accepts a value of type float.");
13     }
14 }

$ javac NoCoercion2.java
NoCoercion2.java:6: error: incompatible types:
    possible lossy conversion from double to float
    f.f(1.1);
        ^
1 error

```

7.4.2 Type Casting: Explicit Conversion

There are two forms of explicit type conversions: type casts and conversion functions. A *type cast* is an explicit conversion that entails *interpreting* the bit pattern used to represent a value of a particular type as another type. For instance, integer division in C truncates the fractional part of the result, which means that the result must be cast to a float-pointing number to retain the fractional part:

```

1  #include<stdio.h>
2
3  int main() {
4
5      /* integer division truncates by default */
6      printf ("%d\n", 10/3);
7
8      /* must use a type cast to interpret the bit pattern
9       resulting from 10/3 as a value of type float
10      to retain the fractional part */
11      printf ("%f\n", (float) 10/3);
12  }

13 $ gcc cast.c
14 $
15 $ ./a.out
16 3
17 3.333333

```

Here, a type cast, `(float)`, is used on line 11 so that the result of the expression `10/3` is interpreted as a floating-point number (line 17) rather than an integer (line 16).

7.4.3 Type Conversion Functions: Explicit Conversion

Some languages also support built-in or library functions to convert values from one data type to another. For example, the following C program invokes the standard C library function `strtol`, which converts a string representing an

integer into the corresponding long integer, to convert the string "250" to the integer 250:³

```
#include<stdio.h>

int main() {

    char string[] = "250";

    int integer = strtol(string, NULL, 10);

    printf ("The string \"%s\" is represented
           by the integer %d.\n", string, integer);
}

$ gcc conversion.c
$
$ ./a.out
The string "250" is represented by the integer 250.
```

Since the statically typed language ML does not have coercion, it needs provisions for converting values between types. ML supports conversions of values between types through functions. Conversion functions are necessary in Haskell, even though types can be mixed in some Haskell expressions.

7.5 Parametric Polymorphism

Both ML and Haskell assign a unique type to every value, expression, operator, and function. Recall that the type of an operator or function describes the types of its domain and range. Certain operators require values of a particular type. For instance, the `div` (i.e., division) operator in ML requires two operands of type `int` and has type `fn : int * int -> int`, whereas the `/` (i.e., division) operator in ML requires two operands of type `real` and has type `fn : real * real -> real`. These operators are *monomorphic*,⁴ meaning they have only one type.

Other operators or functions are *polymorphic*,⁵ meaning they can accept arguments of different types. For instance, the type of the `(+)` (i.e., prefix addition) operator in Haskell is `(+) :: Num a => (a, a) -> a`,⁶ indicating that if type `a` is in the type class `Num`, then the `(+)` operator has type `(a, a) -> a`. In other words, `(+)` is an operator that maps two values of the same type `a` to a value of the same type `a`.⁷ If the first operand to the `(+)` operator is of type `Int`, then `(+)`

3. Technically, the `strtol` function, which replaces the deprecated `atoi` (ascii to integer) function, accepts a pointer to a character (which is idiom for a string in C since C does not have a primitive string type) and returns a `long`, which in this example is then coerced into an `int`. Nevertheless, it serves to convey the intended point here.

4. The prefixes *mono* and *morph* are of Greek origin and mean *one* and *form*, respectively.

5. The prefix *poly* is of Greek origin and means *many*.

6. The type of the `(+)` (i.e., prefix addition) operator in Haskell is actually `Num a => a -> a -> a` because all built-in functions are fully curried in Haskell. Here, we write the type of the domain as a tuple, and we introduce currying in Section 8.3.

7. The *type variable* `a` indicates an “arbitrary type” (as discussed in online Appendices B and C).

is an operator that maps two `Int`s to an `Int`. This means that the `(+)` operator is polymorphic. With this type of polymorphism, referred to as *parametric polymorphism*, a function or data type can be defined generically so that it can handle arguments in an identical manner, no matter what their type. In other words, the types themselves in the type signature are parameterized. In general, when we use the term *polymorphism* in this text, we are referring to parametric polymorphism.

A polymorphic function type in ML or Haskell specifies that the type of any function with that polymorphic type is one of multiple monomorphic types. Recall that a polymorphic function type is a type expression containing type variables. For example, the polymorphic type `reverse :: [a] -> [a]` in Haskell is a shorthand for a collection of the following (non-exhaustive) list of types: `reverse :: [Int] -> [Int]`, `reverse :: [String] -> [String]`, and so on. The same holds for a qualified polymorphic type. For example, `show :: Show a => a -> String` in Haskell is shorthand for

```
show :: Int -> String,
show :: Float -> String,
show :: Char -> String,
show :: String -> String,
show :: [String] -> String,
```

and so on. A qualified type is sometimes referred to as a constrained type.

Just as each identifier `x` in the function definition `square n = n*n` (in Haskell) stands for the same (arbitrary) value, each type variable in a type expression in ML or Haskell stands for the *same* (arbitrary) type. Every occurrence of a particular type variable (e.g., `a`) in the type expression of an ML or Haskell operator or function, including qualified types in Haskell, stands for the same type. In other words, once the type of any type variable is fixed, the type of any other instance of that same type variable in a type expression is also fixed as that fixed type. For example, instances of the type `(a, a) -> a` include `(Int, Int) -> Int` and `(Bool, Bool) -> Bool`, among others, but not `(Int, Bool) -> Int`. If a type includes different type variables, then each different variable need not have the same type—though they can. For example, instances of the type `(a, b) -> a` include `(Int, Bool) -> Int`, `(Bool, Int) -> Bool`, `(Int, Int) -> Int`, and `(Bool, Bool) -> Bool`, among others, but not `(Int, Bool) -> Bool` or `(Bool, Int) -> Int`.

These examples lead us to an important point differentiating typing in ML and Haskell. Unlike in languages with unsafe type systems (e.g., C or C++), in ML, the programmer is not permitted—because a program doing so will not run—to deviate at all from the required types when invoking an operator or function. For instance, the programmer is not permitted to mix operands of `int` and `real` types at all when invoking arithmetic operators. In ML, the `+`, `-`, and `*` operators only accept two `int` or `real` operands, but not one of each in a single invocation:

```
- 3.1 + 1;
stdIn:1.2-1.9 Error: operator and operand do not agree
[overload - bad instantiation]
```

```

operator domain: real * real
operand:         real * 'Z[INT]
in expression:
  3.1 + 1

- 3.1 + 1.0;
val it = 4.1 : real

- 3 + 1.0;
stdIn:2.1-2.8 Error: operator and operand do not agree
[overload - bad instantiation]
operator domain: 'Z[INT] * 'Z[INT]
operand:         'Z[INT] * real
in expression:
  3 + 1.0

- 3 + 1;
val it = 4 : int

```

This does not mean we cannot have a function in ML that accepts a combination of ints or reals. For instance, the following is a valid function in ML:

```

- fun f (x:int, y:real) = 3;
val f = fn : int * real -> int

- f(1, 1.1);
val it = 3 : int

```

Similarly, the `div` division operator only accepts two `int` operands while the `/` division operator only accepts two `real` operands. For instance:

```

- 10 div 2;
val it = 5 : int

- 10 div 3;
val it = 3 : int

- 10 div 2.0;
stdIn:3.1-3.11 Error: operator and operand do not agree
[overload - bad instantiation]
operator domain: 'Z[INT] * 'Z[INT]
operand:         'Z[INT] * real
in expression:
  10 div 2.0

- 10.0 div 2;
stdIn:1.2-2.1 Error: operator and operand do not agree
[overload - bad instantiation]
operator domain: real * real
operand:         real * 'Z[INT]
in expression:
  10.0 div 2
stdIn:1.7-1.10 Error: overloaded variable not defined at type
symbol: div
type: real

- 10.0 div 3.0;
stdIn:1.7-1.10 Error: overloaded variable not defined at type
symbol: div
type: real

```



```

- 10.0 / 3.0;
val it = 3.333333333333 : real

- 4.0 / 2.0;
val it = 2.0 : real

- 4.2 / 2.1;
val it = 2.0 : real

- 4.3 / 2.5;
val it = 1.72 : real

- 10.0 / 3;
stdIn:7.1-7.9 Error: operator and operand do not agree
[overload - bad instantiation]
operator domain: real * real
operand:         real * 'Z[INT]
in expression:
    10.0 / 3

- 10 / 3.0;
stdIn:1.2-1.10 Error: operator and operand do not agree
[overload - bad instantiation]
operator domain: real * real
operand:         'Z[INT] * real
in expression:
    10 / 3.0

- 10 / 3;
stdIn:1.2-1.8 Error: operator and operand do not agree
[overload - bad instantiation]
operator domain: real * real
operand:         'Z[INT] * 'Y[INT]
in expression:
    10 / 3

- false andalso (1 / 0);
stdIn:4.4-4.9 Error: operator and operand do not agree
[overload - bad instantiation]
operator domain: real * real
operand:         'Z[INT] * 'Y[INT]
in expression:
    1 / 0

- false andalso (1 div 0);
stdIn:1.2-5.1 Error: operand of andalso is not of type bool
[overload - bad instantiation]
operand: 'Z[INT]
in expression:
    false andalso (1 div 0)

```

In Haskell, as in ML, the programmer is not permitted to deviate at all from the required types when invoking an operator or function. However, unlike ML, Haskell has a hierarchy of type classes, where a class is a collection of types, which provides flexibility in function definition and application. Haskell's type class system comprises a hierarchy of interoperable types—similar to the class hierarchies in languages supporting object-oriented programming—where a value of a type (e.g., `Integral`) is also considered a value of one of the supertypes of that type in the hierarchy (e.g., `Num`). Thus, the strict adherence to the type of an operator or function in ML does not appear to apply to Haskell, where values of

different numeric types can (seemingly) be mixed without error in expressions. For instance, the `+`, `-`, and `*` operators appear to accept values of different numeric types. To understand why this is an illusion, we must first discuss how Haskell treats numeric literals.

In Haskell, the following two conversion functions are implicitly applied to numeric literals:

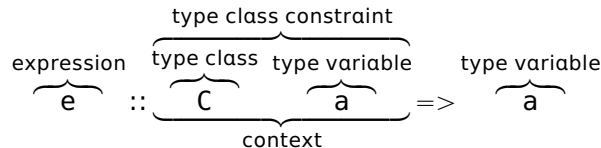
```
Prelude> :type fromInteger
fromInteger :: Num a => Integer -> a

Prelude> :type fromRational
fromRational :: Fractional a => Rational -> a
```

The `fromInteger` function is implicitly (i.e., automatically and transparently to the programmer) applied to every literal number without a decimal point:

```
Prelude> :type 1
1 :: Num p => p
```

This response indicates that if type `p` is in the type class `Num`, then `1` has the type `p`. In other words, `1` is of some type in the `Num` class. Such a type is called a *qualified type* or *constrained type* (Table 7.2). The left-hand side of the `=>` symbol—which here is in the form `C a`—is called the *class constraint* or *context*, where `C` is a type class and `a` is a type variable.



A *type class* is a collection of types that are guaranteed to have definitions for a set of functions—like a Java interface.

The `fromRational` function is similarly implicitly applied to every literal number with a decimal point:

```
Prelude> :type 1.0
1.0 :: Fractional p => p
```

General:

`e :: C a => a` means “If type `a` is in type class `C`, then `e` has type `a`.”

Example:

`3 :: Num a => a` means “If type `a` is in type class `Num`, then `3` has type `a`.”

Table 7.2 The General Form of a *Qualified Type* or *Constrained Type* and an Example

As a result, numeric literals can be mixed as operands to polymorphic numeric functions:

```
Prelude> :type (+)
(+) :: Num a => (a,a) -> a

Prelude> :type (-)
(-) :: Num a => (a,a) -> a

Prelude> :type (*)
(*) :: Num a => (a,a) -> a
```

Consider the following Haskell expression:

```
Prelude> 1 + 1.1
2.1
```

In this expression, the `1` is implicitly passed to `fromInteger`, giving it the type `Num a => a`; the `1.1` is implicitly passed to `fromRational`, giving it the type `Fractional a => a`; and then both are passed to the `+` addition operator. Since the type of the `+` operator is `Num a => (a,a) -> a`, the types of both operands are acceptable because the `Fractional` type class is a subclass of the `Num` class. Here, once the second argument to `+` (`1.1`) is fixed as a fractional type, the first argument to `+` (`1`) is also fixed as the same fractional type, which is acceptable because its qualified type (`Num a => a`) is more general, and the type of the `+` operator is `Fractional a => (a,a) -> a`. Thus, both operands are in agreement. Intuitively, we can say that the `1` is coerced into the most general number type class (`Num`) and then through functional application and type inference (Section 7.9) coerced into the same type class as the `1.1` (`Fractional`) so that both arguments are `Fractional`. The `+` operator expects two `Fractional` operands and receives them as arguments. Note that this is not an example of operator/function overloading. *Overloading* (also called *ad hoc polymorphism*) refers to the provision for multiple definitions of a single function, where the type signature of each definition has a different return type, different types of parameters, and/or a different number of parameters. When an overloaded function is invoked, the applicable function definition to bind to the function call is determined based on the number and/or the types of arguments used in the invocation (Section 7.6). Here, there are not multiple definitions of the `+` addition operator. Instead, the Haskell type class system enables a polymorphic operator/function to accept values of different types in a single invocation. Table 7.3 compares parametric polymorphism and function overloading.

It appears as if Haskell—a statically typed language—uses coercion. However, this is not coercion in the C interpretation of the concept because the programmer can prevent the coercion in Haskell:

```
Prelude> (1::Int) + 1.1

<interactive>:1:12: error:
  No instance for (Fractional Int) arising from the literal '1.1'
  In the second argument of '(+)', namely '1.1'
```

```
In the expression: (1 :: Int) + 1.1
In an equation for 'it': it = (1 :: Int) + 1.1
```

Here we are trying to add a value of type `Int` to a value of type `Fractional a => a` using an operator of type `Num a => (a, a) -> a`. This approach does not work because once the first operand is fixed to be a value of type `Int`, the second operand must be a value of type `Int` as well. However, in this case, the second operand is a value of type `Fractional a => a` and the type `Int` is not a member of the class `Fractional`. Thus, we have a type mismatch. Similar reasoning renders the same type error when the operands are reversed:

```
Prelude> 1.1 + (1::Int)

<interactive>:2:1: error:
  No instance for (Fractional Int) arising from the literal '1.1'
  In the first argument of '(+)', namely '1.1'
  In the expression: 1.1 + (1 :: Int)
  In an equation for 'it': it = 1.1 + (1 :: Int)
```

C uses coercion, whereas Haskell appears to use coercion and, in so doing, provides both safety and flexibility. In C, one can convert a value to any type desired, but values are coerced in an expression if necessary. The same is not true in Haskell: One cannot deviate from the required types of an operator or function. Nevertheless, the type class system in Haskell affords flexibility in allowing values that are not instances of the same type to be operands to operators and functions of qualified types.

There are two division operators in Haskell: one for `Integral` division and one for `Fractional` division.

```
Prelude> :type (div)
div :: Integral a => (a,a) -> a

Prelude> :type (/)
(/) :: Fractional a => (a,a) -> a
```

Reasoning similar to that cited previously indicates that the `/` `Fractional` division operator can also be used to divide a number with a decimal point by a number without a decimal point, or vice versa, or divide a number without a decimal point by another number without a decimal point. However, the `div` `Integral` division operator cannot be used to divide a number with a decimal

Type Concept	Function Definitions	Number of Parameters	Types of Parameters	Example Type Signature(s)
Parametric Polymorphism Function Overloading (Ad Hoc Polymorphism)	single multiple	same varies	parameterized instantiated	[a] -> [a] int -> int int * bool -> float int * float * char -> bool

Table 7.3 Parametric Polymorphism Vis-à-Vis Function Overloading

point by a number without a decimal point, or vice versa, or divide a number with a decimal point by another with a decimal point:

```
Prelude> div 1 2
0

Prelude> div 4 2
2

Prelude> div 4 2.0

<interactive>:1:1: error:
  Ambiguous type variable 'a0' arising from a use of 'print'
  prevents the constraint '(Show a0)' from being solved.
  Probable fix: use a type annotation to specify what 'a0'
    should be.
  These potential instances exist:
    instance Show Ordering -- Defined in 'GHC.Show'
    instance Show Integer -- Defined in 'GHC.Show'
    instance Show a => Show (Maybe a) -- Defined in 'GHC.Show'
    ...plus 22 others
    ...plus 13 instances involving out-of-scope types
    (use -fprint-potential-instances to see them all)
  In a stmt of an interactive GHCi command: print it

Prelude> div 4.0 2

<interactive>:2:1: error:
  Ambiguous type variable 'a0' arising from a use of 'print'
  prevents the constraint '(Show a0)' from being solved.
  Probable fix: use a type annotation to specify what 'a0'
    should be.
  These potential instances exist:
    instance Show Ordering -- Defined in 'GHC.Show'
    instance Show Integer -- Defined in 'GHC.Show'
    instance Show a => Show (Maybe a) -- Defined in 'GHC.Show'
    ...plus 22 others
    ...plus 13 instances involving out-of-scope types
    (use -fprint-potential-instances to see them all)
  In a stmt of an interactive GHCi command: print it

Prelude> div 4.0 2.0

<interactive>:3:1: error:
  Ambiguous type variable 'a0' arising from a use of 'print'
  prevents the constraint '(Show a0)' from being solved.
  Probable fix: use a type annotation to specify what 'a0'
    should be.
  These potential instances exist:
    instance Show Ordering -- Defined in 'GHC.Show'
    instance Show Integer -- Defined in 'GHC.Show'
    instance Show a => Show (Maybe a) -- Defined in 'GHC.Show'
    ...plus 22 others
    ...plus 13 instances involving out-of-scope types
    (use -fprint-potential-instances to see them all)
  In a stmt of an interactive GHCi command: print it

Prelude> 1.0 / 2.0
0.5

Prelude> 4.0 / 2.0
2.0
```

```
Prelude> 4.2 / 2.1
2.0
```

```
Prelude> 4.4 / 2.1
2.0952380952381
```

```
Prelude> 1.0 / 2
0.5
```

```
Prelude> 4 / 2.0
2.0
```

```
Prelude> 4 / 2
2.0
```

The ability of the `/` `Fractional` division operator to divide a number with a decimal point by one without a decimal point is certainly convenient. Moreover, it means that user-defined functions with the same type as the `/` division operator behave similarly when passed arguments of different types. For instance, consider the following definition of a `halve` function in Haskell:

```
halve :: Fractional a => a -> a
halve x = 0.5 * x
```

This function, like the `/` `Fractional` division operator, can be passed a number without a decimal point:

```
*Main> halve 2
1.0
```

However, consider the following definition of a function that accepts the numeric average of a list of numbers:

```
Prelude> listaverage_wrong l = sum l / length l

<interactive>:1:23: error:
  Could not deduce (Fractional Int) arising from a use of '/'
  from the context: Foldable t
    bound by the inferred type of
      listaverage_wrong :: Foldable t => t Int -> Int
  at <interactive>:1:1-38
  In the expression: sum l / length l
  In an equation for 'listaverage_wrong':
    listaverage_wrong l = sum l / length l
```

The problem here is that while the type of the `sum` function is `(Foldable t, Num a) => t a -> a`, the type of the `length` function is `Foldable t => t a -> Int`. Thus, it returns a value of type `Int`, not one of type `Num a => a`, and the type `Int` is not a member of the `Fractional` class required by the `/` `Fractional` division operator. The type class system with coercion used in Haskell to deal with the rigidity of a sound type system adds complexity to the language.

The following transcript of a session with Haskell demonstrates the same arithmetic expressions given previously in ML, but formatted in Haskell syntax:

```

Prelude> 3.1 + 1
4.1

Prelude> 3.1 + 1.0
4.1

Prelude> 3 + 1.0
4.0

Prelude> 3 + 1
4

Prelude> div 10 2
5

Prelude> div 10 3
3

Prelude> div 10 2.0
ERROR - Unresolved overloading
*** Type      : (Fractional a, Integral a) => a
*** Expression : 10 `div` 2.0

Prelude> div 10.0 2
ERROR - Unresolved overloading
*** Type      : (Fractional a, Integral a) => a
*** Expression : 10.0 `div` 2

Prelude> div 10.0 3.0
ERROR - Unresolved overloading
*** Type      : (Fractional a, Integral a) => a
*** Expression : 10.0 `div` 3.0

Prelude> 10.0 / 3.0
3.3333333333333333

Prelude> 4.3 / 2.5
1.72

Prelude> 10.0 / 3
3.3333333333333333

Prelude> 10 / 3.0
3.3333333333333333

Prelude> 10 / 3
3.3333333333333333

Prelude> False && (1 / 0)
ERROR - Cannot infer instance
*** Instance  : Fractional Bool
*** Expression : False && 1 / 0

Prelude> False && (div 1 0)
ERROR - Cannot infer instance
*** Instance  : Integral Bool
*** Expression : False && 1 `div` 0

```

A consequence of having to rigidly follow the prescribed type of an operator or a function is that languages that enforce strict type constraints, including ML, Haskell, and Java, cannot use coercion. If they did, then they could not detect all type errors statically.

7.6 Operator/Function Overloading

Operator/function overloading refers to using the same function name for multiple function definitions, where the type signature of each definition involves a different return type, different types of parameters, and/or a different number of parameters. When an overloaded function is invoked, the applicable function definition to bind to the function call (obtained from a collection of definitions with the same name) is determined based on the number and/or the types of arguments used in the invocation. Function/operator overloading is also called *ad hoc polymorphism*. In general, operators/functions cannot be overloaded in ML and Haskell because every operator/function must have only one type:

```
- (* the second definition of the *)
- (* function f redefines the first *)

- fun f (x:int, y:int) = 4;
val f = fn : int * int -> int

- fun f (x:int, y:real) = 3;
val f = fn : int * real -> int

- f(1,2);
stdIn:8.1-8.7 Error: operator and operand do not agree
[overload - bad instantiation]
operator domain: int * real
operand:          int * 'Z[INT]
in expression:
  f (1,2)

- f(1,2.2);
val it = 3 : int

0 $ cat overloading.hs
1 f :: (Int, Int) -> Int
2 f (x,y) = 3
3
4 f :: (Int, Float) -> Int
5 f (x,y) = 3

$ ghci overloading.hs
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main                ( overloading.hs, interpreted )

overloading.hs:4:1: error:
  Duplicate type signatures for 'f'
    at overloading.hs:1:1
      overloading.hs:4:1
  |
4 | f :: (Int, Float) -> Int
  | ^

overloading.hs:5:1: error:
  Multiple declarations of 'f'
    Declared at: overloading.hs:2:1
                 overloading.hs:5:1
  |
```



```

5 | f (x,y) = 3
  | ^
Failed, no modules loaded.

```

Even in C, functions cannot be overloaded:

```

0 $ cat nooverloading.c
1 #include <stdio.h>
2
3 void f(int x) {
4     printf ("f accepts a value of type int.\n");
5 }
6
7 void f(double x) {
8     printf ("f accepts a value of type double.\n");
9 }
10
11 int main() {
12     f(1.7);
13 }

$ gcc nooverloading.c
nooverloading.c:7:6: error: conflicting types for 'f'
    void f(double x) {
        ^
nooverloading.c:3:6: note: previous definition of 'f' was here
    void f(int x) {
        ^

```

Thus, ML, Haskell, and C do not support function overloading; C++ and Java do support function overloading:

```

$ cat overloading.cpp
#include <iostream>

using namespace std;

void f(int x) {
    cout << "f accepts a value of type int." << endl;
}

void f(double x) {
    cout << "f accepts a value of type double." << endl;
}

int main() {
    f(1.7);
}
$
$ g++ overloading.cpp
$
$ ./a.out
f accepts a value of type double.

$ cat overloading2.cpp
#include <iostream>

using namespace std;

```

```

class Overloading {
public:
    void f (int x) {
        cout << "f accepts a value of type int." << endl;
    }

    void f (double x) {
        cout << "f accepts a value of type double." << endl;
    }

};

int main() {
    Overloading o;

    o.f(1);
    o.f(1.1);
}
$
$ g++ overloading2.cpp
$
$ ./a.out
f accepts a value of type int.
f accepts a value of type double.

$ cat Overloading.java
public class Overloading {
    public static void main(String[] args) {

        Overload o = new Overload();

        o.f(1);

        o.f(1.1);

    }
}

class Overload {

    void f (int x) {
        System.out.println("f accepts a value of type int.");
    }

    void f (double x) {
        System.out.println("f accepts a value of type double.");
    }

}
$
$ javac Overloading.java
$
$ java Overloading
f accepts a value of type int.
f accepts a value of type float.

```

The extraction (i.e., input, `>>`) and insertion (i.e., output, `<<`) operators are commonly overloaded in C++ to make I/O of user-defined objects convenient:

```

$ cat overloading_operators.cpp
#include <iostream>

using namespace std;

```

```

class Employee {
private:
    int id;
    string name;
    double rate;

public:
    friend ostream& operator<< (ostream &out, Employee &e);
    friend istream& operator>> (istream &in, Employee &e);
};

istream& operator>> (istream &in, Employee &e) {
    in >> e.id;
    in >> e.name;
    in >> e.rate;

    return in;
}

ostream& operator<< (ostream &out, Employee &e) {
    out << "(id: " << e.id <<
        ", name: " << e.name <<
        ", rate: " << e.rate << ")";
    return out;
}

int main() {
    Employee Mary, Lucia;

    cin >> Mary;
    cin >> Lucia;

    cout << Mary << endl;
    cout << Lucia << endl;
}
$
$ g++ overloading_operators.cpp
$
$ ./a.out
1234 Mary 3.90
5678 Lucia 9.21
(id: 1234, name: Mary, rate: 3.9)
(id: 5678, name: Lucia, rate: 9.21)

```

Since ML does not support operators/function overloading,⁸ we cannot define a square function in ML that accepts any numeric value (e.g., integer or floating point):

```

1 - fun square(n) = n*n;
2 val square = fn : int -> int
3
4 - square(2);
5 val it = 4 : int
6
7 - square(2.0);
8 stdIn:7.1-7.12 Error: operator and operand do not agree
9   [tycon mismatch]

```

8. Some of the commonly used (arithmetic) primitive operators in ML are overloaded (e.g., binary addition).

```

10  operator domain: int
11  operand:         real
12  in expression:
13    square 2.0

```

The data type `int` is the default numeric type in ML (Section 7.9). However, we can define a `square` function in Haskell that accepts any numeric value:

```

Prelude> square(n) = n*n

Prelude> :type square
square :: Num a => a -> a

Prelude> square(2)
4

Prelude> square(2.0)
4.0

```

The Haskell type class system supports the definition of what seem to be overloaded functions like `square`.⁹ Recall that the type class system allows values of different types to be used interchangeably if those types are properly related in the hierarchy. The flexibility fostered by a type or class hierarchy in the definition of functions is similar to ad hoc polymorphism (i.e., overloading), but is called *interface polymorphism*.

While they take advantage of the various concepts that render a static type system more flexible, ML and Haskell come with irremovable type checks for safety that generate error messages for discovered type errors and type mismatches.¹⁰ Put simply, ML and Haskell programs are thoroughly type-checked before run-time. Almost no ML or Haskell program that can run will ever have a type error. As a result, an ML or Haskell program that passes all of the requisite type checks almost never fails.

7.7 Function Overriding

Function overriding (also called *function hiding*) occurs when multiple function definitions share the same function name, but only one of those function definitions is visible at any point in the program due to the presence of scope holes. For instance:

```

1  (define overriding
2    (lambda ()
3      (let ((f (lambda ()
4                  (let ((g (lambda ()
5                              (let ((f (lambda () (+ 1 2))))
6                                ;; call to inner f
7                                (f))))))

```

⁹ Functions like `square` are often generally referred to as *overloaded* functions in Haskell programming books and resources.

¹⁰ Ada gives the programmer the ability to suspend type checking.

```

8           (g))))
9       ;; call to outer f
10      (f)))

```

Here, the call to function `f` on line 10 binds to the outermost definition of `f` (starting on line 3) because the innermost definition of `f` (line 5) is not visible on line 10—it is defined in a nested block. The call to function `f` on line 7 binds to the innermost definition of `f` (line 5) because on line 7 where `f` is called, the innermost definition of `f` (line 5) shadows the outermost definition of `f`. In other words, the outermost definition of `f` is not visible on line 7.

7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing

The concepts of *static/dynamic typing* and *explicit/implicit typing* are sometimes confused and used interchangeably. The modifiers “static” or “dynamic” on “typing” (or “checking”) indicate the time at which types and type errors are checked. However, the types of those variables can be declared explicitly (e.g., `int x = 1;` in Java) or implicitly (e.g., `x = 1` in Python). Languages that require the type of each variable to be explicitly declared use *explicit typing*; languages that do not require the type of each variable to be explicitly declared use *implicit typing*, which is also referred to as *manifest typing* (Table 7.1). Statically typed languages can use either explicit (e.g., Java) or implicit (e.g., ML and Haskell) typing. Dynamically typed languages typically use implicit typing (e.g., Python, JavaScript, Ruby). There are no advantages to using explicit typing in a dynamically typed language.

7.9 Type Inference

Explicit type declarations of values and variables help inform a static type system. For example, consider these explicit declarations of types for entities in ML:

```

- (* declaring 3.0 to be real *)
- 3.0 : real;
val it = 3.0 : real

- (* declaring identifier x to be real *)
- let val (x: real) = 3.0 in x end;
val it = 3.0 : real

- fun square(n: real) : real = n*n;
val square = fn : real -> real

- fun add(x: real, y: real) : real = x + y;
val add = fn : real * real -> real

```

Types for values, variables, function parameters, and return types are similarly declared in Haskell:

```

$ cat declaring.hs
square(n :: Double) = n*n :: Double

add(x :: Double, y :: Double) = x + y :: Double
$
$ ghci -XScopedTypeVariables declaring.hs
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main                ( declaring.hs, interpreted )
Ok, one module loaded.

*Main> 3.0 :: Double
3.0

*Main> let (x :: Double) = 3.0 in x
3.0

*Main> :type square
square :: Double -> Double

*Main> :type add
add :: (Double, Double) -> Double

```

In some languages with first-class functions, especially statically typed languages, functions have types. Instead of ascribing a type to each individual parameter and the return type of a function, we can declare the type of the entire function. In ML, a programmer can explicitly declare the type of an entire anonymous function and then bind the function definition to an identifier:

```

- val square: real -> real = (fn n => n * n);
val square = fn : real -> real

- val add: real * real -> real = (fn (x,y) => x + y);
val add = fn : real * real -> real

```

In Haskell, a programmer can explicitly declare the type of both a non-anonymous and an anonymous function:

```

$ cat declaring.hs
square :: Double -> Double
square(n) = n*n

add :: (Double, Double) -> Double
add(x,y) = x+y
$
$ ghci declaring.hs

*Main> :type square
square :: Double -> Double

*Main> :type add
add :: (Double, Double) -> Double

*Main> square = (\n -> n*n) :: Double -> Double
*Main> :type square
square :: Double -> Double

*Main> add = (\(x,y) -> (x + y)) :: (Double, Double) -> Double
*Main> :type add
add :: (Double, Double) -> Double

```

Explicitly declaring types requires effort on the part of the programmer and can be perceived as requiring more effort than necessary to justify the benefits of a static type system. Type inference is a concept of programming languages that represents a compromise and attempts to provide the best of both worlds. *Type inference* refers to the automatic deduction of the type of a value or variable *without* an explicit type declaration. ML and Haskell use type inference, so the programmer is not required to declare the type of any variable unless necessary (e.g., in cases where it is impossible for type inference to deduce a type). Both languages include a built-in type inference engine to deduce the type of a value based on context. Thus, ML and Haskell use type inference to relieve the programmer of the burden of associating a type with every name in a program. However, an explicit type declaration is required when it is impossible for the inference algorithm to deduce a type. ML introduced the idea of type inference in programming languages in the 1970s.

Both ML and Haskell use the *Hindley–Milner algorithm* for type inference. While the details of this algorithm are complex and beyond the scope of this text, we will make some cursory remarks on its use. Understanding the fundamentals of how these languages deduce types helps the programmer know when explicit type declarations are required and when they can be omitted. Though not always necessary, in ML and Haskell, a programmer can associate a type with (1) values, (2) variables, (3) function parameters, and (4) return types. The main idea in type inference is this: Since all operands to a function or operator must be of the required type, and since values of differing numeric types cannot be mixed as operands to arithmetic operators, once we know the type of one or more values in an expression (because, for example, it was explicitly declared to be of that type) by transitive inference we can progressively determine the type of each other value. In essence, knowledge of the type of a value (e.g., a parameter or return value) can be leveraged as context to determine the types of other entities in the same expression. For instance, in ML:

```
- fun square'(n : real) = n*n;
val square' = fn : real -> real

- fun square''(n) : real = n*n;
val square'' = fn : real -> real

- (* declaring parameter x to add' to be real *)
- fun add'(x: real, y) = x + y;
val add' = fn : real * real -> real

- (* declaring parameter y to add'' to be real *)
- fun add''(x, y: real) = x + y;
val add'' = fn : real * real -> real

- (* declaring add''' to return a real *)
- fun add'''(x,y) : real = x + y;
val add''' = fn : real * real -> real
```

Declaring the parameter `x` to be of type `real` is enough for ML to deduce the type of the function `add'` as `fn : real * real -> real`. Since the first operand to the `+` operator is a value of type `real`, the second operand must also be of type

real because the types of the two operands must be the same. In turn, the return type is a value of type `real` because the sum of two values of type `real` is a value of type `real`. A similar line of reasoning is used in ML to deduce that the type of `add` and the type of `add''` is `fn : real * real -> real`. The Haskell analogs of these examples follow:

```
$ cat declaring.hs
square' (n :: Double) = n*n

square'' (n) = n*n :: Double

add' (x :: Double, y) = x + y

add'' (x, y :: Double) = x + y

add''' (x,y) = x + y :: Double
$
$ ghci -XScopedTypeVariables declaring.hs
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( declaring.hs, interpreted )
Ok, one module loaded.

*Main> :type square'
square' :: Double -> Double

*Main> :type square''
square'' :: Double -> Double

*Main> :type add'
add' :: (Double, Double) -> Double

*Main> :type add''
add'' :: (Double, Double) -> Double

*Main> :type add'''
add''' :: (Double, Double) -> Double
```

In these ML and Haskell examples, where partial or complete type information is provided, the explicitly declared type is not always the same as the type that would have been inferred. For instance, in ML:

```
- 3.0;
val it = 3.0 : real

- let val x = 3.0 in x end;
val it = 3.0 : real

- fun square(n) = n*n;
val square = fn : int -> int

- fun add(x,y) = x + y;
val add = fn : int * int -> int
```

In Haskell, for these examples, the inferred type is never the same as the declared type:

```
Prelude> :type 3.0
3.0 :: Fractional p => p
```



```

Prelude> :type let x = 3.0 in x
let x = 3.0 in x :: Fractional p => p

Prelude> square(n) = n*n
Prelude> :type square
square :: Num a => a -> a

Prelude> add(x,y) = x+y
Prelude> :type add
add :: Num a => (a, a) -> a

```

In general, we only explicitly declare the type of an entity in ML or Haskell when the inferred type is not the intended type. If the inferred type is the same as the intended type, explicitly declaring the type is redundant. For instance:

```

- (* declaring parameter x to be real *)
- fun add1(x: real, y) = x + 1.0;
val add1 = fn : real -> real

- (* declaring add2 to return a real *)
- fun add2(x,y) : real = x + 2.0;
val add2 = fn : real -> real

- (* the inferred types of these functions are *)
- (* the same as the intended types *)

- fun add1(x,y) = x + 1.0;
val add1 = fn : real -> real

- fun add2(x,y) = x + 2.0;
val add2 = fn : real -> real

```

With a named function, we must provide the type inference engine in ML with partial information from which to deduce the intended type of the function by associating a type with a parameter, variable, and/or return value. Sometimes no explicit type declaration (of a parameter or return value) is required, and the context of the expression is sufficient for ML or Haskell to infer a particular intended function type. For instance, consider the following ML function:

```

- fun f(a, b) = if (a + 0.0) < b then 1 else 2;
val f = fn : real * real -> int

```

Here, the type of `f` is inferred: Adding `0.0` to `a` means that `a` must be of type `real` (because the numeric type of each operand must match), so `b` must be of type `real`. Consider another example where information other than an explicitly declared type is used as a basis for type inference:

```

- fun sum([]) = 0
  | sum(x::xs) = x + sum(xs);
val sum = fn : int list -> int

```

Here, the `0` returned in the first case of the `sum` function causes ML to infer the type `int list -> int` for the function `sum` because `0` is an integer and a function can only return a value of one type.

In ML, when there is no way to determine the type of an operand of an operator, such as `+`, the type of the operand is inferred to be the default type for that operator. The default numeric type of any operand for arithmetic operators (e.g., `+`, `-`, `*`, and `<`) and numeric values is `int` in ML. For instance, consider the following ML functions and their inferred types:

```
- fun add(x,y) = x+y;
val add = fn : int * int -> int

- fun f(a, b) = if (a < b) then 3.0 else 2.0;
val f = fn : int * int -> real
```

In an `if ...then ...else` expression, the conditional expression as well as the expressions following the lexemes `if` and `else` must return a value of the same type:

```
- fun f(a, b) = if (a < b) then 3.0 else 2;
stdIn:1.16-1.42 Error: types of if branches do not agree
[overload - bad instantiation]
    then branch: real
    else branch: 'Z[INT]
in expression:
    if a < b then 3.0 else 2
```

Lastly, remember that ML supports polymorphic types, so the inferred type of some functions includes type variables:

```
- fun reverse([]) = []
  | reverse(x::xs) = reverse(xs) @ [x];
val reverse = fn : 'a list -> 'a list
```

In Haskell every expression must have a type, which is calculated prior to evaluating the expression by a process called *type inference*. The key to this process is a typing rule for function application, which states that if f is a function that maps arguments of type A to results of type B , and e is an expression type A , then the application $f\ e$ has type B :

$$\frac{f :: A \rightarrow B \quad e :: A}{f\ e :: B}$$

For example, the typing `¬ False :: Bool` can be inferred from this rule using the fact that `not :: Bool → Bool` and `False :: Bool`. (Hutton 2007, pp. 17–18)

Recall that it is not possible in ML and Haskell to deviate from the types required by operators and functions. However, type inference offers some relief from having to declare a type for all entities. Notably, it supports static typing without explicit type declarations. If you know the intended type of a user-defined function, but are not sure which type will be inferred for it, you may explicitly declare the type of the entire function (rather than explicitly declaring types of selective parameters or values, or the return type, to assist the inference

engine in deducing the intended type), if possible, rather than risk that the inferred type is not the intended type. Conversely, if it is clear that the type that will be inferred is the same as the intended type, there is no need to explicitly declare the type of a user-defined function. Let the inference engine do that work for you.

Strong typing provides safety, but requires a type to be associated with every name. The use of type inference in a statically typed language obviates the need to associate a type with each identifier:

Static, Safe Type System + Type Inference Obviates the Need to Declare Types

Static, Safe Type System + Type Inference $\rightsquigarrow$ Reliability/Safety + Manifest Typing

7.10 Variable-Length Argument Lists in Scheme

Thus far in our presentation of Scheme we have defined functions where the parameter list of each function, like any other list in Scheme, is enclosed in parentheses. For example, consider the following identity function, which can accept an atom or a list (i.e., it is polymorphic):

```
> (define f (lambda (x) x))

> (f 1)
1

> (f 1 2)
procedure f: expects 1 argument, given 2: 1 2

> (f 1 2 3)
procedure f: expects 1 argument, given 3: 1 2 3

> (f '(1 2 3))
'(1 2 3)
```

The second and third cases fail because `f` is defined to accept only one argument, and not two and three arguments, respectively.

Every function in Scheme is defined to accept only *one* list argument. We did not present Scheme functions in this way initially because most readers are probably familiar with C, C++, or Java functions that can accept one or more arguments. Arguments to any Scheme function are always received collectively as *one* list, not as individual arguments. Moreover, Scheme, like ML and Haskell, does pattern matching from this single list of arguments to the specification of the parameter list in the function definition. For instance, in the first invocation just given, the argument `1` is received as `(1)` and then pattern matched against the parameter specification `(x)`; as a result, `x` is bound to `1`. In the second invocation, the arguments `1 2` are received as the list `(1 2)` and then pattern matched against the parameter specification `(x)`, but the two cannot be matched. Similarly, in the third invocation, the arguments `1 2 3` are received as the list `(1 2 3)` and then pattern matched against the parameter specification `(x)`, but the two cannot be

matched. In the fourth invocation, the argument `' (1 2 3)` is received as the list `((1 2 3))` and then pattern matched against the parameter specification `(x)`; as a result, `x` is bound to `(1 2 3)`.

Scheme, like ML and Haskell, performs pattern matching from arguments to parameters. However, since lists in ML and Haskell must contain elements of the same type (i.e., homogeneous), the pattern matching in those languages is performed against the arguments represented as a *tuple* (which can be heterogeneous). In Scheme, the pattern matching is performed against a *list* (which can be heterogeneous). This difference is syntactically transparent since both lists in Scheme and tuples in ML and Haskell are enclosed in parentheses.

Even though any Scheme function can accept only *one* list argument, because a list may contain any number of elements, including none, any Scheme function can *effectively* accept any *fixed* or *variable* number of arguments. (A function capable of accepting a variable number of input arguments is called a *variadic* function.¹¹) To restrict a function to a particular number of arguments, a Scheme programmer must write the parameter specification, from which the arguments are matched, in a particular way. For instance, `(x)` is a one-element list that, when used as a parameter list, forces a function to accept only one argument. Similarly, `(x y)` is a two-element list that, when used as a parameter list, forces a function to accept only two arguments, and so on. This is the typical way in which we have defined Scheme functions:

```
> ((lambda (x) (cons x '())) 1)
'(1)

> ((lambda (x) (cons x '())) 1 2)
#<procedure>: arity mismatch;
the expected number of arguments does not match the given number
expected: 1
given: 2
arguments...:

> ((lambda (x y) (cons x (cons y '()))) 1 2)
'(1 2)

> ((lambda (x y) (cons x (cons y '()))) 1)
#<procedure>: arity mismatch;
the expected number of arguments does not match the given number
expected: 2
given: 1
arguments...:
```

By removing the parentheses around the parameter list in Scheme, and thereby altering the pattern from which arguments are matched, we can specify a function that accepts a *variable* number of arguments. For instance, consider a slightly modified definition of the identity function, and the same four invocations as shown previously:

11. The word *variadic* is of Greek origin.

```

> (define f (lambda x x))

> (f 1)
'(1)

> (f 1 2) ; x is bound to the list (1 2)
'(1 2)

> (f 1 2 3) ; x is bound to the list (1 2 3)
'(1 2 3)

> (f '(1 2 3))
'((1 2 3))

```

In the first invocation, the argument 1 is received as the list (1) and then pattern matched against the parameter specification x ; as a result, x is bound to (1). In the second invocation, the arguments 1 2 are received as the list (1 2) and then pattern matched against the parameter specification x ; as a result, x is bound to the list (1 2). In the third invocation, the arguments 1 2 3 are received as the list (1 2 3) and then pattern matched against the parameter specification x ; x is bound to the list (1 2 3). In the fourth invocation, the argument '(1 2 3) is received as the list ((1 2 3)) and then pattern matched against the parameter specification x ; x is bound to ((1 2 3)). Thus, now the second and third cases work because this modified identity function can accept a variable number of arguments.

A programmer in ML or Haskell can decompose a *single list argument* in the formal parameter specification of a function definition using the $::$ and $:$ operators, respectively [e.g., `fun f (x::xs, y::ys) = ...` in ML]. A Scheme programmer can decompose an entire *argument list* in the formal parameter specification of a function definition using the dot notation. Note that an *argument list* is not the same as a *list argument*. A function can accept multiple list arguments, but has only one argument list. Therefore, while ML and Haskell allow the programmer to decompose individual list arguments using the $::$ and $:$ operators, respectively, a Scheme programmer can only decompose the entire argument list using the dot notation.

The ability to decompose the entire argument list (and the fact that arguments are received into any function as a single list) provides another way for a function to accept a variable number of arguments. For instance, consider the following definitions of `argcar` and `argcdr`, which return the `car` and `cdr` of the argument list received:

```

;;; uses pattern matching as in ML/Haskell
;;; argcar and argcdr accept a 'variable' number of arguments

> (define argcar (lambda (x . xs) x))

> (define argcdr (lambda (x . xs) xs))

> ;; only 1 argument passed
> (argcar 1)
1

```

```

> ;; still only 1 (albeit list) argument passed
> (argcar '(1 2 3))
'(1 2 3)

> (argcdr 1)
'()

> (argcdr '(1 2 3))
'()

;; only 2 arguments passed
> (argcar 1 2)
1

> (argcar 1 '(2 3))
1

> (argcdr 1 2)
'(2)

> (argcdr 1 '(2 3))
'((2 3))

;; only 3 arguments passed
> (argcar 1 2 3)
1

> (argcar 1 2 '(3))
1

> (argcdr 1 2 3)
'(2 3)

> (argcdr 1 2 '(3))
'(2 (3))

```

Here, the dot (.) in the parameter specifications is being used as the Scheme analog of :: and : in ML and Haskell, respectively, albeit over an entire argument list rather than over an individual list argument as in ML or Haskell. Again, the dot in Scheme cannot be used to decompose individual list arguments:

```

> ((lambda ((x . xs) (y . ys))
  (cons x (cons y '())))) '(1 2) '(3 4))
lambda: not an identifier, identifier with default,
        or keyword in: (x . xs)

```

Again, though transparent, Scheme, like ML and Haskell, also does pattern matching from arguments to parameters. However, in ML and Haskell, individual list arguments can be pattern matched as well. In Scheme, functions can accept only a single list argument, which appears to be restrictive, but means that Scheme functions are flexible and general—they can effectively accept a variable number of arguments. In contrast, any ML or Haskell function can have only one type. If such a function accepted a variable number of parameters, it would have multiple types. Tables 7.4 and 7.5 summarize these nuances of argument lists in Scheme vis-à-vis ML and Haskell.

	Natively		Through Simulation	
	fixed-size	variable-size	fixed-size	variable-size
Scheme	✓ (only one argument)	×	✓ (use .)	✓ (use .)
ML	✓ (one or more arguments)	×	N/A	×
Haskell	✓ (one or more arguments)	×	N/A	×

Table 7.4 Scheme Vis-à-Vis ML and Haskell for Fixed- and Variable-Sized Argument Lists

	Parameter(s) Reception	Single List Arg Decomposition	Example
Scheme	as a list	×	N/A
ML	as a tuple	✓ (use ::)	$x :: xS$
Haskell	as a tuple	✓ (use :)	$x : xS$

Table 7.5 Scheme Vis-à-Vis ML and Haskell for Reception and Decomposition of Argument(s)

Conceptual Exercises for Chapter 7

Exercise 7.1 Explore numeric division in Java (i.e., integer vis-à-vis floating-point division or a mixture of the two). Report your findings.

Exercise 7.2 Is the addition operator (+) overloaded in ML? Explain why or why not.

Exercise 7.3 Explain why the following ML expressions do not type check:

- (a) `false andalso (1 / 0);`
- (b) `false andalso (1 div 0);`
- (c) `false andalso (1 / 2);`
- (d) `false andalso (1 div 2);`

Exercise 7.4 Explain why the following Haskell expressions do not type check:

- (a) `False && (1 / 0)`
- (b) `False && (div 1 0)`
- (c) `False && (1 / 2)`
- (d) `False && (div 1 2)`

Exercise 7.5 Why does integer division in C truncate the fractional part of the result?

Exercise 7.6 Languages with coercion, such as Fortran, C, and C++, are less reliable than those languages with little or no coercion, such as Java, ML, and Haskell. What advantages do languages with coercion offer in return for compromising reliability?

Exercise 7.7 In C++, why is the return type not considered when the compiler tries to resolve (i.e., disambiguate) the call to an overloaded function?

Exercise 7.8 Identify a programming language suitable for each cell in the following table:

	Type safe	Type unsafe
Statically typed		
Dynamically typed		

Exercise 7.9

- (a) Investigate *duck typing* and describe the concept.
- (b) From where does the term *duck typing* derive?
- (c) Is *duck typing* the same concept as *dynamic binding* of messages to methods (based on the type of an object at run-time rather than its declared type) in languages supporting object-oriented programming (e.g., Java and Smalltalk)? Explain.
- (d) Identify three languages that use duck typing.

Exercise 7.10 Suppose we have an ML function `f` with a definition that begins: `fun f(a:int, b, c, d, e)= ....`. State what can be inferred about the types of `b`, `c`, `d`, and/or `e` if the body of the function is each of the following **if-then-else** expressions:

- (a) `if a < b then b+c else d+e`
- (b) `if b < c then d else e`
- (c) `if b < c then d+e else d*e`

Exercise 7.11 Given a function `mystery` with two parameters, the SML-NJ environment produces the following response:

```
val mystery = fn: int list -> int list -> int list
```

List everything you can determine from this type about the definition of `mystery` as well as the ways which it can be invoked.

Exercise 7.12 Consider the following ML function:

```
fun f(g, h) = g(h(g));
```

- (a) What is the type of function f ?
- (b) Is function f *polymorphic*?

Exercise 7.13 Consider the following definition of a merge function in ML:

```
fun merge(l, nil) = l
|   merge(nil, l) = l
|   merge(left as l::ls, right as r::rs) =
    if l < r then l::merge(ls, right)
    else r::merge(left, rs);
```

Explain what in this function definition causes the ML type inference algorithm to deduce its type as:

```
val merge = fn : int list * int list -> int list
```

Exercise 7.14 Explain why the ML function `reverse` (defined in Section 7.9) is *polymorphic*, while the ML function `sum` (also defined in Section 7.9) is not.

Exercise 7.15 Consider the following Scheme code:

```
(define f
  (lambda (x)
    (car x)))

(define f
  (lambda (x y)
    (cons x y)))
```

- (a) Is this an example of function *overloading* or *overriding*?
- (b) Run this program in *DrRacket* with the language set to Racket (i.e., `#lang racket`). Run it with the language set to R5RS (i.e., `#lang r5rs`). What do you notice?
- (c) Is function *overriding* possible without *nested functions*?
- (d) Does JavaScript support function *overloading* or *overriding*, or both? Explain.

Exercise 7.16 Consider the following two ML expressions: $(x+y)$ and `fun f x y = y;`. The first expression is an arithmetic expression and the second expression is a function definition. Which of these expressions involves *polymorphism* and which involves *overloading*? Explain.

7.11 Thematic Takeaways

- Languages using *static type checking* detect nearly all type errors before run-time; languages using *dynamic type checking* delay the detection of most type errors until run-time.
- The use of automatic *type inference* allows a statically typed language to achieve reliability and safety without the burden of having to declare the type of every value or variable:

Static, Safe Type System + Type Inference $\rightsquigarrow$ Reliability/Safety + Manifest Typing

- There are practical trade-offs between *statically* and *dynamically typed* languages—such as other issues in the design and use of programming languages.

7.12 Chapter Summary

In this chapter, we studied language concepts related to types—particularly, type systems and type inference. The *type system* in a programming language broadly refers to the language’s approach to type checking. In a *static type system*, types are checked and almost all type errors are detected before run-time. In a *dynamic type system*, types are checked and most type errors are detected at run-time. Languages with static type systems are said to be *statically typed* or to use *static typing*. Languages with dynamic type systems are said to be *dynamically typed* or to use *dynamic typing*. Reliability, predictability, safety, and ease of debugging are advantages of a statically typed language. Flexibility and efficiency are benefits of using a dynamically typed language. Java, C#, ML, Haskell, and F# are statically typed languages. Python and JavaScript are dynamically typed languages. A *safe type system* does not permit the integrity constraints of types to be deliberately violated (e.g., C#, ML). There are a variety of methods for achieving a degree of flexibility within the confines of a static and safe type system, including *parametric* and *ad hoc polymorphism*, and *type inference*. An *unsafe type system* permits the integrity constraints of types to be deliberately violated (e.g., C/C++). *Explicit typing* requires the type of each variable to be explicitly declared (e.g., C/C++). *Implicit typing* does not require the type of each variable to be explicitly declared (e.g., Python).

The study of typing leads to the exploration of other language concepts related to types: type conversion—type coercion and type casting; type signatures; parametric polymorphism; and function overloading and overriding. Some of these concepts render type safe languages more flexible. *Type conversion* refers to either implicitly or explicitly changing a value from one type to another. *Type coercion* is an implicit conversion where values can deviate from the type required by a function without warning or error because the appropriate conversions are made automatically before or at run-time and are transparent to the programmer. A *type cast* is an explicit conversion that entails interpreting the bit pattern used

to represent a value of a particular type as another type. *Conversion functions* also explicitly convert values from one type to another (e.g., `strtol` in C).

In ML and Haskell, both of which are statically typed languages with first-class functions, functions have types—called *type signatures*—that must be determined before run-time. For instance, the type signature of a function that squares an integer in ML is `int -> int`; this notation indicates that the function maps a domain onto a range. Similarly, the type signature of a function that adds two integers and returns the integer sum in ML is `int * int -> int`. The format of a type signature in ML uses notation indicating that the domain of a function with more than one argument is a Cartesian product of the types (i.e., a set of values) of the individual parameters. Thus, certain functions/operators require values of a particular *monomorphic* type.

Other operators/functions can accept arguments of different types; they are said to have *polymorphic* types. With *parametric polymorphism*, a function can be defined generically so it will handle arguments identically no matter what their type. A polymorphic function type is described using a type signature containing *type variables*. In other words, the types in the type signature are variable. For instance, the Haskell type signature `[a] -> [a]` specifies that a function accepts a list of elements of any type `a` as a parameter and returns a list of elements of type `a`. Any polymorphic function type specifies that any function with this type is any one of multiple monomorphic types. Function *overloading*, in contrast, refers to determining the applicable function definition to bind to a function call, from among a collection of definitions with the same name, based on the number and/or the types of arguments used in the invocation. Thus, parametric polymorphic functions have *one definition* with the *same number of parameters*, whereas overloaded functions have *multiple definitions* each with a *different number* and/or *type of parameters*, and/or *return type*. Function *overriding* occurs when multiple function definitions share the same function name, but only one of the function definitions is visible at any point in the program due to the presence of scope holes. Figure 7.1 presents a hierarchy of these concepts.

While statically typed languages with sound type systems result in programs that can be thoroughly type checked, they often require the programmer to associate an explicit type declaration with each identifier in the program—which inhibits program development and run-time flexibility. *Type inference* refers to the automatic deduction of the type of a value or variable based on context *without* an explicit type declaration. It allows a language to achieve the reliability and safety resulting from a static and sound type system without the burden of having to declare the type of every identifier (i.e., *manifest typing*). Both ML and Haskell use type inference, so they do not require the programmer to declare the type of any variable unless necessary. Both ML and Haskell use the *Hindley–Milner algorithm* for type inference.

Scheme functions can accept only one argument, which is always received as a list. These functions can simulate the reception of a fixed-size argument list containing one or more arguments [e.g., `(x)`, `(x y)`, and so on] or a variable number of arguments [e.g., `x` or `(x . xs)`]. ML and Haskell functions, by

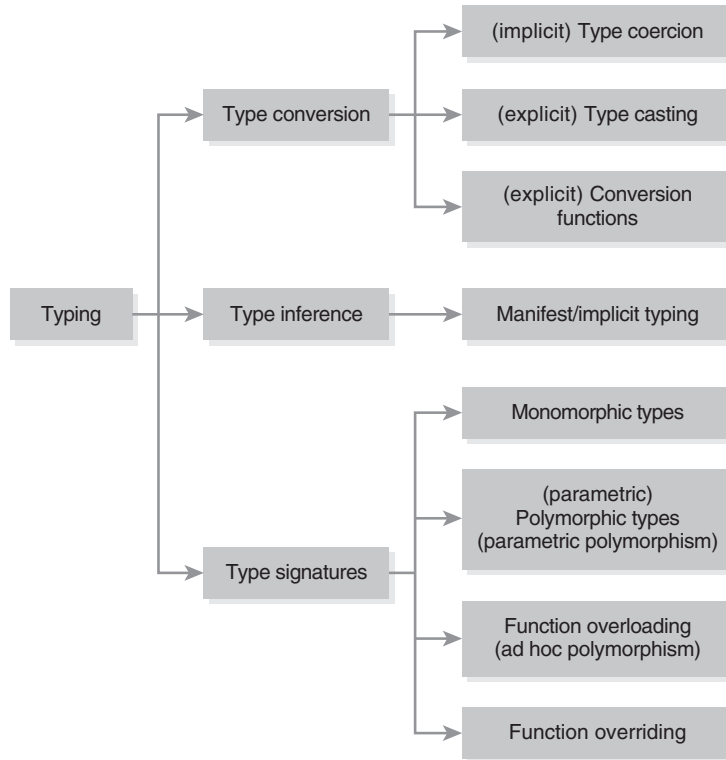


Figure 7.1 Hierarchy of concepts to which the study of typing leads.

contrast, can accept a fixed-size argument tuple containing one or more arguments [e.g., (x) , (x, y) , and so on], but cannot accept a variable number of arguments. (Any function in ML and Haskell must have only one type.) Arguments in ML and Haskell are not received as a list, but rather as a tuple, and any individual list argument can be decomposed using the $::$ and $:$ operators, respectively [e.g., `fun f (x::xs, y::ys) = ...` in ML]. Decomposition of individual list arguments (using dot notation) is not possible in Scheme. The ability of a function to accept a variable number of arguments offers flexibility. Not only does it allow the function to be defined in a general manner, but it also empowers the programmer to implement programming abstractions, which we explore in Chapter 8.

7.13 Notes and Further Reading

The classical type inference algorithm with parametric polymorphism for the λ -calculus used in ML and Haskell is informally referred to as the *Hindley–Milner type inference algorithm* (HM). This algorithm is based on a type inference algorithm, developed by Haskell Curry and Robert Feys in 1958, for the simply typed

λ -calculus. The simply typed λ -calculus ($\lambda \rightarrow$), introduced by Alonzo Church in 1940, is a typed interpretation of the λ -calculus with only one type constructor (i.e., $\rightarrow$) that builds function types. The simply typed λ -calculus is the simplest (and canonical) example of a typed λ -calculus. (The λ -calculus introduced in Chapter 5 is the untyped λ -calculus.) Systems with polymorphic types, including ML and Haskell, are not simply typed.

HM is a practical algorithm and, thus, is used in a variety of programming languages, because it is *complete* (i.e., it always returns an answer), deduces the most *general* type of a given expression without the need for any type declarations or other assistive information, and is *fast* (i.e., it computes a type in near linear time in the size of the source expression). For a succinct overview of the type concepts discussed in this chapter, we refer readers to Wright (2010).

Chapter 8

Currying and Higher-Order Functions

[T]here are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies. The first method is far more difficult.

— Tony Hoare, 1980 ACM A. M. Turing Award Lecture

THE concept of *static typing* leads to *type inference* and *type signatures* for functions (all of which are covered in Chapter 7), which lead to the concepts of *currying* and *partial function application*, which we discuss in this chapter. All of these concepts are integrated in the context of *higher-order functions*, which also provide us with tools and techniques for constructing well-designed and -factored software systems, including interpreters (which we build in Chapters 10—12). The programming languages ML and Haskell are ideal vehicles through which to study and explore these additional typing concepts.

8.1 Chapter Objectives

- Explore the programming language concepts of *partial function application* and *currying*.
- Describe *higher-order functions* and their relationships to curried functions, which together support the development of well-designed, concise, elegant, and reusable software.

8.2 Partial Function Application

The `apply` function in Scheme is a higher-order function that accepts a function f and a list l as arguments, where the elements of l are the individual arguments of

Concept	Function	Type Signature	λ -Calculus
com fun appl	<code>apply</code>	$((a \times b \times c) \rightarrow d) \times a \times b \times c \rightarrow d$	$= \lambda(f, x, y, z).f(x, y, z)$
part fun appl 1	<code>papply1</code>	$((a \times b \times c) \rightarrow d) \times a \rightarrow ((b \times c) \rightarrow d)$	$= \lambda(f, x).\lambda(y, z).f(x, y, z)$
part fun appl n	<code>papply</code>	$((a \times b \times c) \rightarrow d) \times a \rightarrow ((b \times c) \rightarrow d)$	$= \lambda(f, x).\lambda(y, z).f(x, y, z)$
		$((a \times b \times c) \rightarrow d) \times a \times b \rightarrow (c \rightarrow d)$	$= \lambda(f, x, y).\lambda(z).f(x, y, z)$
		$((a \times b \times c) \rightarrow d) \times a \times b \times c \rightarrow (\{\} \rightarrow d)$	$= \lambda(f, x, y, z).\lambda().f(x, y, z)$
currying	<code>curry</code>	$((a \times b \times c) \rightarrow d) \rightarrow (a \rightarrow (b \rightarrow (c \rightarrow d)))$	$= \lambda(f).\lambda(x).\lambda(y).\lambda(z).f(x, y, z)$
uncurrying	<code>uncurry</code>	$(a \rightarrow (b \rightarrow (c \rightarrow d))) \rightarrow ((a \times b \times c) \rightarrow d)$	$= \lambda(f).\lambda l.f(car\ l)(cadr\ l)(caddr\ l)$

Table 8.1 Type Signatures and λ -Calculus for a Variety of Higher-Order Functions. Each signature assumes a ternary function $f: (a \times b \times c) \rightarrow d$. All of these functions except `apply` return a function. In other words, all but `apply` are closed operators.

f , and that applies f to these (individual) arguments and returns the result:

```
> (apply + '(1 2 3))
6
```

This is called *complete function application* because a complete set of arguments is supplied for the parameters to the function. The type signature and λ -calculus for `apply` are given in Table 8.1. The function `eval`, in contrast, evaluates S-expressions representing code in an environment:

```
;; (define f (lambda (x) (cons x ())))
> (define f (list 'lambda '(x) (list 'cons 'x '(quote ())))
> f
'(lambda (x) (cons x '()))
> (eval f)
#<procedure>
> ((eval f) 5)
'(5)
> (eval (list 'lambda '(x) ' (+ x 1)))
#<procedure>
> (eval (list (list 'lambda '(x) ' (+ x 1)) '2))
3
> (eval '(lambda (x) (+ x 1)))
#<procedure>
> (eval '((lambda (x) (+ x 1)) 2))
3
```

Thus, the function `apply` applies a function to arguments and the function `eval` evaluates an expression in environment. The functions `eval` and `apply` are at the heart of any interpreter, as we see in Chapters 10–12.

Partial function application (also called *partial argument application* or *partial function instantiation*), `papply1`, refers to the concept that if a function, which accepts at least one parameter, is invoked with only an argument for its first parameter (i.e., partially applied), it returns a new function accepting the arguments for the remaining parameters; this new function, when invoked with arguments for those parameters, yields the same result as would have been returned had the original function been invoked with arguments for all of its

<pre>(define papply1 (lambda (fun arg) (lambda x (apply fun (cons arg x))))))</pre>	<pre>(define papply (lambda (fun . args) (lambda x (apply fun (append args x))))))</pre>

Table 8.2 Definitions of `papply1` and `papply` in Scheme

parameters (i.e., a complete function application). More formally, with *partial function application*, for any function $f(p_1, p_2, \dots, p_n)$,

$$f(a_1) = g(p_2, p_3, \dots, p_n)$$

such that

$$g(a_2, a_3, \dots, a_n) = f(a_1, a_2, a_3, \dots, a_n)$$

The type signature and λ -calculus for `papply1` are given in Table 8.1. The `papply1` function, defined in Scheme in Table 8.2 (left), accepts a function `fun` and its first argument `arg` and returns a function accepting arguments for the remainder of the parameters. Intuitively, the `papply1` function can partially apply a function with respect to an argument for only its first parameter:

```
> (define add3 (papply1 + 3))
> (add3)
3
> (add3 1)
4
> ((papply1 + 3))
3
> ((papply1 + 3) 1)
4
> (apply (papply1 + 3) '())
3
> (apply (papply1 + 3) '(1))
4
> (define add
  (lambda (x y z)
    (+ x y z)))
> (define add3 (papply1 add 3))
> (add3)
3
> (add3 1 2)
6
> ((papply1 add 3))
3
> ((papply1 add 3) 1 2)
6
> (apply (papply1 add 3) '())
3
> (apply (papply1 add 3) '(1 2))
6
> (define inc
  (lambda (x)
    (+ x 1)))
> (define f (papply1 inc 5))
> (f)
6
> ((papply1 inc 5))
```



```

6
> (apply f '())
6

```

We can generalize partial function application from accepting only the first argument of its input function to accepting arguments for any prefix of the parameters of its input function. Thus, more generally, *partial function application*, `papply`, refers to the concept that if a function, which accepts at least one parameter, is invoked with only arguments for a *prefix* of its parameters (i.e., partially applied), it returns a new function accepting the arguments for the unsupplied parameters; this new function, when invoked with arguments for those parameters, yields the same result as would have been returned had the original function been invoked with arguments for all of its parameters. Thus, more generally, with *partial function application*, for any function $f(p_1, p_2, \dots, p_n)$,

$$f(a_1, a_2, \dots, a_m) = g(p_{m+1}, p_{m+2}, \dots, p_n)$$

where $m \leq n$, such that

$$g(a_{m+1}, a_{m+2}, \dots, a_n) = f(a_1, a_2, \dots, a_m, a_{m+1}, a_{m+2}, \dots, a_n)$$

The type signature and λ -calculus for `papply` are given in Table 8.1. The `papply` function, defined in Scheme in Table 8.2 (right), accepts a function `fun` and arguments for the first n of m parameters to f where $m \leq n$, and returns a function accepting the remainder of the $(n - m)$ parameters. Intuitively, the `papply` function can partially apply a function with respect to arguments for any prefix of its parameters, including all of them:

```

> (define add5 (papply + 3 2))
> (add5)
5
> (add5 1)
6
> ((papply + 3 2))
5
> ((papply + 3 2) 1)
6
> (apply (papply + 3 2) '())
5
> (apply (papply + 3 2) '(1))
6
> (define add6 (papply + 3 2 1))
> (add6)
6
> ((papply + 3 2 1))
6
> (apply (papply + 3 2 1) '())
6
> (define add10 (papply add6 1 1 1 1))
> (add10)
10
> ((papply add6 1 1 1 1))
10
> (apply (papply add6 1 1 1 1) '())
10

```

Thus, the `papply` function subsumes the `papply1` function because the `papply` function generalizes the `papply1` function. For instance, we can replace `papply1` with `papply` in all of the preceding examples:

```
> (define add3 (papply + 3))
> (add3)
3
> (add3 1)
4
> ((papply + 3))
3
> ((papply + 3) 1)
4
> (apply (papply + 3) '())
3
> (apply (papply + 3) '(1))
4
> (define add
>   (lambda (x y z)
>     (+ x y z)))
> (define add3 (papply add 3))
> (add3)
3
> (add3 1 2)
6
> ((papply add 3))
3
> ((papply add 3) 1 2)
6
> (apply (papply add 3) '())
3
> (apply (papply add 3) '(1 2))
6
> (define inc
>   (lambda (x)
>     (+ x 1)))
> (define f (papply inc 5))
> (f)
6
> ((papply inc 5))
6
> (apply f '())
6
```

Partial function application is defined (in `papply1` and `papply`) as a user-defined, higher-order function that accepts a function and arguments for some prefix of its parameters as arguments and returns a new function. Therefore, both definitions of partial function application, `papply1` and `papply`, are *closed*; that is, each accepts a function as input and returns a function as output. They are also general, in that they accept a function of any arity greater than zero as input. The closed nature of both `papply1` and `papply` means that each can be reapplied to its result, and to the result of the other, in a progressive series of applications until one or the other function returns an argumentless function (i.e., until a fixpoint is reached). Also, notice that a single invocation of `papply` can replace a progressive series of calls to `papply1`:

```
> ((papply1 (papply1 (papply1 add 1) 2) 3))
6
> ((papply add 1 2 3))
6
```

Thus, partial function application enables a function to be invoked in n ways, corresponding to all possible prefixes of the function, including a complete function application, where n is the number of parameters of the original, pristine function being partially applied. For instance, the ternary function `add` just defined can be partially applied in four different ways because it has three parameters:

```
> ;; repeatedly partially applying with one argument
> ((papply (papply (papply add 1) 2) 3))
6
> ;; partially applying with one argument followed by two arguments
> ((papply (papply add 1) 2 3))
6
> ;; partially applying with two arguments followed by one argument
> ((papply (papply add 1 2) 3))
6
> ;; partially applying with all three arguments in one stroke
> ((papply add 1 2 3))
6
```

More formally, assuming an n -ary function f , where $n > 0$:

$$\underbrace{\underbrace{\underbrace{\text{papply} (\dots (\text{papply} (\underbrace{\text{papply}(f, 1), 2), \dots), n)}_{\text{argumentless function fixpoint}})}_{(n-1)\text{-ary function}}}_{(n-2)\text{-ary-function}}$$

Each of the following series of progressive applications of `papply1` and `papply` results in the same output:

```
;; three applications
((papply1 (papply1 (papply1 add 1) 2) 3))
((papply1 (papply1 (papply add 1) 2) 3))
((papply1 (papply (papply1 add 1) 2) 3))
((papply1 (papply (papply add 1) 2) 3))
((papply (papply (papply1 add 1) 2) 3))
((papply (papply (papply add 1) 2) 3))
((papply (papply1 (papply1 add 1) 2) 3))
((papply (papply1 (papply add 1) 2) 3))
;; two applications
((papply1 (papply add 1 2) 3))
((papply (papply add 1 2) 3))
((papply1 (papply1 add 1) 2 3))
((papply (papply1 add 1) 2 3))
;; one application
((papply add 1 2 3))
```

Consider a `pow` function defined in Scheme:

```
(define pow
  (lambda (e b)
    (cond
```

```

((eqv? b 0) 0)
((eqv? b 1) 1)
((eqv? e 0) 1)
((eqv? e 1) b)
(else (* b (pow (- e 1) b)))))

```

An alternative approach to partially applying this function without the use of `papply` is to define a function that accepts a function and arguments for a fixed prefix of its parameters and returns an S-expression representing code accepting the remainder of arguments for the remainder of the parameters; this S-expression, when evaluated, returns what the original function would have returned given all of these arguments. Consider the following function `s11`, which does this:

```

1  > (define s11
2    (lambda (f x)
3      (list 'lambda '(y) (list f x 'y))))
4
5  > (pow 2 3)
6  9
7
8  > (define square (s11 pow 2))
9
10 > square
11 (lambda (y) (#<procedure:pow> 2 y))
12
13 > (eval square)
14 #<procedure>
15
16 > ((eval square) 3)
17 9
18
19 > (pow 3 3)
20 27
21
22 > (define cube (s11 pow 3))
23
24 > cube
25 (lambda (y) (#<procedure:pow> 3 y))
26
27 > (eval cube)
28 #<procedure>
29
30 > ((eval cube) 3)
31 27

```

The disadvantages of this approach are the need to explicitly call `eval` (lines 16 and 30) when invoking the residual function and the need to define multiple versions of this function, each corresponding to all possible ways of partially applying a function of n parameters. For instance, partially applying a ternary function in all possible ways (i.e., all possible partitions of parameters) requires functions `s111` (each argument individually), `s12` (first argument individually and last two in one stroke), and `s21` (first two arguments in one stroke and last argument individually). As n increases, the number of functions required combinatorially explodes. However, this approach is advantageous if we desire to restrict the ways in which a function can be partially applied since the function `papply` cannot enforce any restrictions on how a function is partially applied.

Conceptual and Programming Exercises for Section 8.2

Exercise 8.2.1 Reify (i.e., codify) and explain the function returned by the following Scheme expression: `(papply papply papply add 1 2 3)`.

Exercise 8.2.2 Define a function `s21` that enables you to partially apply the following ternary Scheme function `add` using the approach illustrated at the end of Section 8.2 (lines 1–31):

```
(define add
  (lambda (x y z)
    (+ x y z)))
```

Exercise 8.2.3 Define a function `s12` that enables you to partially apply the ternary Scheme function `add` in Programming Exercise 8.2.2 using the approach illustrated at the end of Section 8.2 (lines 1–31).

8.3 Currying

Currying refers to converting an n -ary function into one that accepts only one argument and returns a function, which also accepts only one argument and returns a function that accepts only one argument, and so on. This technique was introduced by Moses Schönfinkel, although the term was coined by Christopher Strachey in 1967 and refers to logician Haskell Curry. For now, we can think of a curried function as one that permits transparent partial function application (i.e., without calling `papply1` or `papply`). In other words, a curried function (or a function written in curried form, as discussed next) can be partially applied without calling `papply1` or `papply`. Later, we see that a curried function is not being partially applied at all.

8.3.1 Curried Form

Consider the following two definitions of a power function (i.e., a function that computes a base b raised to an exponent e , b^e) in Haskell:

```
1  Prelude> :{
2  Prelude| powucf(0, _) = 1
3  Prelude| powucf(1, b) = b
4  Prelude| powucf(_, 0) = 0
5  Prelude| powucf(e, b) = b * powucf(e-1, b)
6  Prelude|
7  Prelude| powcf 0 _ = 1
8  Prelude| powcf 1 b = b
9  Prelude| powcf _ 0 = 0
10 Prelude| powcf e b = b * powcf (e-1) b
11 Prelude| :}
```

These definitions are almost the same. Notice that the definition of the `powucf` function has a comma between each parameter in the tuple of parameters, and that tuple is enclosed in parentheses; conversely, there are no commas and parentheses in the parameters tuple in the definition of the `powcf` function. As a result, the types of these functions are different.

```
12 Prelude> :type powucf
13 powucf :: (Num a, Num b, Eq a, Eq b) => (a, b) -> b
14 Prelude>
15 Prelude> :type powcf
16 powcf :: (Num t1, Num t2, Eq t1, Eq t2) => t1 -> t2 -> t2
```

The type of the `powucf` function states that it accepts a tuple of values of a type in the `Num` class and returns a value of a type in the `Num` class. In contrast, the type of the `powcf` function indicates that it accepts a value of a type in the `Num` class and returns a function mapping a value of a type in the `Num` class to a value of the same type in the `Num` class. The definition of `powcf` is written in *curried form*, meaning that it accepts only one argument and returns a function, also with only one argument:

```
17 Prelude> square = powcf 2
18 Prelude>
19 Prelude> :type square
20 square :: (Num t2, Eq t2) => t2 -> t2
21 Prelude>
22 Prelude> cube = powcf 3
23 Prelude>
24 Prelude> :type cube
25 cube :: (Num t2, Eq t2) => t2 -> t2
26 Prelude>
27 Prelude> (powcf 2) 3
28 9
29 Prelude> square 3
30 9
31 Prelude> (powcf 3) 3
32 27
33 Prelude> cube 3
34 27
```

By contrast, the definition of `powucf` is written in *uncurried form*, meaning that it must be invoked with arguments for all of its parameters with parentheses around the argument list and commas between individual arguments. In other words, `powucf` cannot be partially applied, without the use of `papply1` or `papply`, but rather must be completely applied:

```
35 Prelude> powucf(2,3)
36 9
37 Prelude>
38 Prelude> powucf(2)
39
40 <interactive>:36:1: error:
41   Non type-variable argument in the constraint: Num (a, b)
42   (Use FlexibleContexts to permit this)
43   When checking the inferred type
44     it :: forall a b. (Eq a, Eq b, Num a, Num b, Num (a, b)) => b
```

```

45 Prelude>
46 Prelude> powucf 2
47
48 <interactive>:38:1: error:
49   Non type-variable argument in the constraint: Num (a, b)
50   (Use FlexibleContexts to permit this)
51   When checking the inferred type
52     it :: forall a b. (Eq a, Eq b, Num a, Num b, Num (a, b)) => b

```

In these function applications, notice the absence of parentheses and commas when invoking the curried function and the presence of parentheses and commas when invoking the uncurried function. These syntactic differences are not stylistic; they are required. Parentheses and commas must not be included when invoking a curried function, while parentheses and commas must be included when invoking an uncurried function:

```

53 Prelude> powcf (2,3)
54
55 <interactive>:42:1: error:
56   Could not deduce (Num (Integer, Integer))
57   arising from a use of 'powcf'
58   from the context: (Eq t2, Num t2)
59   bound by the inferred type of it :: (Eq t2, Num t2) => t2 -> t2
60   at <interactive>:42:1-10
61   In the expression: powcf (2, 3)
62   In an equation for 'it': it = powcf (2, 3)
63 Prelude> powucf 2 3
64
65 <interactive>:43:1: error:
66   Non type-variable argument in the constraint: Eq (t1 -> t2)
67   (Use FlexibleContexts to permit this)
68   When checking the inferred type
69     it :: forall a t1 t2.
70         (Eq a, Eq (t1 -> t2), Num a, Num t1, Num (t1 -> t2),
71          Num (a, t1 -> t2)) =>
72         t2

```

These examples bring us face-to-face with the fact that Haskell (and ML) perform literal pattern matching from function arguments to parameters (i.e., the parentheses and commas must also match).

8.3.2 Currying and Uncurrying

In general, currying transforms a function $f_{\text{uncurried}}$ with the type signature

$$(p_1 \times p_2 \times \cdots \times p_n) \rightarrow r$$

into a function f_{curried} with the type signature

$$p_1 \rightarrow (p_2 \rightarrow (\cdots \rightarrow (p_n \rightarrow r) \cdots))$$

such that

$$f_{\text{uncurried}}(a_1, a_2, \cdots, a_n) = (\cdots ((f_{\text{curried}}(a_1))(a_2)) \cdots)(a_n)$$

Currying $f_{\text{uncurried}}$ and running the resulting f_{curried} function has the same effect as progressively partially applying $f_{\text{uncurried}}$. Inversely, uncurrying transforms a function f_{curried} with the type signature

$$p_1 \rightarrow (p_2 \rightarrow (\cdots \rightarrow (p_n \rightarrow r) \cdots))$$

into a function $f_{\text{uncurried}}$ with the type signature

$$(p_1 \times p_2 \times \cdots \times p_n) \rightarrow r$$

such that

$$f_{\text{uncurried}}(a_1, a_2, \cdots, a_n) = (\cdots ((f_{\text{curried}}(a_1))(a_2)) \cdots)(a_n)$$

8.3.3 The `curry` and `uncurry` Functions in Haskell

The built-in Haskell functions `curry` and `uncurry` are used to convert a binary function between uncurried and curried forms:

```

73 Prelude> :type curry
74 curry :: ((a,b) -> c) -> a -> b -> c
75 Prelude>
76 Prelude> :type uncurry
77 uncurry :: (a -> b -> c) -> (a,b) -> c
78 Prelude>
79 Prelude> powcf2 = curry powucf
80 Prelude>
81 Prelude> powucf2 = uncurry powcf
82 Prelude>
83 Prelude> square2 = powcf2 2
84 Prelude>
85 Prelude> cube2 = powcf2 3
86 Prelude>
87 Prelude> ((curry powucf) 2) 3
88 9
89 Prelude> curry powucf 2 3
90 9
91 Prelude> (uncurry powcf) (2,3)
92 9
93 Prelude> uncurry powcf (2,3)
94 9
95 Prelude> ((curry powucf) 3) 3
96 27
97 Prelude> curry powucf 3 3
98 27
99 Prelude> (uncurry powcf) (3,3)
100 27
101 Prelude> uncurry powcf (3,3)
102 27
103 Prelude> :type powucf2
104 powucf2 :: (Num t1, Num c, Eq t1, Eq c) => (t1, c) -> c
105 Prelude>
106 Prelude> :type powcf2
107 powcf2 :: (Num a, Num c, Eq a, Eq c) => a -> c -> c
108 Prelude>
109 Prelude> :type square2
110 square2 :: (Num c, Eq c) => c -> c

```



```

111 Prelude>
112 Prelude> square2 3
113 9
114 Prelude>
115 Prelude> :type cube2
116 cube2 :: (Num c, Eq c) => c -> c
117 Prelude>
118 Prelude> cube 3
119 27

```

Currying and uncurrying are defined as higher-order functions (i.e., `curry` and `uncurry`, respectively) that each accept a function as an argument and return a function as a result (i.e., they are *closed* functions). In Haskell, the built-in function `curry` can accept only an uncurried binary function with type $(a,b) \rightarrow c$ as input. Similarly, the built-in function `uncurry` can accept only a curried function with type $a \rightarrow b \rightarrow c$ as input. The type signatures and λ -calculus for the functions `curry` and `uncurry` are given in Table 8.1. Definitions of `curry` and `uncurry` for binary functions in Haskell are given in Table 8.3. Notice that the definitions of `curry` and `uncurry` in Haskell are written in *curried* form. (Programming Exercises 8.3.22 and 8.3.23 involve defining `curry` and `uncurry`, respectively, in *uncurried* form in Haskell for binary functions.) Definitions of `curry` and `uncurry` for binary functions in Scheme are given in Table 8.4 and applied in the following examples:

```

> ((curry pow 2) 3)
9
> (define square (curry pow 2))
> (square 3)
9
> ((curry pow 3) 3)
27
> (define cube (curry pow 3))
> (cube 3)
27
> (curry (lambda (x y) (+ x y)))
#<procedure>
> ((curry (lambda (x y) (+ x y))) 1)
#<procedure>
> (((curry (lambda (x y) (+ x y))) 1) 2)
3
> ((curry (lambda (x y) (+ x y))) 1 2)
. . #<procedure>: expects 1 argument, given 2: 1 2
> (uncurry (lambda (x) (lambda (y) (+ x y))))
#<procedure>
> ((uncurry (lambda (x) (lambda (y) (+ x y)))) 1 2)
3
> (((uncurry (lambda (x) (lambda (y) (+ x y)))) 1) 2)
. . cadr: expects argument of type <cadrable value>; given (1)

```

curry :: $((a,b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$	uncurry :: $(a \rightarrow b \rightarrow c) \rightarrow ((a,b) \rightarrow c)$
curry f a b = f (a,b)	uncurry f (a,b) = f a b

Table 8.3 Definitions of `curry` and `uncurry` in Curried Form in Haskell for Binary Functions

<pre>(define curry (lambda (fun_ucf) (lambda (x) (lambda (y) (fun_ucf x y))))))</pre>	<pre>(define uncurry (lambda (fun_cf) (lambda args ; (x y) ((fun_cf (car args)) (cadr args))))))</pre>

Table 8.4 Definitions of `curry` and `uncurry` in Scheme for Binary Functions

A function that accepts only one argument is neither uncurried or curried. Therefore, we can only curry a function that accepts at least two arguments. User-defined and built-in functions in Haskell that accept only one argument can be invoked with or without parentheses around that single argument:

```
120 Prelude> f x = x
121 Prelude>
122 Prelude> f 1
123 1
124 Prelude> f(1)
125 1
```

More generally, when a function is defined in curried form (or is curried), parentheses can be placed around any individual argument:

```
126 Prelude> add x y z = x + y + z
127 Prelude>
128 Prelude> :type add
129 add :: Num a => a -> a -> a -> a
130 Prelude>
131 Prelude> add 1 2 3
132 6
133 Prelude> add 1 (2) 3
134 6
135 Prelude> add (1) 2 (3)
136 6
137 Prelude> add (1) (2) (3)
138 6
```

The functions `papply1`, `papply`, `curry`, and `uncurry` are *closed*: Each accepts a function as input and returns a function as output. It is necessary, but not sufficient, for a function to be closed to be able to be reapplied to its result. For instance, `curry` and `uncurry` are both closed, but neither can be reapplied to its own result. The functions `papply1` and `papply` are closed, however, so each can be reapplied to its result as demonstrated previously.

8.3.4 Flexibility in Curried Functions

Technically, we do not and cannot partially apply a curried function because a curried function accepts only one argument. (This is why the functions `papply1` and `papply` are not used.) Instead, we simply invoke a curried function in a manner conforming to its type, as with any other function. It just so happens that any curried function, and any function it returns, and any function which that function

returns, and so on, accept only one argument. Therefore, with respect to its uncurried version, invoking a curried function appears to correspond to partially applying it, and partially applying its result, and so on. Consider the following definitions of a ternary addition function in uncurried and curried forms in Haskell:

```

139 Prelude> adducf(x,y,z) = x + y + z
140 Prelude>
141 Prelude> :type adducf
142 adducf :: Num a => (a, a, a) -> a
143 Prelude>
144 Prelude> addcf x y z = x + y + z
145 Prelude>
146 Prelude> :type addcf
147 addcf :: Num a => a -> a -> a -> a

```

While the function `adducf` can only be invoked one way [i.e., with the same number and types of arguments; e.g., `adducf(1, 2, 3)`], the function `addcf` can *effectively* be invoked in the following ways, including the one and only way the type of `adducf` specifies it must be invoked (i.e., with only one argument, as in the first invocation here):

```

addcf 1
addcf 1 2
addcf 1 2 3

```

Because the type of `addcf` is `Num a => a -> a -> a -> a`, we know it can accept only one argument. However, the second and third invocations of `addcf` just given make it appear as if it can accept two or three arguments as well. The absence of parentheses for precedence makes this illusion stronger. Let us consider the third invocation of `addcf`—that is, `addcf 1 2 3`. The `addcf` function is called as required with only one argument (`addcf 1`), which returns a new, unnamed function that is then implicitly invoked with one argument (`<first returned proc> 2` or *`addcf'`* 2), which returns another new, unnamed function, which is then implicitly invoked with one argument (`<second returned proc> 3` or *`addcf''`* 3) and returns the sum 6. Using parentheses to make the implied precedence salient, the expression `addcf 1 2 3` is evaluated as `((addcf 1) 2) 3`:

$$\begin{array}{c}
 \text{addcf''} \\
 \underbrace{\hspace{1.5cm}} \\
 \text{addcf'} \\
 \underbrace{\hspace{1.5cm}} \\
 \text{addcf 1 2 3} = ((\text{addcf 1}) 2) 3
 \end{array}
 \qquad
 \begin{array}{c}
 \text{addcf''} \\
 \underbrace{\hspace{1.5cm}} \\
 \text{addcf'} \\
 \underbrace{\hspace{1.5cm}} \\
 \text{addcf 1 2 3} = ((\text{addcf 1}) 2) 3
 \end{array}$$

Thus, even though a function written in curried form (e.g., `addcf`) can appear to be invoked with more than one argument (e.g., `addcf 1 2 3`), it can never accept more than one argument because the type of a curried function (or a function written in curried form) specifies that it must accept only one argument (e.g., `Num a => a -> a -> a -> a`).

The omission of superfluous parentheses for precedence in an invocation of a curried function must not be confused with the required absence of parentheses around the list of arguments:

```

148 Prelude> -- works without optional parentheses for precedence
149 Prelude> addcf 1 2 3
150 6
151 Prelude> -- works, but optional parentheses for precedence superfluous
152 Prelude> ((addcf 1) 2) 3)
153 6
154 Prelude> -- does not work; parentheses and commas must be omitted
155 Prelude> addcf(1, 2, 3)
156
157 <interactive>:7:1: error:
158   Non type-variable argument in the constraint: Num (a, b, c)
159   (Use FlexibleContexts to permit this)
160   When checking the inferred type
161     it :: forall a b c.
162         (Num a, Num b, Num c, Num (a, b, c)) =>
163         (a, b, c) -> (a, b, c) -> (a, b, c)

```

Moreover, notice that in Haskell (and ML) an open parenthesis to the immediate right of the returned function is not required to force its implicit application, as is required in Scheme:

```

164 Prelude> addcf 1 2 3 -- without optional parentheses
165 6
166 Prelude> ((addcf 1) 2) 3) -- with optional parentheses
167 6

;; with parentheses; parentheses required
> ((papply (papply (papply add 1) 2) 3))
6
;; without parentheses; parentheses required
;; does not work as expected when parentheses omitted
> (papply papply papply add 1 2 3)
#<procedure>

```

It is important to understand that the outermost parentheses around the Scheme expression `((papply (papply (papply add 1) 2) 3))` are needed to force the application of the returned function, and not for precedence.

A curried function is more flexible than its uncurried analog because it can *effectively* be invoked in n ways, where n is the number of arguments its uncurried analog accepts:

- the one and only way its uncurried analog is invoked (i.e., with all arguments as a complete application)
- the one and only way it itself can be invoked (i.e., with only one argument)
- $n - 2$ other ways corresponding to implicit partial applications of each returned function

More generally, if a curried function, whose uncurried analog accepts more than one parameter, is invoked with only arguments for a *prefix* of the parameters of its uncurried analog, it returns a new function accepting the arguments for the parameters of the uncurried analog whose arguments were left unsupplied; that new function, when invoked with arguments for those parameters, yields the same result as would have been returned had the original, uncurried function been invoked with arguments for all of its parameters. Thus, akin to partial function

application, the invocation of a curried definition of a function $f(p_1, p_2, \dots, p_n)$ with arguments for a prefix of its parameters is

$$f(a_1, a_2, \dots, a_m) = g(p_{m+1}, p_{m+2}, \dots, p_n)$$

where $m \leq n$, such that

$$g(a_{m+1}, a_{m+2}, \dots, a_n) = f(a_1, a_2, \dots, a_m, a_{m+1}, a_{m+2}, \dots, a_n)$$

Thus, any curried function can *effectively* be invoked with arguments for any prefix, including all of the parameters of its uncurried analog, without parentheses around the list of arguments or commas between individual arguments:

```
168 Prelude> powcf 2 3
169 9
```

It might appear as if the complete application of an uncurried function is supported through its curried version, but it is not. Rather, the complete application is simulated, albeit transparently to the programmer, by a series of transparent, progressive partial function applications—one for the number of the parameters that the uncurried version of the function accepts—until a final result (i.e., an argumentless fixpoint function) is returned.

Given any uncurried, n -ary function f , currying supports—in a single function without calls to `papply1` or `papply`—all n ways by which f can be partially applied and re-partially applied, and so on. For instance, given the ternary, uncurried Scheme function `add`, the function returned by the expression `(curry add)` supports the following three ways of partially and re-partially applying `add`:

```
> ;; each argument individually
> ((papply (papply (papply add 1) 2) 3))
6
> ;; two arguments followed by one
> ((papply (papply add 1 2) 3))
6
> ;; all three arguments in one stroke
> ((papply add 1 2 3))
6
> (((curry add) 1) 2) 3)
6
```

In summary, any function accepting one or more arguments can be partially applied using `papply1` and `papply`. Any curried function or any function written in curried form can be *effectively* partially applied without the use of the functions `papply1` or `papply`. The advantage of partial function application is that it can be used with any function of any arity greater than zero even if the source code for the function to be partially applied is not available (e.g., in the case of a built-in function such as `map` in ML). The disadvantage of partial function application is that we must call the function `papply1` or `papply` to partially apply a function, and this can get cumbersome and error prone, especially when re-partially applying the result of a partial application, and so on. The

advantage of a curried function or a function written in curried form is that calls to `papply1` or `papply` are unnecessary, so the effective partial function application is transparent. The disadvantage is that the function to be partially applied must be curried or written in curried form, and the function `curry` in Haskell only accepts a binary function. If we want to partially apply a function whose arity is greater than 2, we have two options. We can define it in curried form, which is not possible if its source code is unavailable. We can also define a version of `curry` that accepts a function with the same arity of the function we desire to curry. The latter approach is taken with the definition of `curry` in λ -calculus for a ternary function given in Table 8.1 and the definition of a function capable of currying a 4-ary function:

```
(define curry4ary
  (lambda (f)
    (lambda (a)
      (lambda (b)
        (lambda (c)
          (lambda (d)
            (f a b c d)))))))
```

We can build general `curry` and `uncurry` functions that accept functions of any arity greater than 1, called *implicit currying*, through the use of Scheme macros, which we do not discuss here.

8.3.5 All Built-in Functions in Haskell Are Curried

All built-in Haskell functions are curried. This is why Haskell is referred to as a fully curried language. This is not the case in ML (Section 8.3.7 and Section 8.4). Built-in functions in Haskell that accept only one argument (e.g., `even` or `odd`) are neither uncurried nor curried and can be invoked with or without parentheses around their single argument:

```
Prelude> :type even
even :: Integral a => a -> Bool
Prelude> even (2)
True
Prelude> even 2
True
```

Since all functions built into Haskell are curried, in online Appendix C we do not use parentheses around the argument tuples (or commas between individual arguments) when invoking built-in Haskell functions. For instance, consider our final definition of `mergesort` in Haskell given in online Appendix C:

```
1 Prelude> :{
2 Prelude| mergesort(_, []) = []
3 Prelude| mergesort(_, [x]) = [x]
4 Prelude| mergesort(compop, lat) =
5 Prelude|     let
6 Prelude|         mergesort1([]) = []
7 Prelude|         mergesort1([x]) = [x]
8 Prelude|         mergesort1(lat1) =
```

```

9  Prelude|          let
10 Prelude|          split ([]) = ([], [])
11 Prelude|          split ([x]) = ([], [x])
12 Prelude|          split (x:y:excess) =
13 Prelude|              let
14 Prelude|                  (left, right) = split (excess)
15 Prelude|              in
16 Prelude|                  (x:left, y:right)
17 Prelude|
18 Prelude|          merge (l, []) = l
19 Prelude|          merge ([], r) = r
20 Prelude|          merge (l:ls, r:rs) =
21 Prelude|              if compop (l, r) then l:merge (ls, r:rs)
22 Prelude|              else r:merge (l:ls, rs)
23 Prelude|
24 Prelude|          -- split it
25 Prelude|          (left, right) = split (lat1)
26 Prelude|
27 Prelude|          -- mergesort each side
28 Prelude|          leftsorted = mergesort1 (left)
29 Prelude|          rightsorted = mergesort1 (right)
30 Prelude|          in
31 Prelude|          -- merge
32 Prelude|          merge (leftsorted, rightsorted)
33 Prelude|      in
34 Prelude|      mergesort1 (lat)
35 Prelude| :}
36 Prelude>
37 Prelude> :type mergesort
38 mergesort :: ((a, a) -> Bool, [a]) -> [a]

```

Neither the `mergesort` function nor the `compop` function is curried. Thus, we cannot pass in the built-in `<` or `>` operators, because they are curried:

```

39 Prelude> :type (<)
40 (<) :: Ord a => a -> a -> Bool
41 Prelude>
42 Prelude> :type (>)
43 (>) :: Ord a => a -> a -> Bool

```

When passing an operator as an argument to a function, the passed operator must be a prefix operator. Since the operators `<` and `>` are infix operators, we cannot pass them to this version of `mergesort` without first converting them to prefix operators. We can convert an infix operator to a prefix operator either by wrapping it in a user-defined function or by enclosing it within parentheses:

```

44 Prelude> :type (+)
45 (+) :: Num a => a -> a -> a
46 Prelude>
47 Prelude> (+) 7 2
48 9
49 Prelude> add1 = (+) 1
50 Prelude>
51 Prelude> :type add1
52 add1 :: Num a => a -> a
53 Prelude>
54 Prelude> add1 9
55 10

```

This is why we wrapped these built-in, curried functions around uncurried, anonymous, user-defined functions when invoking `mergesort`:

```
56 Prelude> mergesort((\ (x,y) -> (x<y)), [9,8,7,6,5,4,3,2,1])
57 [1,2,3,4,5,6,7,8,9]
58 Prelude>
59 Prelude> mergesort((\ (x,y) -> (x>y)), [1,2,3,4,5,6,7,8,9])
60 [9,8,7,6,5,4,3,2,1]
```

However, we can use the `uncurry` function to simplify these invocations:

```
61 Prelude> mergesort((uncurry (<)), [9,8,7,6,5,4,3,2,1])
62 [1,2,3,4,5,6,7,8,9]
63 Prelude>
64 Prelude> mergesort((uncurry (>)), [1,2,3,4,5,6,7,8,9])
65 [9,8,7,6,5,4,3,2,1]
```

We cannot pass in one of the built-in, curried Haskell comparison operators [e.g., `(<)` or `(>)`] as is to `mergesort` without causing a type error:

```
66 Prelude> mergesort((<), [9,8,7,6,5,4,3,2,1])
67
68 <interactive>:61:11: error:
69   Couldn't match type '(a, a) -> Bool' with 'Bool'
70   Expected type: (a, a) -> Bool
71   Actual type: (a, a) -> (a, a) -> Bool
72   Probable cause: '(<)' is applied to too few arguments
73   In the expression: (<)
74   In the first argument of 'mergesort', namely
75     '(<), [9, 8, 7, 6, ....]'
```

```
76   In the expression: mergesort (<), [9, 8, 7, 6, ....])
77   Relevant bindings include it :: [a] (bound at <interactive>:61:1)
78
79 Prelude> mergesort(>), [1,2,3,4,5,6,7,8,9])
80
81 <interactive>:63:11: error:
82   Couldn't match type '(a, a) -> Bool' with 'Bool'
83   Expected type: (a, a) -> Bool
84   Actual type: (a, a) -> (a, a) -> Bool
85   Probable cause: '(>)' is applied to too few arguments
86   In the expression: (>)
87   In the first argument of 'mergesort', namely
88     '(>), [1, 2, 3, 4, ....]'
```

```
89   In the expression: mergesort (>), [1, 2, 3, 4, ....])
90   Relevant bindings include it :: [a] (bound at <interactive>:63:1)
```

For this version of `mergesort` to accept one of the built-in, curried Haskell comparison operators as a first argument, we must replace the subexpression `compop(l, r)` in line 21 of the definition of `mergesort` with `(compop l r)`; that is, we must call `compop` without parentheses and a comma. This changes the type of `mergesort` from `((a, a) -> Bool, [a]) -> [a]` to `(a -> a -> Bool, [a]) -> [a]`:

```
91 Prelude> :type mergesort
92 mergesort :: (a -> a -> Bool, [a]) -> [a]
```


While this simple change causes the following invocations to work, we are mixing curried and uncurried functions. Specifically, the function `mergesort` is uncurried, while the function `compop` is curried:

```
93 Prelude> mergesort(<), [9,8,7,6,5,4,3,2,1])
94 [1,2,3,4,5,6,7,8,9]
95 Prelude>
96 Prelude> mergesort(>), [1,2,3,4,5,6,7,8,9])
97 [9,8,7,6,5,4,3,2,1]
```

Of course, now the following invocations no longer work, as expected:

```
98 Prelude> mergesort((\ (x,y) -> (x<y)), [9,8,7,6,5,4,3,2,1])
99
100 <interactive>:39:23: error:
101   Couldn't match expected type '(a, a) -> Bool'
102         with actual type 'Bool'
103   Possible cause: '<>' is applied to too many arguments
104   In the expression: (x < y)
105   In the expression: (\ (x, y) -> (x < y))
106   In the first argument of 'mergesort', namely
107     '(\ (x, y) -> (x < y)), [9, 8, 7, 6, ....]'
108   Relevant bindings include
109     y :: a (bound at <interactive>:39:16)
110     x :: a (bound at <interactive>:39:14)
111     it :: [(a, a)] (bound at <interactive>:39:1)
112
113 Prelude> mergesort((\ (x,y) -> (x>y)), [1,2,3,4,5,6,7,8,9])
114 <interactive>:40:23: error:
115   Couldn't match expected type '(a, a) -> Bool'
116         with actual type 'Bool'
117   Possible cause: '>' is applied to too many arguments
118   In the expression: (x > y)
119   In the expression: (\ (x, y) -> (x > y))
120   In the first argument of 'mergesort', namely
121     '(\ (x, y) -> (x > y)), [1, 2, 3, 4, ....]'
122   Relevant bindings include
123     y :: a (bound at <interactive>:40:16)
124     x :: a (bound at <interactive>:40:14)
125     it :: [(a, a)] (bound at <interactive>:40:1)
```

Since all built-in Haskell functions are curried, we recommend consistently using only curried functions in a Haskell program. With a curried function we do not lose the ability to completely apply a function and we gain the flexibility and power that come with curried functions. Although uniformity is not required (i.e., it is akin to using consistent indentation to make a program more readable and reveal intended semantics), we recommend only using all curried functions or all uncurried functions in a Haskell (or ML) program. We recommend the former in Haskell since all built-in functions in Haskell are curried and since curried functions provide flexibility. Being consistent in using either all curried or all uncurried functions provides uniformity, helps avoid confusion, reduces program and type complexity, and reduces the scope for type errors. Following this guideline is challenging in ML because not all built-in functions are curried in ML; therefore, when defining functions in curried form in ML that call those uncurried built-in functions (e.g., `Int.+ : int * int -> int`

or `String.sub : string * int -> char`), mixing the two forms is unavoidable.

The function `mergesort` is an ideal candidate for currying because by applying it in curried form with the `<` or `>` operators, we get back ascending-sort and descending-sort functions, respectively:

```

126 Prelude> (curry mergesort) (<) [9,8,7,6,5,4,3,2,1]
127 [1,2,3,4,5,6,7,8,9]
128 Prelude>
129 Prelude> ascending_sort = (curry mergesort) (<)
130 Prelude>
131 Prelude> :type ascending_sort
132 ascending_sort :: Ord a => [a] -> [a]
133 Prelude>
134 Prelude> ascending_sort [9,8,7,6,5,4,3,2,1]
135 [1,2,3,4,5,6,7,8,9]
136 Prelude>
137 Prelude> (curry mergesort) (>) [1,2,3,4,5,6,7,8,9]
138 [9,8,7,6,5,4,3,2,1]
139 Prelude>
140 Prelude> descending_sort = (curry mergesort) (>)
141 Prelude>
142 Prelude> :type descending_sort
143 descending_sort :: Ord a => [a] -> [a]
144 Prelude>
145 Prelude> descending_sort [1,2,3,4,5,6,7,8,9]
146 [9,8,7,6,5,4,3,2,1]

```

The following is the final, fully curried version of `mergesort` in curried form:

```

1  Prelude> :{
2  Prelude| mergesort _ [] = []
3  Prelude| mergesort _ [x] = [x]
4  Prelude| mergesort compop lat =
5  Prelude|     let
6  Prelude|         mergesort1 [] = []
7  Prelude|         mergesort1 [x] = [x]
8  Prelude|         mergesort1 lat1 =
9  Prelude|             let
10 Prelude|                 split [] = ([], [])
11 Prelude|                 split [x] = ([], [x])
12 Prelude|                 split (x:y:excess) =
13 Prelude|                     let
14 Prelude|                         (left, right) = split excess
15 Prelude|                     in
16 Prelude|                         (x:left, y:right)
17 Prelude|
18 Prelude|                 merge l [] = l
19 Prelude|                 merge [] l = l
20 Prelude|                 merge (l:ls) (r:rs) =
21 Prelude|                     if compop l r then l:(merge ls (r:rs))
22 Prelude|                     else r:(merge (l:ls) rs)
23 Prelude|
24 Prelude|                 -- split it
25 Prelude|                 (left, right) = split lat1
26 Prelude|
27 Prelude|                 -- mergesort each side
28 Prelude|                 leftsorted = mergesort1 left
29 Prelude|                 rightsorted = mergesort1 right
30 Prelude|     in

```

```

31 Prelude|           -- merge
32 Prelude|           merge leftsorted rightsorted
33 Prelude|   in
34 Prelude|           mergesort1 lat
35 Prelude| :}
36 Prelude>
37 Prelude> :type mergesort
38 mergesort :: (a -> a -> Bool) -> [a] -> [a]
39 Prelude>
40 Prelude> mergesort (<) [9,8,7,6,5,4,3,2,1]
41 [1,2,3,4,5,6,7,8,9]
42 Prelude>
43 Prelude> ascending_sort = mergesort (<)
44 Prelude>
45 Prelude> :type ascending_sort
46 ascending_sort :: Ord a => [a] -> [a]
47 Prelude>
48 Prelude> ascending_sort [9,8,7,6,5,4,3,2,1]
49 [1,2,3,4,5,6,7,8,9]
50 Prelude>
51 Prelude> mergesort (>) [1,2,3,4,5,6,7,8,9]
52 [9,8,7,6,5,4,3,2,1]
53 Prelude>
54 Prelude> descending_sort = mergesort (>)
55 Prelude>
56 Prelude> :type descending_sort
57 descending_sort :: Ord a => [a] -> [a]
58 Prelude>
59 Prelude> descending_sort [1,2,3,4,5,6,7,8,9]
60 [9,8,7,6,5,4,3,2,1]

```

Using `compop` with `mergesort` demonstrates why in Haskell it is advantageous for purposes of uniformity to define all functions in curried form. That uniformity is a challenge to achieve in ML because not all built-in functions in ML are curried. For instance, the function `Int.+ : int * int -> int` is built into ML and uncurried, while the function `map : ('a -> 'b) -> 'a list -> 'b list` is built into ML and curried. Thus, defining a curried function that uses some built-in, uncurried ML functions leads to a mixture of curried and uncurried functions. In summary, four different types of `mergesort` are possible:

```

([a], (a,a) -> Bool) -> [a] -- mergesort uncurried, compop uncurried
([a], a -> a -> Bool) -> [a] -- mergesort uncurried, compop curried
((a,a) -> Bool) -> [a] -> [a] -- mergesort curried, compop uncurried
(a -> a -> Bool) -> [a] -> [a] -- mergesort curried, compop curried

```

The first and last types are recommended (for purposes of uniformity) and the last type is preferred.

A consequence of all functions being fully curried in Haskell is that sometimes we must use parentheses to group syntactic entities. (We can think of this practice as forcing order or precedence, though that is not entirely true in Haskell; see Chapter 12.) For instance, in the expression `isDigit (head "string")`, the parentheses around `head "string"` are required to indicate that the entire argument to `isDigit` is `head "string"`. Omitting these parentheses, as in `isDigit head "string"`, causes the `head` function to be passed to the function `isDigit`, with the argument `"string"` then being passed to the result.

In this case, enclosing the single argument head "string" in parentheses is not the same as enclosing the entire argument tuple in parentheses [i.e., `isDigit (head "string")` is not the same as `isDigit ('s')`] because the former expression will generate an error without the parentheses and the latter does not. In other words, `isDigit head "string"` is incorrect and does not work while `isDigit 's'` is fine:

```
Prelude> import Data.Char
Data.Char> :type isDigit
isDigit :: Char -> Bool
Data.Char> isDigit('s')
False
Data.Char> isDigit 's'
False
Data.Char> :type head
head :: [a] -> a
Data.Char> isDigit (head "string")
False
Data.Char> isDigit head "string"
ERROR - Type error in application
*** Expression      : isDigit head "string"
*** Term            : isDigit
*** Type            : Char -> Bool
*** Does not match : a -> b -> c
```

Moreover, and more importantly, curried functions open up new possibilities in programming, especially with respect to higher-order functions, as we will see in Section 8.4.

8.3.6 Supporting Curried Form Through First-Class Closures

Any language with first-class closures can be used to define functions in curried form. For instance, given that Haskell has first-class closures, even if Haskell did not have a syntax for curried form, we can define a function in curried form:

```
Prelude> :{
Prelude| pow e = (\b -> if e == 0 then 1 else
Prelude|                  if e == 1 then b else
Prelude|                  if b == 0 then 0 else
Prelude|                  b*(pow (e-1) b))
Prelude| :}
Prelude>
Prelude> :type pow
pow :: (Num t1, Num t2, Eq t1, Eq t2) => t1 -> t2 -> t2
Prelude>
Prelude> pow 2 3
9
Prelude>
Prelude> square = pow 2
Prelude>
Prelude> :type square
square :: (Num t2, Eq t2) => t2 -> t2
Prelude>
Prelude> square 3
9
Prelude>
Prelude> pow 3 3
```

```

27
Prelude> cube = pow 3
Prelude>
Prelude> :type cube
cube :: (Num t2, Eq t2) => t2 -> t2
Prelude>
Prelude> cube 3
27

```

Defining functions in this manner weaves the curried form too tightly into the definition of the function and, as a result, makes the definition of the function cumbersome. Again, the main idea in these examples is that we can support the definition of functions in curried form in any language with first-class closures. For instance, because Python supports first-class closures, we can define the `pow` function in curried form in Python as well:

```

>>> def pfa_pow(e):
...     def pow_e(b):
...         if e == 0:
...             return 1
...         elif e == 1:
...             return b
...         elif b == 0:
...             return 0
...         else: return b * (pfa_pow(e-1)(b))
...     return pow_e
...
>>> pfa_pow(2)(3)
9
>>> square = pfa_pow(2)
>>>
>>> square(3)
9
>>> pfa_pow(3)(3)
27
>>> cube = pfa_pow(3)
>>>
>>> cube(3)
27

```

8.3.7 ML Analogs

Curried form is the same in ML as it is in Haskell:

```

- fun powucf(0,_) = 1
=|   powucf(1,b) = b
=|   powucf(_,0) = 0
=|   powucf(e,b) = b * powucf(e-1, b);
val powucf = fn : int * int -> int
- fun powcf 0 _ = 1
=|   powcf 1 b = b
=|   powcf _ 0 = 0
=|   powcf e b = b * powcf (e-1) b;
val powcf = fn : int -> int -> int
- val square = powcf 2;
val square = fn : int -> int

```

```

- square 3;
9
- (powcf 3) 3;
val it = 27 : int
- powcf 3 3;
val it = 27 : int
- val cube = powcf 3;
val cube = fn : int -> int
- cube 3;
27
- powucf(2,3)
val it = 9 : int
- powucf(2);
stdIn:4.1-4.10 Error: operator and operand don't agree [literal]
operator domain: int * int
operand:          int
in expression:
  powucf 2
- powucf 2
stdIn:1.1-1.9 Error: operator and operand don't agree [literal]
operator domain: int * int
operand:          int
in expression:
  powucf 2
- powcf(2,3)
stdIn:1.1-1.11 Error: operator and operand don't agree
[tycon mismatch]
operator domain: int
operand:          int * int
in expression:
  powcf (2,3)
- powucf 2 3
stdIn:1.1-1.11 Error: operator and operand don't agree [literal]
operator domain: int * int
operand:          int
in expression:
  powucf 2
- (powcf 2) 3;
val it = 9 : int
- powcf 2 3;
val it = 9 : int

```

Not all built-in ML functions are curried as in Haskell. For example, `map` is curried, while `Int.+` is uncurried. Also, there are no built-in `curry` and `uncurry` functions in ML. User-defined and built-in functions in ML that accept only one argument, and which are neither uncurried or curried, can be invoked with or without parentheses around that single argument:

```

- ord("#a"); (* built-in function ord *)
val it = 97 : int
- ord #a;
val it = 97 : int
- fun f x = x; (* user-defined function f *)
val f = fn : 'a -> 'a
- f 1;
val it = 1 : int
- f(1);
val it = 1 : int

```

More generally, when a function is defined in curried form in ML, parentheses can be placed around any individual argument (as in Haskell):

```
- fun f x y z = x+y+z;
val f = fn : int -> int -> int -> int
- f 1 2 3;
val it = 6 : int
- f 1 (2) 3;
val it = 6 : int
- f (1) 2 (3);
val it = 6 : int
- f (1) (2) (3);
val it = 6 : int
```

Conceptual Exercises for Section 8.3

Exercise 8.3.1 Differentiate between *currying* and *curried form*.

Exercise 8.3.2 Give one reason why you might want to *curry* a function.

Exercise 8.3.3 What is the motivation for *currying*?

Exercise 8.3.4 Consider the following function definition in Haskell:

```
f a (b,c) d = c
```

This definition requires that the arguments for parameters *b* and *c* arrive together, as would happen when calling an uncurried function. Is *f* curried? Explain.

Exercise 8.3.5 Would the definition of *curry* in Haskell given in this section work as intended if *curry* was defined in uncurried form? Explain.

Exercise 8.3.6 Can a function *f* be defined in Haskell that returns a function with the same type as itself (i.e., as *f*)? If so, define *f*. If not, explain why not.

Exercise 8.3.7 In some languages, especially type-safe languages, including ML and Haskell, functions also have types, called *type signatures*. Consider the following three type signatures, which assume a binary function $f : (a \times b) \rightarrow c$.

Concept	Function	Type Signature	λ -Calculus
part fun appl 1	papply1	$((a \times b) \rightarrow c) \times a \rightarrow (b \rightarrow c)$	$= \lambda(f, x). \lambda y. f(x, y)$
part fun appl n	papply	$((a \times b) \rightarrow c) \times a \rightarrow (b \rightarrow c)$	$= \lambda(f, x). \lambda(y). f(x, y)$
		$((a \times b) \rightarrow c) \times a \times b \rightarrow ({} \rightarrow d)$	$= \lambda(f, x, y). \lambda(). f(x, y)$
currying	curry	$((a \times b) \rightarrow c) \rightarrow (a \rightarrow (b \rightarrow c))$	$= \lambda(f). \lambda(x). \lambda(y). f(x, y)$

Is `curry = curry papply1`? In other words, is `curry` returned if we pass the function `papply1` to the function `curry`? Said differently, is `curry` *self-generating*? Explain why or why not, using type signatures to prove your case. Write a Haskell or ML program to prove why or why not.

Exercise 8.3.8 What might it mean to state that the *curry* operation acts as a *virtual* compiler (i.e., translator) to λ -calculus? Explain.

Exercise 8.3.9 We can sometimes factor out constant parameters from recursive function definitions so to avoid passing arguments that are not modified across multiple recursive calls (see Section 5.10.3 and *Design Guideline 6: Factor Out Constant Parameters* in Table 5.7).

- (a) Does a recursive function with any constant parameters factored out execute more efficiently than one that is automatically generated using *partial function application* or *currying* to factor out those parameters?
- (b) Which approach makes the function easier to define? Discuss trade-offs.
- (c) Is the order of the parameters in the parameter list of the function definition relevant to each approach? Explain.
- (d) Does the programming language used in each case raise any issues? Consider the language Scheme vis-à-vis the language Haskell.

Programming Exercises for Section 8.3

Return an *anonymous function* in each of the first four exercises.

Exercise 8.3.10 Define the function `papply1` in *curried form* in Haskell for binary functions.

Exercise 8.3.11 Define the function `papply1` in *curried form* in ML for binary functions.

Exercise 8.3.12 Define the function `papply1` in *uncurried form* in Haskell for binary functions.

Exercise 8.3.13 Define the function `papply1` in *uncurried form* in ML for binary functions.

Exercise 8.3.14 Define an ML function in *curried form* and then apply to its argument to create a new function. The function in curried form and the function resulting from applying it must be practical. For example, we could apply a sorting function parameterized on the list to be sorted and the type of items in the list or the comparison operator to be used, a root finding function parameterized by the degree and the number whose *n*th-root we desire, or a number converter parameterized by the base from which to be converted and the base to which to be converted.

Exercise 8.3.15 Complete Programming Exercise 8.3.14 in Haskell.

Exercise 8.3.16 Complete Programming Exercise 8.3.15, but this time define the function in *uncurried form* and then curry it using `curry`.

Exercise 8.3.17 Using higher-order functions and *curried form*, define a Haskell function `dec2bin` that converts a non-negative decimal integer to a list of zeros and ones representing the binary equivalent of that input integer.

Examples:

```
Prelude> :type dec2bin
dec2bin :: Integer -> [Integer]
Prelude>
Prelude> dec2bin 0
[0]
Prelude> dec2bin 1
[1]
Prelude> dec2bin 2
[1,0]
Prelude> dec2bin 3
[1,1]
Prelude> dec2bin 4
[1,0,0]
Prelude> dec2bin 345
[1,0,1,0,1,1,0,0,1]
```

Exercise 8.3.18 Define an ML function `map_ucf` as a user-defined version of the built-in `map` function. The `map_ucf` function must be written in *uncurried form* and, therefore, is slightly different from the built-in ML `map` function. Explain this difference in a program comment.

Exercise 8.3.19 Define the `pow` function from this section in Scheme so that it can be partially applied without the use of the functions `papply1`, `papply`, or `curry`. The `pow` function must have the type *integer* $\rightarrow$ *integer* $\rightarrow$ *integer*. Then use that definition to define the functions `square` and `cube`. Do not define any other named function or any named, nested function other than `pow`.

Exercise 8.3.20 Define the function `curry` in *curried form* in ML for binary functions. Do *not* return an anonymous function.

Exercise 8.3.21 Define the function `uncurry` in *curried form* in ML for binary functions. Do *not* return an anonymous function.

Return an *anonymous function* in each of the following six exercises.

Exercise 8.3.22 Define the function `curry` in *uncurried form* in Haskell for binary functions.

Exercise 8.3.23 Define the function `uncurry` in *uncurried form* in Haskell for binary functions.

Exercise 8.3.24 Define the function `curry` in *uncurried form* in ML for binary functions.

Exercise 8.3.25 Define the function `uncurry` in *uncurried form* in ML for binary functions.

Exercise 8.3.26 Define the function `curry` in Python for binary functions.

Exercise 8.3.27 Define the function `uncurry` in Python for binary functions.

8.4 Putting It All Together: Higher-Order Functions

Curried functions and partial function application open up new possibilities in programming, especially with respect to higher-order functions (HOFs). Recall that a higher-order function, such as `map` in Scheme, is a function that either accepts functions as arguments or returns a function as a value, or both. Such functions capture common, typically recursive, programming patterns as functions. They provide the glue that enables us to combine simple functions to make more complex functions. The use of curried HOFs lifts us to the third layer of functional programming: *More Efficient and Abstract Functional Programming* (Figure 5.10).

Most HOFs are curried, which makes them powerful and flexible. The use of currying, partial function application, and HOFs in conjunction with each other provides support for creating powerful programming abstractions. (We define most functions in this section in curried form.)

Writing a program to solve a problem with HOFs requires:

- creative insight to discern the applicability of a HOF approach to solving a problem
- the ability to decompose the problem and develop atomic functions at an appropriate level of granularity to foster:
 - a solution to the problem at hand by composing atomic functions with HOFs
 - the possibility of recomposing the constituent functions with HOFs to solve alternative problems in a similar manner

8.4.1 Functional Mapping

Programming Exercise 5.4.4 introduces the HOF `map` in Scheme. The `map` function in ML and Haskell accepts only a unary function and returns a function that accepts a list and applies the unary function to each element of the list, and returns a list of the results. The HOF `map` is also built into both ML and Haskell and is curried in both:

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
- map (fn (x) => x*x) [1,2,3,4,5,6];
val it = [1,4,9,16,25,36] : int list
- fun square x = x*x;
val square = fn : int -> int
```

```

- map square [1,2,3,4,5,6];
val it = [1,4,9,16,25,36] : int list
- map (fn x => [x, x]) ["hello", "world"];
val it = [["hello","hello"],["world","world"]] : string list list
- map (fn (x,y) => x+y) [1,2,3,4,5,6];
stdIn:6:1-6:36 Error: operator and operand don't agree [literal]
operator domain: ('Z * 'Z) list
operand:         int list
in expression:
  (map (fn (<pat>,<pat>) => <exp> + <exp>))
    (1 :: 2 :: 3 :: <exp> :: <exp>)
- map (fn x => fn y => x+y) [1,2,3,4,5,6];
val it = [fn,fn,fn,fn,fn,fn] : (int -> int) list

```

In the last two examples, while `map` accepts only a unary function as an argument, that function can be curried. Notice also the difference in the following two uses of `map`, even though both produce the same result:

```

1 - fun squarelist lon = map square lon;
2 val squarelist = fn : int list -> int list
3
4 - val squarelist = map square;
5 val squarelist = fn : int list -> int list
6
7 - squarelist [1,2,3,4,5,6];
8 val it = [1,4,9,16,25,36] : int list

```

The first use of `map` (line 1) is in the context of a new function definition. The function `map` is called (as a complete application) in the body of the new function every time the function is invoked, which is unnecessary. The second use of `map` (line 4) involves partially applying it, which returns a function (with type `int list -> int list`) that is then bound to the identifier `squarelist`. In the second case, `map` is invoked only once, rather than every time `squarelist` is invoked as in the first case. The function `map` has the same semantics in Haskell:

```

Prelude> :type map
map :: (a -> b) -> [a] -> [b]
Prelude>
Prelude> map (\x -> x*x) [1,2,3,4,5,6]
[1,4,9,16,25,36]
Prelude>
Prelude> map (\x -> [x,x]) ["hello", "world"]
[["hello","hello"],["world","world"]]
Prelude>
Prelude> map (\(x,y) -> x+y) [1,2,3,4,5,6]

<interactive>:7:1: error:
  Non type-variable argument in the constraint: Num (b, b)
  (Use FlexibleContexts to permit this)
  When checking the inferred type
    it :: forall b. (Num b, Num (b, b)) => [b]
Prelude>
Prelude> square x = x*x
Prelude>
Prelude> :type square
square :: Num a => a -> a
Prelude>
Prelude> map square [1,2,3,4,5,6]
[1,4,9,16,25,36]

```

```

Prelude>
Prelude> squarelist lon = map square lon
Prelude>
Prelude> :type squarelist
squarelist :: Num b => [b] -> [b]
Prelude>
Prelude> squarelist [1,2,3,4,5,6]
[1,4,9,16,25,36]
Prelude>
Prelude> squarelist1 = map square
Prelude>
Prelude> :type squarelist1
squarelist1 :: Num b => [b] -> [b]
Prelude>
Prelude> squarelist1 [1,2,3,4,5,6]
[1,4,9,16,25,36]

```

8.4.2 Functional Composition

Another HOF is the function composition operator that accepts only two unary functions and returns a function that invokes the two in succession. In mathematics, $g \circ f = g(f(x))$, which means “first apply f and then apply g ” or “ g followed by f ” or “ g of f of x .” The functional composition operator is $\circ$ in ML:

```

- (op o);
val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
- fun add3 x = x+3;
val add3 = fn : int -> int
- fun mult2 x = x*2;
val mult2 = fn : int -> int
- val add3_then_mult2 = mult2 o add3;
val add3_then_mult2 = fn : int -> int
- val mult2_then_add3 = add3 o mult2;
val mult2_then_add3 = fn : int -> int
- add3_then_mult2 4;
val it = 14 : int
- mult2_then_add3 4;
val it = 11 : int

```

The functional composition operator is `.` in Haskell:

```

Prelude> add3 x = x+3
Prelude>
Prelude> :type add3
add3 :: Num a => a -> a
Prelude>
Prelude> mult2 x = x*2
Prelude>
Prelude> :type mult2
mult2 :: Num a => a -> a
Prelude>
Prelude> add3_then_mult2 = mult2 . add3
Prelude>
Prelude> :type add3_then_mult2
add3_then_mult2 :: Num c => c -> c
Prelude>
Prelude> mult2_then_add3 = add3 . mult2
Prelude>
Prelude> :type mult2_then_add3

```

```

mult2_then_add3 :: Num c => c -> c
Prelude>
Prelude> add3_then_mult2 4
14
Prelude>
Prelude> mult2_then_add3 4
11

```

In these Haskell examples, defining functions such as `add3` and `mult2` is unnecessary. To demonstrate why, we must first discuss the concept of a *section* in Haskell.

8.4.3 Sections in Haskell

In Haskell, any binary function or prefix operator (e.g., `div` and `mod`) can be converted into an equivalent infix operator by enclosing the name of the function in grave quotes (e.g., ``div``):

```

Prelude> add x y = x+y
Prelude>
Prelude> :type add
add :: Num a => a -> a -> a
Prelude>
Prelude> 3 `add` 4
7
Prelude> 7 `div` 2
3
Prelude> div 7 2
3
Prelude> 7 `div` 2
3
Prelude> mod 7 2
1
Prelude> 7 `mod` 2
1

```

More importantly for the discussion at hand, the converse is also possible—parenthesizing an infix operator in Haskell converts it to the equivalent *curried prefix* operator:

```

Prelude> :type (+)
(+) :: Num a => a -> a -> a
Prelude> (+) (1,2)
<interactive>:12:1: error:
    Non type-variable argument in the constraint: Num (a, b)
    (Use FlexibleContexts to permit this)
    When checking the inferred type
      it :: forall a b. (Num a, Num b, Num (a, b)) =>
        (a, b) -> (a, b)
Prelude> (+) 1 2
3

```

An operator in Haskell can be partially applied only if it is both curried and invocable in prefix form:

```

Prelude> :type (+) 1
(1 +) :: Num a => a -> a

```

This convention also permits one of the arguments to be included in the parentheses, which both converts the infix binary operator to a prefix binary operator and partially applies it in one stroke:

```
Prelude> :type (1+)
(1 +) :: Num a => a -> a
Prelude> (1+) 3
4
Prelude> :type (+3)
flip (+) 3 :: Num a => a -> a
Prelude> (+3) 1
4
```

In general, if $\oplus$ is an operator, then expressions of the form $(\oplus)$, $(x \oplus)$, and $(\oplus y)$ for arguments x and y are called *sections*, whose meaning as functions can be formalized using lambda expressions as follows:

$$\begin{aligned} (\oplus) &= \lambda x \rightarrow (\lambda y \rightarrow x \oplus y) \\ (x \oplus) &= \lambda y \rightarrow x \oplus y \\ (\oplus y) &= \lambda x \rightarrow x \oplus y \end{aligned} \quad (\text{Hutton 2007, p. 36})$$

Uses of sections include:

1. Constructing simple and succinct functions. Example: `(+3)`
2. Declaring the type of an operator (because an operator itself is not a valid expression in Haskell). Example: `(+) :: Num a => a -> a -> a`
3. Passing a function to a HOF. Example: `map (+1) [1, 2, 3, 4]`

Uses 1 and 3 are discussed in detail in Section 8.4. Returning to the topic of functional composition, we can define the functions using sections in Haskell:

```
Prelude> add3_then_mult2_1 = (*2) . (+3)
Prelude>
Prelude> :type add3_then_mult2_1
add3_then_mult2_1 :: Num c => c -> c
Prelude>
Prelude> add3_then_mult2_2 = (*2) . (3+)
Prelude>
Prelude> :type add3_then_mult2_2
add3_then_mult2_2 :: Num c => c -> c
Prelude>
Prelude> add3_then_mult2_3 = (2*) . (+3)
Prelude>
Prelude> :type add3_then_mult2_3
add3_then_mult2_3 :: Num c => c -> c
Prelude>
Prelude> add3_then_mult2_4 = (2*) . (3+)
Prelude>
Prelude> :type add3_then_mult2_4
add3_then_mult2_4 :: Num c => c -> c
Prelude>
Prelude> mult2_then_add3 = (+3) . (*2)
Prelude>
Prelude> :type mult2_then_add3
mult2_then_add3 :: Num c => c -> c
Prelude>
Prelude> add3_then_mult2_1 4
```

```

14
Prelude> add3_then_mult2_2 4
14
Prelude> mult2_then_add3 4
11

```

The same is not possible in ML because built-in operators (e.g., `+` and `*`) are not curried. In ML, to convert an infix operator (e.g., `+` and `*`) to the equivalent prefix operator, we must enclose the operator in parentheses (as in Haskell) and also include the lexeme `op` after the opening parenthesis:

```

- val add3_then_mult2 = (op o) (mult2, add3);
val add3_then_mult2 = fn : int -> int

```

Recall that while built-in operators in Haskell are curried, built-in operators in ML are not curried. Thus, unlike in Haskell, in ML converting an infix operator to the equivalent prefix operator does not curry the operator, but merely converts it to prefix form:

```

- (op +) (1,2);
val it = 3 : int
- (op +) 1;
stdIn:4.1-4.9 Error: operator and operand don't agree [literal]
operator domain: 'Z * 'Z
operand:         int
in expression:
+ 1

```

Therefore, we cannot define the function `add3_then_mult2` in ML as `val add3_then_mult2 = ( *2) o (+3);`.

The concepts of mapping, functional composition, and sections are inter-related:

```

Prelude> inclist = map ((+) 1)
Prelude>
Prelude> :type inclist
inclist :: Num b => [b] -> [b]
Prelude>
Prelude> inclist [1,2,3,4,5,6]
[2,3,4,5,6,7]

```

Another helpful higher-order function in Haskell that represents a recurring pattern common in programming is `filter`. Intuitively, `filter` selects all the elements of a list that have a particular property. The `filter` function accepts a predicate and a list and returns a list of all elements of the input list that satisfy the predicate:

```

Prelude> :type filter
filter :: (a -> Bool) -> [a] -> [a]
Prelude>
Prelude> filter (>3) [1,2,3,4,5,6]
[4,5,6]
Prelude> filter (/=4.0) [4.0, 3.8, 4.0, 2.2, 2.0, 4.0, 2.7, 3.1, 4.0]
[3.8,2.2,2.0,2.7,3.1]

```

```

Prelude>
Prelude> -- purges space from a string
Prelude> filter (/=' ') "th e uq r q mm io p q g ra "
"theuqrqmmiopqgra"

```

8.4.4 Folding Lists

The built-in ML and Haskell functions `foldl` (“fold left”) and `foldr` (“fold right”), like `map`, capture a common pattern of recursion. As illustrated later in Section 8.4.5, they are helpful for defining a variety of functions.

Folding Lists in Haskell

The functions `foldl` and `foldr` both accept only a prefix binary function (sometimes called the *folding function* or the *combining function*), a base value (i.e., the base of the recursion), and a list, in that order:

```

Prelude> :type foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
Prelude> :type foldr
foldr :: (a -> b -> b) -> b -> [a] -> b

```

The function `foldr` folds a function, given an initial value, across a list from right to left:

$$\text{foldr } \oplus \ v \ [e_0, e_1, \dots, e_n] = e_0 \oplus (e_1 \oplus (\dots (e_{n-1} \oplus (e_n \oplus v)) \dots))$$

where $\oplus$ is a symbol representing an operator. Although `foldr` captures a pattern of recursion, in practice it is helpful to think of its semantics in a non-recursive way. Consider the expression `foldr (+) 0 [1, 2, 3, 4]`. Think of the input list as a series of calls to `cons`, which we know associates from right to left:

```

Prelude> 1:2:3:4:[]
[1,2,3,4]
Prelude> 1:(2:(3:(4:[])))
[1,2,3,4]

```

Now replace the base of the recursion `[]` with `0` and the `cons` operator with `+`:

```

Prelude> 1+(2+(3+(4+0)))
10
Prelude> :{
Prelude| sumlist1 [] = 0
Prelude| sumlist1 (x:xs) = x + sumlist1 xs
Prelude| :}
Prelude>
Prelude> :type sumlist1
sumlist1 :: Num p => [p] -> p
Prelude>
Prelude> sumlist1 [1,2,3,4]
10
Prelude> sumlist = foldr (+) 0
Prelude>
Prelude> :type sumlist

```



```
sumlist :: (Foldable t, Num b) => t b -> b
Prelude>
Prelude> sumlist [1,2,3,4]
10
```

Notice that the function `sumlist`, through the use of `foldr`, implicitly captures the pattern of recursion, including the base case, that is explicitly captured in the definition of `sumlist1`. Figure 8.1 illustrates the use of `foldr` in Haskell.

The function `foldl` folds a function, given an initial value, across a list from left to right:

$$\text{foldl } \oplus \ v \ [e_0, e_1, \dots, e_n] = ((\dots ((v \oplus e_0) \oplus e_1) \dots) \oplus e_{n-1}) \oplus e_n$$

where $\oplus$ is a symbol representing an operator. Notice that the initial value v appears on the left-hand side of the operator with `foldl` and on the right-hand side with `foldr`.

Since `cons` associates from right to left, when thinking of `foldl` in a non-recursive manner we must replace `cons` with an operator that associates from left to right. We use the symbol $\oplus_{l \rightarrow r}$ to indicate a left-associative operator. For instance, consider the expression `foldl (-) 0 [1, 2, 3, 4]`. Think of the input list as a series of calls to $\oplus_{l \rightarrow r}$, which associates from left to right:

$$\begin{aligned} &[] \oplus_{l \rightarrow r} 1 \oplus_{l \rightarrow r} 2 \oplus_{l \rightarrow r} 3 \oplus_{l \rightarrow r} 4 \\ &((([] \oplus_{l \rightarrow r} 1) \oplus_{l \rightarrow r} 2) \oplus_{l \rightarrow r} 3) \oplus_{l \rightarrow r} 4 \end{aligned}$$

Now replace the base of the recursion `[]` with `0` and the $\oplus_{l \rightarrow r}$ operator with `-`:

```
Prelude> (((0-1)-2)-3)-4
-10
```

Figure 8.2 (left) illustrates the use of `foldl` in Haskell.

Folding Lists in ML

The types of `foldr` in ML and Haskell are the same.

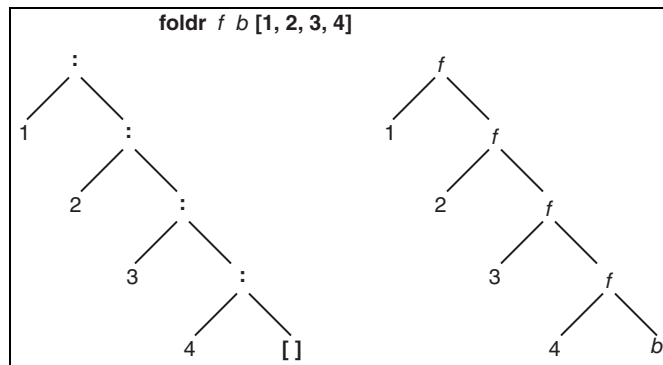


Figure 8.1 `foldr` using the right-associative `:` cons operator.

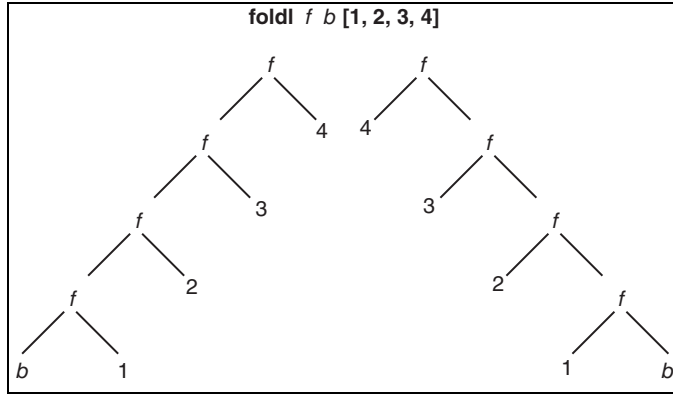


Figure 8.2 `foldl` in Haskell (left) vis-à-vis `foldl` in ML (right).

```
- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

Prelude> :type foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Moreover, `foldr` has the same semantics in ML and Haskell. Figure 8.1 illustrates the use of `foldr` in ML.¹

```
- foldr (op -) 0 [1,2,3,4]; (* 1-(2-(3-(4-0))) *)
val it = ~2 : int

Prelude> foldr (-) 0 [1,2,3,4] -- 1-(2-(3-(4-0)))
-2
```

However, the types of `foldl` in ML and Haskell differ:

```
- foldl;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

Prelude> :type foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Moreover, the function `foldl` has different semantics in ML and Haskell. In ML, the function `foldl` is computed as follows:

$$\text{foldl } \oplus \ v \ [x_0, x_1, \dots, x_n] = x_n \oplus (x_{n-1} \oplus (\dots \oplus (x_1 \oplus (x_0 \oplus v)) \dots))$$

Therefore, unlike in Haskell, `foldl` in ML is the same as `foldr` in ML (or Haskell) with a reversed list:

```
- foldl (op -) 0 [1,2,3,4]; (* 4-(3-(2-(1-0))) *)
val it = 2 : int
- foldr (op -) 0 [4,3,2,1]; (* 4-(3-(2-(1-0))) *)
val it = 2 : int
```

1. The cons operator is `::` in ML.

```
Prelude> foldr (-) 0 [4,3,2,1] -- 4-(3-(2-(1-0)))
2
```

Another way to think of `foldl` in ML is to imagine it as `foldl` in Haskell, but where the folding function accepts its arguments in the reverse of the traditional order:

```
Prelude> foldl (-) 0 [1,2,3,4] -- (((0-1)-2)-3)-4
-10

- (* f(4, f(3, f(2, f(1,0)))) = (((0-1)-2)-3)-4 *)
- foldl (fn (x,y) => (y-x)) 0 [1,2,3,4];
val it = ~10 : int
```

Figure 8.2 illustrates the difference between `foldl` in Haskell and ML.

The pattern of recursion encapsulated in these higher-order functions is recognized as important in other languages, too. For instance, `reduce` in Python, `inject` in Ruby, `Aggregate` in C#, `accumulate` in C++, `reduce` in Clojure, `List::Util::reduce` in Perl, `array_reduce` in PHP, `inject::into::` in Smalltalk, and `Fold` in Mathematica are analogs of the `foldl` family of functions. The `reduce` function in Common Lisp defaults to a left fold, but there is an option for a right fold.

Haskell includes the built-in, higher-order functions `foldl1` and `foldr1` that operate like `foldl` and `foldr`, respectively, but do not require an initial value because they use the first and last elements of the list, respectively, as base values. Thus, `foldl1` and `foldr1` are only defined for non-empty lists. The function `foldl1` folds a function across a list from left to right:

$$\text{foldl1 } \oplus [e_0, e_1, \dots, e_n] = ((\dots ((e_0 \oplus e_1) \oplus e_2) \dots) \oplus e_{n-1}) \oplus e_n$$

The function `foldr1` folds a function across a list from right to left:

$$\text{foldr1 } \oplus [e_0, e_1, \dots, e_n] = e_0 \oplus (e_1 \oplus (\dots (e_{n-2} \oplus (e_{n-1} \oplus e_n)) \dots))$$

```
Prelude> :type foldl1
foldl1 :: (a -> a -> a) -> [a] -> a
Prelude> :type foldr1
foldr1 :: (a -> a -> a) -> [a] -> a
Prelude> foldl1 (+) [1,2,3,4] -- ((1+2)+3)+4
10
Prelude> foldr1 (+) [1,2,3,4] -- 1+(2+(3+4))
10
Prelude> foldl1 (-) [1,2,3,4]
-8
Prelude> foldr1 (-) [1,2,3,4]
-2
Prelude> max 1 2
2
Prelude> max 4 3
4
Prelude> foldl1 max [3,4,2,1,9,7,4,6,8,5]
9
Prelude> foldr1 max [3,4,2,1,9,7,4,6,8,5]
9
```

```
Prelude> foldl1 max []
Program error: pattern match failure: foldl1 max []
```

When to Use `foldl` Vis-à-Vis `foldr`

The functions `foldl` and `foldr` have different semantics and, therefore, which to use depends on the context of the application. Since addition is *associative*,² in this case, `foldr (+) 0 [1,2,3,4]` and `foldl (+) 0 [1,2,3,4]` yield the same result:

```
Prelude> foldl (+) 0 [1,2,3,4] -- ((0+1)+2)+3)+4
10
Prelude> foldr (+) 0 [1,2,3,4] -- 1+(2+(3+(4+0)))
10
```

However, since `foldl` and `foldr` have different semantics, if the folding operator is non-associative (i.e., associates in a particular evaluation order), such as subtraction, `foldr` and `foldl` produce different values. In such a case, we need to use the higher-order function that is appropriate for the operator and application:

```
Prelude> foldl (-) 0 [1,2,3,4] -- (((0-1)-2)-3)-4
-10
Prelude> foldr (-) 0 [1,2,3,4] -- 1-(2-(3-(4-0)))
-2
```

Sometimes `foldl` or `foldr` is used in an application where the values of the elements of the list over which it is applied are not used. For instance, consider the task of determining the length of the list. The values of the elements of the list are irrelevant; all that is of interest is the size of the list. We can define a list length function in Haskell³ with `foldl` succinctly:

```
Prelude> length1 = foldl (\acc _-> acc+1) 0
Prelude>
Prelude> length1 [1,2,3,4]
4
```

Here, the folding operator (i.e., `(\acc _-> acc+1)`) is non-associative. However, since the values of the elements of the list are not considered, the length of the list is always the same regardless of the order in which we traverse it. Thus, even though the folding operator is non-associative, `foldr` is equally as applicable as `foldl` here. However, to use `foldr` we must invert the parameters of the folding operator. With `foldl`, the accumulator value (which starts at 0 in this case) always

2. A binary operator $\oplus$ on a set S is *associative* if $(a \oplus b) \oplus c = a \oplus (b \oplus c) \forall a, b, c \in S$. Intuitively, *associativity* means that the value of an expression containing more than one instance of a single, binary, associative operator is independent of the evaluation order as long as the sequence of the operands is unchanged. In other words, parentheses are unnecessary and rearranging the parentheses in such an expression does not change its value. Addition and multiplication are associative operations, whereas subtraction, division, and exponentiation are non-associative operations.

3. We use the function name `length1` here because Haskell has a built-in function named `length` with the same semantics.

appears on the left-hand side of the folding operator, so it is the first operand; with `foldr`, it appears on the right-hand side, so it is the second operand:

```
Prelude> length1 = foldr (\_ acc -> acc+1) 0
Prelude>
Prelude> length1 [1,2,3,4]
4
```

Thus, when the values of the elements of the input list are not considered, even though the folding operator is non-associative, both `foldl` and `foldr` result in the same value, although the parameters of the folding operator must be inverted in each application. The following is a summary of when `foldl` and `foldr` are applicable based on the associativity of the folding operator:

- If the folding, binary operator is non-associative, each function results in a different value and only one can be used based on the application.
- If the folding, binary operator is associative, either function can be used since each results in the same value.
- If the binary operator is non-associative, but does not depend on the values of the elements in the input list (e.g., list length), either function can be used since each results in the same value, though the operands of the folding operation must be inverted in each invocation.

While `foldl` and `foldr` may result in the same value (i.e., the last two items in the list in our example), one typically results in a more efficient execution and, therefore, is preferred over the other.

- In a language with an eager evaluation strategy (e.g., ML; see Chapter 12), if the folding operator is associative (in other words, when `foldl` and `foldr` yield the same result), it is advisable to use `foldl` rather than `foldr` for reasons of efficiency. Sections 13.7 and 13.7.4 explain this point in more detail.
- In a language with a lazy evaluation strategy (e.g., Haskell; see Chapter 12), if the folding operator is associative, depending on the context of the application, the two functions may not yield the same result, because one may not yield a result at all. If both yield a result, that result will be the same if the folding operator is associative. However, even though they yield the same result, one function may be more efficient than the other. Follow the guidelines given in Section 13.7.4 for which function to use when programming in a lazy language.

8.4.5 Crafting Cleverly Conceived Functions with Curried HOFs

Curried HOFs are powerful programming abstractions that support the definition of functions succinctly. We demonstrate the construction of the following three functions using curried HOFs:

- `implode`: a list-to-string conversion function (online Appendix B)
- `string2int`: a function that converts a string representing a non-negative integer to the corresponding integer
- `powerset`: a function that computes the powerset of a set represented as a list

`implode`

Consider the following `explode` and `implode` functions from online Appendix B:

```
- explode;
val it = fn : string -> char list
- explode "apple";
val it = [#"a",#"p",#"p",#"l",#"e"] : char list
- implode;
val it = fn : char list -> string
- implode [#"a", #"p", #"p", #"l", #"e"];
val it = "apple" : string
- implode (explode "apple");
val it = "apple" : string
```

We can define `implode` using HOFs:

```
- val implode = foldr (op ^) #"";
stdIn:1.29-1.31 Error: character constant not length 1
```

The problem here is that the string concatenation operation `^` only concatenates strings, and not characters:

```
- "hello " ^ "world";
val it = "hello world" : string
- #"h" ^ #"e";
stdIn:6.1-6.12 Error: operator and operand don't agree
[tycon mismatch]
operator domain: string * string
operand:         char * char
in expression:
  #"h" ^ #"e"
```

Thus, we need a helper function that converts a value of type `char` to value of type `string`:

```
- str;
val it = fn : char -> string
```

Now we can use the HOFs `foldr`, `map`, and `o` (i.e., functional composition) to compose the atomic elements:

```
- (* parentheses unnecessary, but present for clarity *)
- val implode = (foldr op ^ "" o map str);
val implode = fn : char list -> string
- val implode = foldr op ^ "" o map str;
val implode = fn : char list -> string
- implode [#"a", #"p", #"p", #"l", #"e"];
val it = "apple" : string
- foldr op ^ "" (map str [#"a", #"p", #"p", #"l", #"e"]);
```

```

val it = "apple" : string
- foldr op ^ "" ["a", "p", "p", "l", "e"];
val it = "apple" : string
- "a" ^ ("p" ^ ("p" ^ ("l" ^ ("e" ^ ""))));
val it = "apple" : string

```

string2int

We now turn to implementing a function that converts a string representing a non-negative integer into the equivalent integer. We know that we can use `explode` to decompose a string into a list of chars. We must recognize that, for example, $123 = (3 + 0) + (2 * 10) + (1 * 100)$. Thus, we start by defining a function that converts a char to an int:

```

- fun char2int c = ord c - ord #"0";
val char2int = fn : char -> int

```

Now we can define another helper function that invokes `char2int` and acts as an accumulator for the integer being computed:

```

- fun helper(c, sum) = char2int c + 10*sum;
val helper = fn : char * int -> int

```

We are now ready to glue the elements together with `foldl`:

```

(* helper (#"3", helper (#"2", helper (#"1", 0))) *)
- foldl helper 0 (explode "123");
val it = 123 : int

```

Since we use `foldl` in ML, we can think of the characters of the reversed string as being processed from right to left. The function `helper` converts the current character to an int and then adds that value to the product of 10 times the running sum of the integer representation of the characters to the right of the current character:

```

- foldl helper 0 (explode "123");
val it = 123 : int
- foldl helper 0 [#"1",#"2",#"3"];
val it = 123 : int
- helper(#"3", helper(#"2", helper(#"1", 0)));
val it = 123 : int
- foldl (fn (c, sum) => char2int c + 10*sum) 0 [#"1",#"2",#"3"];
val it = 123 : int
- foldl (fn (c, sum) =>
    ord c - ord #"0" + 10*sum) 0 [#"1",#"2",#"3"];
val it = 123 : int

```

Thus, we have:

```

- fun string2int s = foldl helper 0 (explode s);
val string2int = fn : string -> int

```

After inlining an anonymous function for helper, the final version of the function is:

```
- fun string2int s =
    foldl (fn (c, sum) => ord c - ord #"0" + 10*sum) 0 (explode s);
val string2int = fn : string -> int
- string2int "0";
val it = 0 : int
- string2int "1";
val it = 1 : int
- string2int "123";
val it = 123 : int
- string2int "321";
val it = 321 : int
- string2int "5643452";
val it = 5643452 : int
```

powerset

The following code from online Appendix B is the definition of a powerset function:

```
$ cat powerset.sml
fun powerset(nil) = [nil]
| powerset(x::xs) =
    let
        fun insertineach(_, nil) = nil
        | insertineach(item, x::xs) =
            (item::x)::insertineach(item, xs);
        val y = powerset(xs)
    in
        insertineach(x, y)@y
    end;

$
$ sml powerset.sml
Standard ML of New Jersey (64-bit) v110.98
[opening powerset.sml]
val powerset = fn : 'a list -> 'a list list
```

Using the HOF map, we can make this definition more succinct:

```
$ cat powerset.sml
fun powerset nil = [nil]
| powerset (x::xs) =
    let
        val temp = powerset xs
    in
        (map (fn excess => x::excess) temp) @ temp
    end;

$
$ sml powerset.sml
Standard ML of New Jersey (64-bit) v110.98
[opening powerset.sml]
val powerset = fn : 'a list -> 'a list list
- powerset [1];
val it = [[1],[]] : int list list
- powerset [1,2];
val it = [[1,2],[1],[2],[]] : int list list
```



```
- powerset [1,2,3];
val it = [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]] : int list list
```

Use of the built-in HOF `map` in this revised definition obviates the need for the nested helper function `insertineach`. Using *sections*, we can make this definition even more succinct in Haskell (Programming Exercise 8.4.23).

Until now we have discussed the use of curried HOFs to create new functions. Here, we briefly discuss the use of such functions to support partial application. Recall that a function can only be partially applied with respect to its first argument or a prefix of its arguments, rather than, for example, its third argument only. To simulate partially applying a function with respect to an argument or arguments other than its first argument or a prefix of its arguments, we need to first transform the order in which the function accepts its arguments and only then partially apply it. The built-in Haskell function `flip` is a step in this direction. The function `flip` reverses (i.e., flips) the order of the parameters to a binary curried function:

```
Prelude> :type flip
flip :: (a -> b -> c) -> b -> a -> c
Prelude>
Prelude> :{
Prelude| powucf(0, _) = 1
Prelude| powucf(1, b) = b
Prelude| powucf(_, 0) = 0
Prelude| powucf(e, b) = b * powucf(e-1, b)
Prelude| :}
Prelude>
Prelude> :type powucf
powucf :: (Num a, Num b, Eq a, Eq b) => (a, b) -> b
Prelude>
Prelude> :{
Prelude| powcf 0 _ = 1
Prelude| powcf 1 b = b
Prelude| powcf _ 0 = 0
Prelude| powcf e b = b * powcf (e-1) b
Prelude| :}
Prelude>
Prelude> :type powcf
powcf :: (Num t1, Num t2, Eq t1, Eq t2) => t1 -> t2 -> t2
Prelude>
Prelude> :type (flip powcf)
(flip powcf) :: (Num t1, Num c, Eq t1, Eq c) => c -> t1 -> c
Prelude>
Prelude> powbase10 = (flip powcf) 10
Prelude>
Prelude> :type powbase10
powbase10 :: (Num t1, Num c, Eq t1, Eq c) => t1 -> c
Prelude>
Prelude> powbase10 2
100
Prelude> powbase10 3
1000
Prelude> (flip powcf) 10 2
100
Prelude>
Prelude> flip (curry powucf) 10 2
100
```

Conceptual Exercises for Section 8.4

Exercise 8.4.1 Explain the motivation for *higher-order functions* such as `map` or `foldl/foldr`.

Exercise 8.4.2 In the definition of `string2int` in ML given in this section, explain why the anonymous function `(fn (c, v) => ord c - ord #"0" + 10*v)` must be defined in uncurried form.

Exercise 8.4.3 Explain the implications of the difference between `foldl` in ML and Haskell for the definition of `string2int` in each of these languages.

Exercise 8.4.4 Typically when composing functions using the functional composition operator, the two operators being composed must both be unary operators, and the second function applied must be capable of receiving a value of the same type as returned by the first function applied. For instance, in Haskell:

```
Prelude> f x = x+1
Prelude>
Prelude> :type f
f :: Num a => a -> a
Prelude>
Prelude> g x = x*2
Prelude>
Prelude> :type g
g :: Num a => a -> a
Prelude>
Prelude> h = g.f
Prelude>
Prelude> :type h
h :: Num c => c -> c
Prelude>
Prelude> h 5
12
```

Explain why the composition on line 10 in the first listing here works in Haskell, but not does not work on line 3 in the second listing in ML. The first function applied—`(+1)` in Haskell and `plus1` in ML—accepts only one argument, while the second function applied—`(:)` in Haskell and `(op ::)` in ML—accepts two arguments:

```
1  Prelude> :type (:)
2  (:) :: a -> [a] -> [a]
3  Prelude>
4  Prelude> :type (+1)
5  (+1) :: Num a => a -> a
6  Prelude>
7  Prelude> :type ((:).(+)1)
8  ((:).(+)1) :: Num a => a -> [a] -> [a]
9  Prelude>
10 Prelude> ((:).(+)1) 2 [1]
11 [3,1]

1  - fun plus1 x = 1 + x;
2  - val plus1 = fn : int -> int
3  - val composition = ((op ::) o plus1);
```

```

4  stdIn:2.20-2.35 Error: operator and operand do not agree
5    [tycon mismatch]
6    operator domain:('Z * 'Z list -> 'Z list) * (int -> 'Z * 'Z list)
7    operand:        ('Z * 'Z list -> 'Z list) * (int -> int)
8    in expression:
9      :: o plus1

```

Exercise 8.4.5 Which of the following two Haskell definitions of `summing` is preferred? Which is more efficient? Explain and justify your explanation.

```

summing l = foldl (+) 0 l
summing = foldl (+) 0

```

Exercise 8.4.6 Explain with function type notation why Programming Exercise 8.4.18 cannot be completed in ML.

Exercise 8.4.7 Explain why there is no need to define `implode` in Haskell.

Programming Exercises for Section 8.4

Exercise 8.4.8 Define a binary function in Haskell that is *commutative*, but not *associative*. Then demonstrate that folding this function across the same list with the same initial value yields different results with `foldl` and `foldr`. A binary operator $\oplus$ on a set S is *commutative* if $(a \oplus b) = (b \oplus a) \forall a, b \in S$. In other words, a binary operator is commutative if changing the order of the operands does not change the result.

Exercise 8.4.9 Define `filter` in Haskell. Name your function `filter1`.

Exercise 8.4.10 Define `foldl` in Haskell. Name your function `foldl2`.

Exercise 8.4.11 Define `foldl1` in Haskell. Name your function `foldl3`.

Exercise 8.4.12 Define `foldr` in Haskell. Name your function `foldr2`.

Exercise 8.4.13 Define `foldr1` in Haskell. Name your function `foldr3`.

Exercise 8.4.14 Define `foldl` in ML. Name your function `foldl2`.

Exercise 8.4.15 Define `foldr` in ML. Name your function `foldr2`.

Exercise 8.4.16 Define a function `map1` in Haskell using a higher-order function in one line of code. The function `map1` behaves like the built-in Haskell function `map`.

Exercise 8.4.17 Use one higher-order function and one anonymous function to define a one-line function `length1` in Haskell that accepts only a list as an argument and returns the length of the list.

Examples:

```

Prelude> :type length1
length1 :: [a] -> Int

```

```

Prelude>
Prelude> length1 []
0
Prelude> length1 [1]
1
Prelude> length1 [1,2]
2
Prelude> length1 [1,2,3]
3
Prelude> length1 [1,2,3,4]
4
Prelude> length1 [1,2,3,4,5,6,7,8,9,10]
10

```

Exercise 8.4.18 Apply a higher-order, curried function to an anonymous function and a base in one line of code to return a function `reverse1` in Haskell that accepts only a list as an argument and returns that list reversed. Try not to use the `++` append operator.

Examples:

```

Prelude> :type reverse1
reverse1 :: [a] -> [a]
Prelude>
Prelude> reverse1 []
[]
Prelude> reverse1 [1]
[1]
Prelude> reverse1 [1,2]
[2,1]
Prelude> reverse1 [1,2,3]
[3,2,1]
Prelude> reverse1 [1,2,3,4]
[4,3,2,1]
Prelude> reverse1 [1,2,3,4,5,6,7,8,9,10]
[10,9,8,7,6,5,4,3,2,1]
Prelude> reverse1 ["cats", "and", "dogs"]
["dogs", "and", "cats"]

```

Exercise 8.4.19 In one line of code, use a higher-order function to define a Haskell function `dnappa` that appends two lists without using the `++` operator.

Examples:

```

Prelude> :type dnappa
dnappa2 :: Foldable t => t a -> [a] -> [a]
Prelude>
Prelude> dnappa [1,2] [3,4]
[1,2,3,4]
Prelude> dnappa ["append"] ["reversed"]
["append", "reversed"]

```

Exercise 8.4.20 Using the higher-order functions `foldl` or `foldr`, define an ML or Haskell function `xorList` that computes the exclusive or (i.e., XOR) of a list of booleans.

Exercise 8.4.21 Use higher-order functions to define a one-line Haskell function `string2int` that accepts only a string representation of a non-negative integer and returns the corresponding integer.

Examples:

```
Prelude> :type string2int
string2int :: [Char] -> Int
Prelude>
Prelude> string2int "0"
0
Prelude> string2int "1"
1
Prelude> string2int "123"
123
Prelude> string2int "321"
321
Prelude> string2int "5643452"
5643452
```

You may assume the Haskell `ord` function, which returns the integer representation of its ASCII character argument. For example:

```
Prelude> import Data.Char (ord)
Prelude Data.Char> :type ord
ord :: Char -> Int
Prelude Data.Char> ord '0'
48
Prelude Data.Char> ord '8'
56
```

The expression `ord(c) - ord('0')` returns the integer analog of the character `c` when `c` is a digit. You may not use the built-in Haskell function `read`.

Note that `string2int` is the Haskell analog of `strtol` in C.

Exercise 8.4.22 Redefine `string2int` in ML or Haskell so that it is capable of converting a string representing any integer, including negative integers, to the corresponding integer.

Haskell examples:

```
Prelude> :type string2int
string2int :: [Char] -> Int
Prelude>
Prelude> string2int "0"
0
Prelude> string2int "1"
1
Prelude> string2int "-1"
-1
Prelude> string2int "123"
123
Prelude> string2int "-123"
-123
Prelude> string2int "321"
321
Prelude> string2int "-321"
-321
```

```

Prelude> string2int "5643452"
5643452
Prelude> string2int "-5643452"
-5643452

```

You may not use the built-in Haskell function `read`.

Exercise 8.4.23 Use a section to define in Haskell, in no more than six lines of code, a more succinct version of the `powerset` function defined in ML in this chapter.

Examples:

```

Prelude> :type powerset
powerset :: [a] -> [[a]]
Prelude>
Prelude> powerset []
[]
Prelude> powerset [1]
[[1], []]
Prelude> powerset [1,2]
[[1,2], [1], [2], []]
Prelude> powerset [1,2,3]
[[1,2,3], [1,2], [1,3], [1], [2,3], [2], [3], []]

```

Exercise 8.4.24 Using higher-order functions and a section, define a recursive function `permutations` in Haskell that accepts only a list representing a set as an argument and returns all permutations of that list as a list of lists.

Examples:

```

Prelude> :type permutations
permutations :: [a] -> [[a]]
Prelude>
Prelude> permutations []
[]
Prelude> permutations [1]
[[1]]
Prelude> permutations [1,2]
[[1,2], [2,1]]
Prelude> permutations [1,2,3]
[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
Prelude> permutations [1,2,3,4]
[[1,2,3,4], [1,2,4,3], [1,3,2,4], [1,3,4,2], [1,4,2,3],
 [1,4,3,2], [2,1,3,4], [2,1,4,3], [2,3,1,4], [2,3,4,1],
 [2,4,1,3], [2,4,3,1], [3,1,2,4], [3,1,4,2], [3,2,1,4],
 [3,2,4,1], [3,4,1,2], [3,4,2,1], [4,1,2,3], [4,1,3,2],
 [4,2,1,3], [4,2,3,1], [4,3,1,2], [4,3,2,1]]
Prelude> permutations ["oranges", "and", "tangerines"]
[["oranges", "and", "tangerines"], ["oranges", "tangerines", "and"],
 ["and", "oranges", "tangerines"], ["and", "tangerines", "oranges"],
 ["tangerines", "oranges", "and"], ["tangerines", "and", "oranges"]]

```

Hint: The solution requires fewer than 10 lines of code.

Exercise 8.4.25 Define `flip2` in Haskell using one line of code. The function `flip2` transposes (i.e., reverses) the arguments to its binary, *curried* function argument.

Examples:

```
Prelude> :type flip2
flip2 :: (a -> b -> c) -> b -> a -> c
Prelude>
Prelude> flip2 elem [1,2,3,4,5] 3
True
Prelude> flip2 powcf 10 2
100
Prelude> flip2 (curry powucf) 10 2
100
```

Exercise 8.4.26 Define `flip2` in Haskell using one line of code. The function `flip2` flips (i.e., reverses) the arguments to its binary, *uncurried* function argument.

Examples:

```
Prelude> :type flip2
flip2 :: ((a,b) -> c) -> (b,a) -> c
Prelude>
Prelude> (flip2 powucf) (10,2)
100
Prelude> (flip2 (uncurry powcf)) (10,2)
100
Prelude> flip2 (uncurry elem) ([1,2,3,4,5], 3)
True
```

Exercise 8.4.27 Write a Haskell program using higher-order functions to solve a complex problem using a few lines of code (e.g., no more than 25). For inspiration, think of some of the functions from this section: the function that reverses a list in linear time in one line of code, the function that converts a string representation of an integer to a integer, and the powerset function.

8.5 Analysis

Higher-order functions capture common, typically recursive, programming patterns as functions. When HOFs are curried, they can be used to automatically define atomic functions—rendering the HOFs more powerful. Curried HOFs help programmers define functions in a modular, succinct, and easily modifiable/reconfigurable fashion. They provide the glue that enables these atomic functions to be combined to construct more complex functions, as the examples in the prior section demonstrate. The use of curried HOFs lifts us to a higher-order style of functional programming—the third tier of functional programming in Figure 5.10. In this style of programming, programs are composed of a series of concise function definitions that are defined through the application of (curried) HOFs (e.g., `map`; functional composition: `o` in ML and `.` in Haskell; and `foldl`/`foldr`). For instance, in our ML definition of `string2int`, we use `foldl`, `explode`, and `char2int`. With this approach, programming becomes essentially the process of creating composable building blocks and combining

them like LEGO® bricks in creative ways to solve a problem. The resulting programs are more concise, modular, and easily reconfigurable than programs where each individual function is defined literally (i.e., hardcoded).

The challenge and creativity in this style of programming require determining the appropriate level of granularity of the atomic functions, figuring out how to automatically define them using (built-in) HOFs, and then combining them using other HOFs into a program so that they work in concert to solve the problem at hand. This style of programming resembles building a library or API more than an application program. The focus is more on identifying, developing, and using the appropriate higher-order abstractions than on solving the target problem. Once the abstractions and essential elements have crystallized, solving the problem at hand is an afterthought. The pay-off, of course, is that the resulting abstractions can be reused in different arrangements in new programs to solve future problems. Lastly, encapsulating patterns of recursion in curried HOFs and applying them in program is a step toward *bottom-up programming*. Instead of writing an all-encompassing program, using a bottom-up style of programming involves building a language with abstract operators and then using that language to write a concise program (Graham 1993, p. 4).

8.6 Thematic Takeaways

- First-class, lexical closures are an important primitive construct for creating programming abstractions (e.g., partial function application and currying).
- Higher-order functions capture common, typically recursive, programming patterns as functions.
- Currying a higher-order function enhances its power because such a function can be used to automatically define new functions.
- Curried, higher-order functions also provide the glue that enables you to combine these atomic functions to construct more complex functions.
- HOFs + Currying $\rightsquigarrow$ Concise Functions + Reconfigurable Programs
- HOFs + Currying (Curried HOFs) $\rightsquigarrow$ Modular Programming

8.7 Chapter Summary

The concepts of partial function application and currying lead to a modular style of functional programming when applied as and with other *higher-order functions* (HOFs). *Partial function application* refers to the concept that if a function—which accepts at least one parameter—is invoked with only an argument for its first parameter (i.e., partially applied), it returns a new function accepting the arguments for the remaining parameters; the new function, when invoked with arguments for those parameters, yields the same result as would have been returned had the original function been invoked with arguments for all of its parameters (i.e., a complete function application). *Currying* refers to converting an n -ary function into one that accepts only one argument and returns a function

that also accepts only one argument and returns a function that accepts only one argument, and so on. Function currying helps us achieve the same end as partial function application (i.e., invoking a function with arguments for only a prefix of its parameters) in a transparent manner—that is, without having to call a function such as `papply1` every time we desire to do so. Thus, while the invocation of a curried function might appear as if it is being partially applied, it is not because every curried function is a unary function.

Higher-order functions support the capture and reuse of a pattern of recursion or, more generally, a pattern of control. (The concept of programming abstractions in this manner is explored further in Section 13.6.) Curried HOFs provide the glue that enables programmers to compose reusable atomic functions together in creative ways. (Lazy evaluation supports gluing whole programs together and is the topic of Section 12.5.) The resulting functions can be used in concert to craft a malleable/reconfigurable program. What results is a general set of (reusable) tools resembling an API rather than a monolithic program. This style of modular programming makes programs easier to debug, maintain, and reuse (Hughes 1989).

8.8 Notes and Further Reading

The concepts of *partial function application* and *currying* are based on Kleene’s S_n^m *theorem* in computability theory. A closely related concept to currying is *partial evaluation*, which is a source-to-source program transformation based on Kleene’s S_n^m *theorem* (Jones 1996). The concept of *currying* is named after the mathematician Haskell Curry who explored the concept. For more information about currying in ML, we refer the reader to Ullman (1997, Chapter 5, Section 5.5, pp. 168–173). For more information on higher-order functions, we refer the reader to Hutton (2007, Chapter 7). For sophisticated examples of the use of higher-order functions in Haskell to create new functions, we refer readers to Chapters 8–9 of Hutton (2007). The built-in Haskell higher-order functions `scanl`, `scanl1`, `scanr`, and `scanr1` are similar to `foldl`, `foldl1`, `foldr`, and `foldr1`. *MapReduce* is a programming model based on the higher-order functions `map` and `fold` (i.e., `reduce`) for processing massive data sets in parallel using multiple computers (Lämmel 2008).

Chapter 9

Data Abstraction

Reimplementing [familiar] algorithms and data structures in a significantly different language often is an aid to understanding of basic data structure and algorithm concepts.

— Jeffrey D. Ullman, *Elements of ML Programming* (1997)

T*ype systems* support *data abstraction* and, in particular, the definition of user-defined data types that have the properties and behavior of primitive types. We discuss a variety of aggregate and inductive data types and the type systems through which they are constructed in this chapter. A *type system* of a programming language includes the mechanism for creating new data types from existing types. A type system should support the creation of new data types easily and flexibly. We also introduce *variant records* and *abstract syntax*, which are of particular use in data structures for representing computer programs. Armed with an understanding of how new types are constructed, we introduce *data abstraction*, which involves factoring the conception and use of a data structure into an *interface*, *implementation*, and *application*. The implementation is hidden from the application such that a variety of representations can be used for the data structure in the implementation without requiring changes to the application since both conform to the interface. A data structure created in this way is called an *abstract data type*. We discuss a variety of representation strategies for data structures, including abstract syntax and closure representations. This chapter prepares us for designing efficacious and efficient data structures for the interpreters we build in Part III of this text (Chapters 10–12).

9.1 Chapter Objectives

- Introduce *aggregate data types* (e.g., arrays, records, unions) and type systems supporting their construction in a variety of programming languages.

- Introduce *inductive data types*—an aggregate data type that refers to itself—and *variant records*—a data type useful as a node in a tree representing a computer program.
- Introduce *abstract syntax* and its role in representing a computer program.
- Describe the design, implementation, and manipulation of efficacious and efficient data structures representing computer programs.
- Explore the conception and use of a data structure as an *interface*, *implementation*, and *application*, which render it an *abstract data type*.
- Recognize and use a *closure representation* of a data structure.
- Describe the design and implementation of data structures for language environments using a variety of representations.

9.2 Aggregate Data Types

An *aggregate data type* is a data type composed of a combination of primitive data types. We both discuss and demonstrate in C the following four primary types of aggregate data types: arrays, records, indiscriminated union, and discriminated union.

9.2.1 Arrays

An *array* is an aggregate data type indexed by integers:

```
/* declaration of integer array scores */
int scores[10];
/* use of integer array scores */
scores[0] = 97;
scores[1] = 98;
```

9.2.2 Records

A *record* (also referred to as a `struct`) is an aggregate data type indexed by strings called field names:

```
/* declaration of struct employee */
struct {
    int id;
    double rate;
} employee;
/* use of struct employee */
employee.id = 555;
employee.rate = 7.25;
```

Records are called *tuples* in the Miranda family of languages, including Miranda, ML, and Haskell. Tuples are indexed by numbers and records are indexed by field names. A record can store at any one time any element of the Cartesian product of the sets of possible values for the data types included in the record. In other words, it can store any element of the Cartesian product of the set of all ints and the set of all doubles.

The parameter and argument list of any uncurried function in ML/Haskell is a tuple; thus, ML/Haskell use tuples to specify the domain of a function. In the context of a tuple or a parameter or argument list of an uncurried function of more than one parameter or argument, the `*` and comma `(,)` in ML and Haskell, respectively, are analogs of the Cartesian-product operator $\times$. The parameter and argument list of any function in C or C++ can similarly be thought of as a `struct`. In C, in the context of a function parameter or argument list with more than one parameter or argument, the comma `(,)` is the analog of the Cartesian-product operator $\times$. Thus, the Cartesian product is the theoretical basis for records. Two instances of records in programming languages are `structs` (in C and C++) and `tuples` (in ML and Haskell).

Before moving onto the next type of aggregate data type, we consider the process of declaring types and variables in C. In C, we declare a variable using the syntax `<type> <identifier>`. The `<type>` can be a named type (e.g., `int` or `double`) or a nameless, literal type as in the previous example. For instance:

```
/* declaration of struct employee */
struct {
    int id;
    double rate;
} employee;
```

Here, we are declaring the variable `employee` to be of the nameless, literal type preceding it, rather than naming the literal type `employee`. The C reserved word `typedef`, with syntax `typedef <type> <type-identifier>`, is used to give a new name to an existing type or to name a literal type. For instance, to give a new name to an existing type, we write `typedef int boolean;`. To give a name to a literal type, for example, we write:

```
/* declaration of a new data type int_and_double */
typedef struct {
    int id;
    double rate;
} int_and_double;
```

The mnemonic `int_and_double` can now be used to declare variables of that nameless `struct` type. The following example declares a variable `int_or_double` using a nameless, literal data type:

```
/* declaration of a struct int_and_double */
struct {
    int id;
    double rate;
} int_and_double;
```

In contrast, the next example assigns a name to a literal data type (lines 2–5) and then, using the type name `int_and_double` given to the literal data type, declares `X` to be an instance of `int_and_double` (line 8):

```

1  /* declaration of a new struct data type int_and_double */
2  typedef struct {
3      int id;
4      double rate;
5  } int_and_double;
6
7  /* declaration of X as type int_and_double */
8  int_and_double X;

```

ML and Haskell each have an expressive type system for creating new types with a clean and elegant syntax. The reserved word `type` in ML and Haskell introduces a new name for an existing type (akin to `typedef` in C or C++):

```

1  (* "type" introduces a new name for an existing type;
2     like a typedef/struct in C/C++ *)
3
4  type id    = int;
5  type name  = string;
6  type yob    = int;
7  type yod    = int;
8
9  (* type composer = (id * name * yob * int); *)
10 type composer = (id * name * yob * yod);
11
12 (*
13 struct {
14     int a;
15     float b;
16 }
17 *)
18
19 val bach      = (1, "Johann Sebastian Bach", 1685, 1750) : composer;
20 val mozart    = (2, "Wolfgang Amadeus Mozart", 1756, 1791) : composer;
21 val beethoven = (3, "Ludwig van Beethoven", 1770, 1827) : composer;
22 val debussy   = (4, "Claude Debussy", 1862, 1918) : composer;
23 val brahms    = (5, "Johannes Brahms", 1833, 1897) : composer;
24 val liszt     = (6, "Franz Liszt", 1811, 1886) : composer;
25
26 type symphony = composer list;
27
28 val composers : symphony = [bach,mozart,beethoven,debussy,brahms,liszt];
29
30 type point = (real * real);
31
32 type rectangle = (point * point * point * point);
33
34 (* can be parameterized like a template in C++ *)
35 type ('domain_type, 'range_type) mapping =
36     ('domain_type * 'range_type) list;
37
38 val floor1 = [(2.1,2), (2.2,2), (2.9,2)] :                (real, int) mapping;
39
40 val composer_mapping =
41     [(4, "Claude Debussy"), (5, "Johannes Brahms")] : (int, string) mapping;
42
43 val lookup = [(beethoven,1), (brahms,2)] :                (composer, id) mapping;
44
45 (* recursive types not permitted
46 type tree = (int * tree list) *)

```

9.2.3 Undiscriminated Unions

An *undiscriminated union* is an aggregate data type that can only store a value of one of multiple types (i.e., a union of multiple types):

```
/* declaration of an undiscriminated union int_or_double */
union {
    /* C compiler only allocates memory for the largest */
    int id;
    double rate;
} int_or_double;
int main() {
    /* use of union int_or_double */
    int_or_double.id = 555;
    int_or_double.rate = 7.25;
}
```

The C compiler allocates memory at least sufficiently large enough to store only the largest of the fields since the union can only store a value of one of the types at any time.¹ The following C program, using the `sizeof (<type>)` function, demonstrates that for a struct, the system allocates memory equal to the sum of its types. This program also demonstrates that the system allocates memory sufficiently large enough to store only the largest of the constituent types of a union:

```
#include <stdio.h>
int main() {
    /* declaration of a new struct data type int_and_double */
    typedef struct {
        int id;
        double rate;
    } int_and_double;
    /* declaration of a new union data type int_or_double */
    typedef union {
        /* C compiler does no checking or enforcement */
        int id;
        double rate;
    } int_or_double;
    /* declaration of X as type int_or_double */
    int_or_double X;
    printf ("An int is %lu bytes.\n", sizeof(int));
    printf ("A double is %lu bytes.\n", sizeof(double));
    printf ("A struct of an int and a double is %lu bytes.\n",
        sizeof(int_and_double));
    printf ("A union of an int or a double is %lu bytes.\n",
        sizeof(int_or_double));
    printf ("A pointer to an int is %lu bytes.\n", sizeof(int*));
    printf ("A pointer to a double is %lu bytes.\n",
        sizeof(double*));
    printf ("A pointer to a union of the two is %lu bytes.\n",
        sizeof(int_or_double*));
    X.rate = 7.777;
    printf("%f\n", X.id);
}
```

1. Memory allocation generally involves *padding* to address an architecture's support for aligned versus unaligned reads; processors generally require either 1-, 2-, or 4-byte alignment for reads.

```

$ gcc sizeof.c
$ ./a.out
An int is 4 bytes.
A double is 8 bytes.
A struct of an int and a double is 16 bytes.
A union of an int or a double is 8 bytes.
A pointer to an int is 8 bytes.
A pointer to a double is 8 bytes.
A pointer to a union of the two is 8 bytes.
0.000000

```

An identifier (e.g., `employee_tag`), if present, between the reserved words `struct` or `union` and the opening curly brace can also be used to name a `struct` or `union` (lines 8 and 17 in the following example). However, when declaring a variable of the `struct` or `union` type named in this way, the identifier (for the type) used in the declaration must be prefaced with `struct` or `union` (lines 14 and 22):

```

1  // example 1:
2  struct {
3      int id;
4      double rate;
5  } lucia;
6
7  // example 2:
8  struct employee_tag {
9      int id;
10     double rate;
11 };
12
13 // can omit the reserved word struct in C++
14 struct employee_tag lucia;
15
16 // example 3:
17 struct employee_tag {
18     int id;
19     double rate;
20 };
21
22 typedef struct employee_tag employee;
23
24 employee lucia;
25
26 // example 4:
27 typedef struct {
28     int id;
29     double rate;
30 } employee;
31
32 employee lucia;

```

Each of the previous four declarations in C (or C++) of the variable `lucia` is valid. Use of the literal, unnamed type in the first example (lines 1–5) is recommended only if the type will be used just once to declare a variable. Which of the other three styles to use is a matter of preference.

While most readers are probably more familiar with records (or `structs`) than unions, unions are helpful types for nodes of a parse or abstract-syntax tree

because each node must store values of different types (e.g., ints, floats, chars), but the tree must be declared to store a single type of node.

9.2.4 Discriminated Unions

A *discriminated union* is a record containing a union as one field and a flag as the other field. The flag indicates the type of the value currently stored in the union:

```
#include <stdio.h>
int main() {
    /* declaration of a discriminated union
       int_or_double_wrapper */
    struct {
        /* declaration of flag as an enumerated type */
        enum {i, f} flag;
        /* declaration of a union int_or_double */
        union {
            /* C compiler does no checking or enforcement */
            int id;
            double rate;
        } int_or_double;
    } int_or_double_wrapper;
    int_or_double_wrapper.flag = i;
    int_or_double_wrapper.int_or_double.id = 555;
    int_or_double_wrapper.flag = f;
    int_or_double_wrapper.int_or_double.rate = 7.25;
    if (int_or_double_wrapper.flag == i)
        printf ("%d\n", int_or_double_wrapper.int_or_double.id);
    else
        printf ("%f\n",
                int_or_double_wrapper.int_or_double.rate);
}
$ gcc discr_union.c
$ ./a.out
7.250000
```

While we have presented examples of four types of aggregate data types in C, these types are not specific to any particular programming language and can be implemented in a variety of languages.

Programming Exercises for Section 9.2

Exercise 9.2.1 Consider the following two `structs` and variable declarations in C:

```
struct recordA {
    int x;
    double y;
};
struct recordB {
    double y;
    int x;
};
struct recordA A;
struct recordB B;
```


Do variables `A` and `B` require the same amount of memory? If not, why not? Write a program using the `sizeof (<type>)` function to determine the answer to this question, which should be given in a comment in the program.

Exercise 9.2.2 Can a `union` in `C` be used to convert `ints` to `doubles`, and vice versa? Write a `C` program to answer this question. Show your program and explain how it illustrates that a `union` in `C` can or cannot be used for these in a comment in the program.

Exercise 9.2.3 Can an *undiscriminated* `union` in `C` be statically type checked? Write a `C` program to answer this question. Show and use your program to support your answer to this question, which should be given in a comment in the program.

Exercise 9.2.4 Rewrite the `ML` program in Section 9.2.2 in `Haskell`. The two programs are nearly identical, with the differences resulting from the syntax in `Haskell` being slightly more terse than that in `ML`. See Table 9.7 later in this chapter for a comparison of the main concepts and features, including syntactic differences, of `ML` and `Haskell`.

9.3 Inductive Data Types

An *inductive data type* is an aggregate data type that refers to itself. In other words, the type being defined is one of the constituent types of the type being defined. A node in a singly linked list is a classical example of an inductive data type. The node contains some value and a pointer to the next node, which is also of the same node type:

```
struct node_tag {
    int id;
    struct node_tag* next;
};
struct node_tag head;
```

Technically, this example type is not an inductive data type because the type being defined (`struct node_tag`) is not a member of itself. Rather, this type contains a pointer to a value of its type (`struct node_tag*`). This discrepancy highlights a key difference between a compiled language and an interpreted language. `C` is a compiled language, so, when the compiler encounters the preceding code, it must generate low-level code that allocates enough memory to store a value of type `struct node_tag`. To determine the number of bytes to allocate, the compiler must sum the constituent parts. An `int` is four bytes and a pointer (to any type) is also four bytes. Therefore, the compiler generates code to allocate eight bytes. Had the compiler encountered the following definition, which is a pure inductive data type because a `struct node_tag` contains a field of type `struct node_tag`, it would have no way of determining statically (i.e., before run-time) how much memory to allocate for the variable `head`:

```
struct node_tag {
    int id;
```

```

    struct node_tag next;
};
struct node_tag head;

```

While the recursion must end somewhere (because the memory of a computer is finite), there is no way for the compiler to know in advance how much memory is required. C, and other compiled languages, address this problem by using pointers, which are always a consistent size irrespective of the size of the data to which they point. In contrast, interpreted languages do not encounter this problem because an interpreter only operates at run-time—a point at which the size of data type is known or can be grown or shrunk. Moreover, in some languages, including Scheme, all denoted values are references to literal values, and references are implicitly dereferenced when used. A *denoted value* is the value to which a variable refers. For instance, if $x = 1$, the denotation of x is the value 1. In Scheme, since all denoted values are references to literal values, the denotation of x is a reference to the value 1. The following C program demonstrates that in C all denoted values are not references, and includes an example of explicit pointer dereferencing (line 15):

```

1  #include <stdio.h>
2
3  int main() {
4
5      /* the denotation of x is the value 1 */
6      int x = 1;
7
8      /* the denotation of ptr_x is the address of x */
9      int* ptr_x = &x;
10
11     printf ("The denotation of x is %d.\n", x);
12     printf ("The denotation of ptr_x is %x.\n", ptr_x);
13
14     /* explicit dereferencing ptr_x */
15     printf ("The denotation of ptr_x points to %d.\n", *ptr_x);
16 }
17
18 $ gcc deref.c
19 $ ./a.out
20 The denotation of x is 1.
21 The denotation of ptr_x is bffff628.
22 The denotation of ptr_x points to 1.

```

We cannot write an equivalent Scheme program. Since all denoted values are references in Scheme, it is not possible to distinguish between a denoted value that is a literal and a denoted value that is a reference:

```

;; the denotation of x is a reference to the value 1
(let ((x 1))
  ;; x is implicitly dereferenced
  (+ x 1))

```

Similarly, in Java, all denoted values except primitive types are references. In other words, in Java, unlike in C++, it is not possible to refer to an object literally. All objects must be accessed through a reference. However, since Java, like Scheme, also has implicit dereferencing, the fact that all objects are accessed through a reference is transparent to the programmer. Therefore, languages such as Java and

Scheme enjoy the efficiency of manipulating memory through references (which is fast) while shielding the programmer from the low-level details of (manipulating) memory, which are requisite in C and C++. For instance, consider the following two equivalent programs—the first in C++ and the second in Java:

```

1  #include <iostream>
2
3  using namespace std;
4
5  class Ball {
6
7      public:
8          void roll1();
9  };
10
11 void Ball::roll1() {
12     cout << "Ball roll." << endl;
13 }
14
15 int main() {
16
17     // the denotation of b is a value of type Ball
18     Ball b = Ball();
19
20     // the denotation of ref_b is a reference to
21     // the same value of type Ball
22     Ball* ref_b = &b;
23
24     // sending the message roll to the object b of type Ball
25     b.roll1();
26
27     // sending the message roll to the object b of type Ball
28     // through the reference ref_b
29     ref_b->roll1();
30
31     // explicit pointer dereferencing
32     // (*ref_b) results in a value of type Ball
33     // sending the message roll to that value
34     (*ref_b).roll1();
35 }

```

```

$ g++ BallDemo.cpp
$ ./a.out
Ball roll.
Ball roll.
Ball roll.

```

This C++ program demonstrates accessing an object through a non-pointer value (i.e., directly through itself; line 25), through a pointer with implicit dereferencing (line 29), and through a pointer with explicit dereferencing (line 34). Now consider the same program written in Java:

```

1  class Ball {
2      public void roll() {
3          System.out.println ("Ball roll.");
4      }
5  }
6
7  public class BallDemo {
8      public static void main(String[] args) {

```

```

9
10      // the denotation of b is a reference to a value of type Ball
11      Ball b = new Ball();
12
13      // the denotation of ref_b is a reference to
14      // the same value of type Ball
15      Ball ref_b = b;
16
17      // both references are implicitly dereferenced
18      b.roll();
19
20      ref_b.roll();
21  }
22  }

$ javac BallDemo.java
$ java BallDemo
Ball roll.
Ball roll.

```

This Java program demonstrates object access through a pointer with implicit dereferencing (lines 18 and 20). In short, it is natural to create pure inductive data types in languages where all denoted values are references (e.g., Scheme and Java for all non-primitives).

9.4 Variant Records

A *variant record* is an aggregate data type that is a union of records (i.e., a union of structs); it can hold any one of a variety of records. Each constituent record type is called a *variant* of the union type. Variant records can also be inductive. Consider the idea that context-free grammars can be used to define data structures in addition to languages, which we explored in Chapter 5. A variant record is an effective building block for building a data structure defined with a context-free grammar because the variant record mirrors the EBNF definition of the data structure. We can use a linked list of integers in C to illustrate a variant record. Consider the following EBNF definition of a list:

$$\begin{aligned}
 \langle L \rangle &::= \langle A \rangle \\
 \langle L \rangle &::= \langle A \rangle \langle L \rangle \\
 \langle A \rangle &::= -2^{31} \mid \dots \mid 2^{31} - 1 \text{ (i.e., ints)}
 \end{aligned}$$

The following example shows a variant record for this linked list in C:

```

1  /* a variant record: a union of structs */
2  #include <stdio.h>
3
4  int main() {
5
6      typedef union llist_tag {
7
8          /* <L> ::= <A> variant */
9          struct aatom_tag {
10             int number;
11         } aatom;
12

```

```

13      /* <L> ::= <A> <L> variant */
14      struct {
15          struct aatom_tag aatom;
16          union llist_tag* next_llist;
17      } aatom_llist;
18  } llist;
19
20  printf ("llist is %ld bytes.\n", sizeof (llist));
21
22  /* list1 is the list "1 2 3" */
23  llist list1, list2, list3;
24
25  list3.aatom.number = 1;
26
27  list2.aatom_llist.aatom.number = 2;
28  list2.aatom_llist.next_llist = &list3;
29
30  list1.aatom_llist.aatom.number = 3;
31  list1.aatom_llist.next_llist = &list2;
32  }

```

```

$ gcc List.c
$ ./a.out
llist is 16 bytes.

```

While observably superfluous, wrapping `int number;` in a `struct` (lines 8–11) is required to make `List` a variant record because a variant record is a union of `structs`. More importantly, this implementation of a variant record of the list completely mirrors the EBNF definition of the list. Each variant of the union corresponds to an alternative of the non-terminal $\langle L \rangle$. Specifically, the variant `aatom` (lines 8–11) corresponds to the production rule $\langle L \rangle ::= \langle A \rangle$, while the variant `aatom_llist` (lines 13–18) corresponds to the production rule $\langle L \rangle ::= \langle A \rangle \langle L \rangle$. *Key theme*: There is a one-to-one mapping between production rules and constructors.

9.4.1 Variant Records in Haskell

The reserved word `data` in Haskell introduces a new type. Consider the following Haskell program illustrating both the power of the ML type system and the ease with which it can be used to rapidly construct complex data types, akin to building with LEGO® bricks. Run this program incrementally through the Haskell interpreter multiple times while exploring the types declared, the values of those types constructed, and the functions that manipulate the values of those data types (and the values they return):

```

1  -- "data" introduces a new type
2  -- a variant record or a union of structs
3  -- comparable to define-datatype construct
4  -- from [EOPL2] or ML's "datatype"
5
6  -- data Bool = True | False is in Prelude.hs
7
8  data Color = Red | Green | Blue | Orange | Yellow
9
10 type Mapping domain_type range_type = [(domain_type , range_type)]
11

```

```

12 decorate :: Mapping String Color
13 decorate = [("la cuisine", Yellow), ("le salon", Blue)]
14
15 {--
16 union {
17     int a;
18     float b;
19 } --}
20
21 data Daysoftheweek = Sun | Mon | Tue | Wed | Thu | Fri | Sat
22                     deriving (Show, Eq)
23
24 onholiday :: Daysoftheweek -> Bool
25 onholiday Sun = True
26 onholiday day = (day == Sun) || (day == Sat)
27
28 val = onholiday Mon
29 val2 = onholiday Sat
30
31 -- can be parameterized like a template in C++
32 data Student a = New | Id a
33
34 -- can be recursive
35 data Natural = Zero | Succ Natural
36
37 four = (Succ (Succ (Succ (Succ Zero))))
38
39 data IntTree = Leaf Int | NodeIT IntTree Int IntTree
40
41 data Bintreeofints = EmptyI | NodeI Bintreeofints Int Bintreeofints
42                     deriving (Show, Eq)
43
44 ourbintreeofints :: Bintreeofints
45 ourbintreeofints = (NodeI
46     (NodeI
47         (NodeI EmptyI 1 EmptyI)
48         7
49         (NodeI EmptyI 2 EmptyI))
50     6
51     (NodeI
52         (NodeI EmptyI 3 EmptyI)
53         8
54         (NodeI
55             (NodeI EmptyI 5 EmptyI)
56             4
57             (NodeI EmptyI 10 EmptyI))))
58
59 -- if inorder returns a sorted list,
60 -- then its argument is a binary search tree
61 -- inorderI :: Bintreeofints -> [Int]
62 inorderI EmptyI = []
63 inorderI (NodeI left i right) =
64     (inorderI left) ++ [i] ++ (inorderI right)
65
66 preorderI EmptyI = []
67 preorderI (NodeI left i right) =
68     [i] ++ (preorderI left) ++ (preorderI right)
69
70 postorderI EmptyI = []
71 postorderI (NodeI left i right) =
72     (postorderI left) ++ (postorderI right) ++ [i]
73
74 val3 = inorderI ourbintreeofints

```

```

75 val4 = preorderI ourbintreeofints
76 val5 = postorderI ourbintreeofints
77
78 -- like typedef in C
79 -- type and constructor names must begin with a capital letter
80 type Id = Int
81 type Name = String
82 type DoB = Int
83
84 type Professor = (Id, Name, DoB)
85 --type Professor = (Id, Name, Int)
86
87 {--
88 struct {
89     int a;
90     float b;
91 } --}
92
93 turing = (1, "Alan Turing", 19120623) :: Professor
94 church = (2, "Alonzo Church", 19030614) :: Professor
95 mccarthy = (3, "John McCarthy", 19270904) :: Professor
96 hindley = (4, "J. Roger Hindley", 1939) :: Professor
97 milner = (5, "Robin Milner", 19340113) :: Professor
98 keller = (6, "Mary Kenneth Keller", 191312017) :: Professor
99 curry1 = (7, "Haskell Curry", 19000912) :: Professor
100
101 type Department = [Professor]
102
103 computing = [turing,church,mccarthy,hindley,milner,keller,curry1] ::
104     Department
105
106 julia = (8, "Julia Robinson", 19191208) :: Professor
107 yuri = (9, "Yuri Matiyasevich", 19470302) :: Professor
108
109 mathematics = [julia,yuri] :: Department
110
111 mendel = (10, "Gregor Mendel", 18220720) :: Professor
112 boveri = (11, "Marcella Boveri", 18631007) :: Professor
113
114 biology = [mendel,boveri] :: Department
115
116 type University = [Department]
117
118 univofgenius = [computing,mathematics,biology] :: University
119
120 -- can be parameterized and recursive
121 data List a = Nil | Kons a (List a)
122     deriving (Show,Eq)
123
124 listofcsprofs =
125     (Kons turing
126     (Kons church
127     (Kons mccarthy
128     (Kons hindley
129     (Kons milner
130     (Kons keller
131     (Kons curry1 Nil)))))) :: List Professor
132
133 -- parameterized and recursive data type
134 data Bintree a = Empty | Node (Bintree a) a (Bintree a)
135     deriving (Show,Eq)
136
137 ourbintreeofints2 :: Bintree Integer

```

```

138 ourbintreeofints2 = (Node
139   (Node
140     (Node Empty 1 Empty)
141     7
142     (Node Empty 2 Empty))
143   6
144   (Node
145     (Node Empty 3 Empty)
146     8
147     (Node
148       (Node Empty 5 Empty)
149       4
150       (Node Empty 10 Empty))))
151
152 ourbintreeofstrs = (Node
153   (Node
154     (Node Empty "one" Empty)
155     "seven"
156     (Node Empty "two" Empty))
157   "six"
158   (Node
159     (Node Empty "three" Empty)
160     "eight"
161     (Node
162       (Node Empty "five" Empty)
163       "four"
164       (Node Empty "ten" Empty)))) :: Bintree String
165
166 ourbintreeofstrs2 = (Node
167   (Node
168     (Node Empty "The" Empty)
169     "name"
170     (Node Empty "of" Empty))
171   "the"
172   (Node
173     (Node Empty "data type" Empty)
174     "is"
175     (Node Empty "Bintree." Empty)))
176
177 ourbintreeofProfessors = (Node
178   (Node
179     (Node Empty turing Empty)
180     church
181     (Node Empty mccarthy Empty))
182   hindley
183   (Node
184     Empty
185     milner
186     (Node Empty curry1 Empty)))
187
188 ourbintreeofDepartments = (Node
189   (Node Empty computing Empty)
190   mathematics
191   (Node Empty biology Empty))
192
193 {-# Declaring the type of a function is not required, but
194    can be used to either resolve any type errors that would
195    otherwise occur or constrain the type of the function
196    beyond the type that the Hindley-Milner type inference algorithm
197    would otherwise infer. -#}
198 inorder :: Bintree a -> [a]
199 inorder Empty = []
200 inorder (Node left i right) =

```


Data Type	C/C++	ML	Haskell	Python	Java
records	struct	type	type	class	class
unions/variant records	union	datatype	data	class	class

Table 9.1 Support for C/C++ Style structs and unions in ML, Haskell, Python, and Java

```

201     (inorder left) ++ [i] ++ (inorder right)
202
203 preorder Empty = []
204 preorder (Node left i right) =
205     [i] ++ (preorder left) ++ (preorder right)
206
207 postorder Empty = []
208 postorder (Node left i right) =
209     (postorder left) ++ (postorder right) ++ [i]
210
211 val6 = inorder ourbintreeofstrs
212 val7 = preorder ourbintreeofstrs
213 val8 = postorder ourbintreeofstrs
214
215 val9 = inorder ourbintreeofstrs2
216 val10 = preorder ourbintreeofstrs2
217 val11 = postorder ourbintreeofstrs2
218
219 val12 = inorder ourbintreeofProfessors
220 val13 = preorder ourbintreeofProfessors
221 val14 = postorder ourbintreeofProfessors

```

Table 9.1 summarizes the support for C/C++-style structs and unions in ML, Haskell, Python, and Java.

9.4.2 Variant Records in Scheme: (define-datatype ...) and (cases ...)

Unlike ML and Haskell, Scheme does not have built-in support for defining and manipulating variant records, so we need a tool for these tasks in Scheme. The (define-datatype ...) and (cases ...) extensions to Racket Scheme created by Friedman, Wand, and Haynes (2001) provide support for constructing and deconstructing respectively, variant records in Scheme. The (define-datatype ...) form *defines* variant records.

Syntax:

```

(define-datatype <type-name> <type-predicate-name>
  { ( <variant-name> { (<fieldname> <predicate> ) }*) }+)

```

A new function called a *constructor* is automatically created for each variant to construct data values belonging to that variant. The following code is a data type definition of a list of integers:

```
#lang eopl
(define-datatype llist llist?
  (aatom
    (aatom_tag number?))
  (aatom_llist
    (aatom_tag number?)
    (next llist?)))
```

To interpret this definition, set the language to *Essentials of Programming Languages* by including `#lang eopl` as the first line of the program in *DrRacket* IDE. This definition automatically creates a linked list variant record and an *implementation* of the following *interface*:

- a unary function `aatom`, which creates an atom node
- a binary function `aatom_llist`, which creates an atom list node
- a binary predicate `llist?`

We build `llists` using the constructors:

```
> (aatom 3)
#(struct:aatom 3)
> (define ouraatom (aatom 3))
> (llist? ouraatom)
#t
> (aatom_llist 2 (aatom 3))
#(struct:aatom_llist 2 #(struct:aatom 3))
> (define ouraatom_llist (aatom_llist 2 (aatom 3)))
> (llist? ouraatom_llist)
#t
> (llist? (aatom_llist 1 ouraatom_llist))
#t
> (define ourllist (aatom_llist 1 ouraatom_llist))
> (llist? ourllist)
#t
```

The `(cases ...)` form, in the EOPL extension to Racket Scheme, provides support for *decomposing* and *manipulating* the constituent parts of a variant record created with the constructors automatically generated with the `(define-datatype ...)` form.

Syntax:

```
(cases <type-name> <expression>
  {(<variant-name> ({<field-name>}*) <consequent>}) *
  (else <default>))
```

The following function accepts a value of type `llist` as an argument and manipulates its fields with the `cases` form to sum its nodes:

```
(define llist_sum
  (lambda (ll)
    (cases llist ll
      (aatom (aatom_tag) aatom_tag)
      (aatom_llist (aatom_tag next)
        (+ aatom_tag (llist_sum next))))))
> (llist_sum ouraatom)
```

Language	Composition	Decomposition
C	union of structs with flag	switch (...) { case ... }
C++	union of structs with flag	switch (...) { case ... }
Java	class with flag	switch (...) { case ... }
Python	class with flag	if/elif/else
ML	datatype	pattern-directed invocation
Haskell	data	pattern-directed invocation
Racket Scheme	define-datatype form	cases form

Table 9.2 Support for Composition (Definition) and Decomposition (Manipulation) of Variant Records in a Variety of Programming Languages

```

3
> (l1ist_sum ouraatom_llist)
5
> (l1ist_sum ourllist)
6

```

Notice that the `(cases ...)` form binds the values of the fields of the value of the data type to symbols (for subsequent manipulation). The `define-datatype` and `cases` forms are the analogs of the composition and decomposition operators, respectively. Data types defined with `(define-datatype ...)` can also be *mutually recursive* (recall the grammar for *S-expressions*). In SLLGEN, the `sllgen:make-define-datatypes` procedure is used to automatically generate the `define-datatype` declarations from the grammar (or we can manually define them). Table 9.2 summarizes the support for defining and manipulating variant records in the programming languages we have discussed here.

Programming Exercises for Section 9.4

Exercise 9.4.1 Explain why the following C++ code does not compile successfully. Explain why the Racket Scheme `(define-datatype ...)` construct does not suffer from this problem. (Java also does not suffer from this problem.) Modify the following code so that it will compile successfully.

```

union bintree {
    struct {
        int number;
    } leaf;
    struct {
        int key;
        union bintree left;
        union bintree right;
    } interior_node;
} B;

```

Show your code and explain your observations in a comment in the program.

Exercise 9.4.2 Rewrite the Haskell program in Section 9.4.1 in ML. The two programs are nearly identical, with the differences resulting from the syntax in ML

being slightly more verbose than that in Haskell. Table 9.7 (later in the chapter) compares the main concepts and features, including the syntactic differences, of ML and Haskell.

Exercise 9.4.3 Pascal implements a form of *discriminated union* using *variant records*. Write a Pascal program to determine whether the Free Pascal compiler² tests the discriminant of a variant record when a variant field is accessed. Report your observations.

Exercise 9.4.4 Consider the following definition of a list in EBNF:

```

<L> ::= <A>
<L> ::= <A><L>
<A> ::=  $-2^{31} \mid \dots \mid 2^{31} - 1$  (i.e., ints)
<A> ::= any floating point number of type float
<A> ::= a | b | c | ... | x | y | z (i.e., lowercase alphabetic chars)

```

Define a *variant record* `list` in C++ for this list data structure. The data type must be inductive and must completely conform to (i.e., naturally reflect) the grammar shown here. Do not use more than 25 lines of code in your definition of the data type, and do not use a `class` or any other object-oriented features of C++.

Exercise 9.4.5 (Ullman 1997, Exercise 6.2.8, pp. 209–210) Define a Haskell data type for boolean expressions. Boolean expressions are made up of boolean values, boolean variables, and operators. There are two boolean values: `True` or `False`. A boolean variable (e.g., “`p`”) can be bound to either of the two boolean values. Boolean expressions are constructed from boolean variables and values using the operators `AND`, `OR`, and `NOT`. An example of a boolean expression is `(AND (OR p q) (NOT q))`, where `p` and `q` are boolean variables. Another example is `(AND p True)`.

- (a) Define a Haskell data type `Boolexp` whose values represent legal boolean expressions. You may assume that boolean variables (but not the expressions themselves) are represented as strings.
- (b) Define a function `eval exp env :: Boolexp -> [[Char]] -> Bool` in Haskell that accepts a boolean expression `exp` and a list of true boolean variables `env`, and determines the truth value of `exp` based on the assumption that the boolean variables in `env` are true and all other boolean variables are false. You may use the Haskell `elem` list member function in your definition of `eval`.

Bear in mind that `exp` is not a string, but rather a value constructed from the `Boolexp` data type.

2. <https://www.freepascal.org>

Examples:

```

Prelude> eval (Variable "p") []
False
Prelude> eval (Literal True) ["p", "q", "r"]
True
Prelude> eval (NOT(Variable "p")) []
True
Prelude> eval (NOT(Variable "p")) ["q"]
True
Prelude> eval (NOT(Variable "q")) ["q"]
False
Prelude> eval (AND (OR (Variable "p") (Variable "q"))
                  (NOT (Variable "q"))) ["p"]
True

```

Solve this exercise with at most one data type definition and a five-line eval function.

Exercise 9.4.6 Consider the following definition of a binary tree in BNF:

```

<bintree> ::= <number>
<bintree> ::= ( <bintree> <symbol> <bintree> )

```

- (a) Define a *variant record* `binarytree` in Racket Scheme using `(define-datatype ...)` for this binary tree data structure. The data type must be inductive and must completely conform to (i.e., naturally reflect) the grammar shown here.
- (b) Define a function `sum_leaves` in Racket Scheme using `(cases ...)` to sum the leaves of a binary tree created using the data type defined in (a).

9.5 Abstract Syntax

Consider the string `((lambda (x) (f x)) (g y))` representing an expression in λ -calculus. An implementation of a programming language, such as an interpreter or compiler, reads strings like this, typically from standard input, and processes them. This program string is an *external representation* (i.e., it is external to the system processing it) and uses *concrete syntax*. Programs in concrete syntax are not readily processable. Scheme, however, is a homoiconic language, meaning that program codes and data are both represented using the same representation—in the case of Scheme as a list. In consequence, the availability of a Scheme program as an S-expression is convenient for any system processing it. The `(read)` facility in Scheme reads from standard input and returns the data read as an S-expression, sometimes called a list-and-symbol representation:

```

> (define program (read))
((lambda (x) (f x)) (g y))
> program
((lambda (x) (f x)) (g y))

```

```
> (car program)
(lambda (x) (f x))
> (cdr program)
((g y))
```

While an S-expression representing a program can be more easily processed using calls to `car` and `cdr` than can a string representing a program, we still consider the former as concrete syntax.

Accessing the individual lexemes of this program to evaluate this expression requires cryptic and lengthy chains of calls to `car` and `cdr`. For instance, consider accessing the operand `x` of the call to `f`:

```
> (car (cdr (car (cdr (cdr (car program))))))
x
```

Notably, the preceding program is more manipulable and, thus, processable when represented using the following definition of an expression data type:

```
(define-datatype expression expression?
  (variable-expression
    (identifier symbol?))
  (lambda-expression
    (identifier symbol?)
    (body expression?))
  (application-expression
    (operator expression?)
    (operand expression?)))
```

An *abstract-syntax tree* (AST) is similar to a parse tree, except that it uses *abstract syntax* or an *internal representation* (i.e., it is internal to the system processing it) rather than *concrete syntax*. Specifically, while the structure of a parse tree depicts how a sentence (in concrete syntax) conforms to a grammar, the structure of an abstract-syntax tree illustrates how the sentence is represented internally, typically with an inductive, variant record data type. For instance, Figure 9.1 illustrates an AST for the λ -calculus expression `((lambda (x) (f x)) (g y))`. Abstract syntax is a representation of a program as a data structure—in this case, an inductive variant record. Consider the following grammar for λ -calculus, which is annotated with variants of this `expression` inductive variant record data type above the right-hand side of each production rule:³

		variable-expression (identifier)
<expression>	::=	<identifier>
		lambda-expression (identifier body)
<expression>	::=	(lambda (<identifier>) <expression>)
		application-expression (operator operand)
<expression>	::=	(<expression> <expression>)

3. This is the annotative style used in Friedman, Wand, and Haynes (2001).

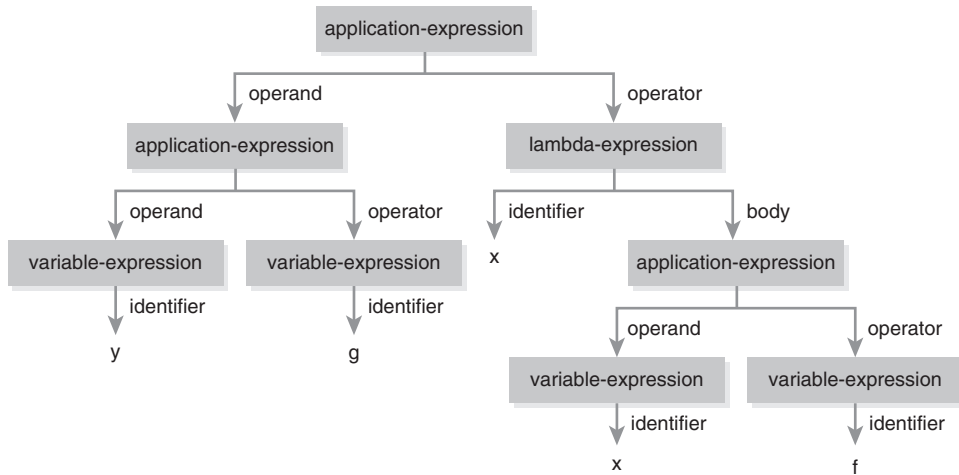


Figure 9.1 Abstract-syntax tree for `((lambda (x) (f x)) (g y))`.

Use of the `expression` data type makes other language-processing functions, such as `occurs-free?` (discussed in Chapter 6), more readable, because it eliminates cryptic and lengthy chains of calls to `car` and `cdr`:

```

(define occurs-free?
  (lambda (variable expr)
    (cases expression expr
      (variable-expression (identifier)
        (eqv? identifier variable))
      (lambda-expression (identifier body)
        (and (not (eqv? identifier variable))
              (occurs-free? variable body)))
      (application-expression (operator operand)
        (or (occurs-free? variable operator)
            (occurs-free? variable operand))))))

```

Recall, from Chapter 3, that *parsing* is the process of determining whether a string is a sentence (in some language) and, if so, typically converting the concrete representation of that sentence into an abstract representation that facilitates the intended subsequent processing. An abstract representation does not contain the details of the concrete representation that are irrelevant to the subsequent processing. The *parser* component of an interpreter or compiler typically converts the source program, once syntactically validated, into an abstract, or more easily manipulable, representation.

It is easier to (parse and) convert a list-and-symbol representation of a λ -calculus expression into abstract syntax than a string representation of the same expression. The following `concrete2abstract` function converts a concrete-syntax representation—in this case, a list-and-symbol S-expression—of a λ -calculus expression into its abstract-syntax representation:

```

(define concrete2abstract
  (lambda (expr)
    (cond

```

```

((symbol? expr) (variable-expression expr))
(pair? expr)
  (cond
    ((eqv? (car expr) 'lambda)
     (lambda-expression (caadr expr)
                        (concrete2abstract (caddr expr))))
    (else (application-expression
           (concrete2abstract (car expr))
           (concrete2abstract (cadr expr)))))
  (else (eopl:error 'concrete2abstract
                    "Invalid concrete syntax ~s" expr))))

```

Now consider an application of `concrete2abstract` to the λ -calculus expression:

```

> (concrete2abstract '((lambda (x) (f x)) (g y)))
#(struct:application-expression
  #(struct:lambda-expression
    x
    #(struct:application-expression
      #(struct:variable-expression f)
      #(struct:variable-expression x)))
  #(struct:application-expression
    #(struct:variable-expression g)
    #(struct:variable-expression y)))

```

Use of abstract syntax makes data representing code easier to manipulate and a program that processes code (i.e., programs) more readable.

9.6 Abstract-Syntax Tree for Camille

A goal of Part II of this text is to establish an understanding of data abstraction techniques so we can harness them in our construction of environment-passing interpreters, for purposes of simplicity and efficiency, in Part III.

9.6.1 Camille Abstract-Syntax Tree Data Type: `TreeNode`

The following abstract-syntax tree data type `TreeNode` is used in the abstract-syntax trees of Camille programs for our Camille interpreters developed in Part III:

```

1  import re
2  import sys
3  import operator
4  import ply.lex as lex
5  import ply.yacc as yacc
6  from collections import defaultdict
7
8  # begin expression data type #
9
10 #list of node types
11 ntPrimitive = 'Primitive'
12 ntPrimitive_op = 'Primitive Operator'
13
14 ntNumber = 'Number'
15 ntIdentifier = 'Identifier'
16
17 ntIfElse = 'Conditional'

```



```

18
19 ntArguments = 'Arguments'
20 ntFuncCall = 'Function Call'
21 ntFuncDecl = 'Function Declaration'
22 ntRecFuncDecl = 'Recursive Function Declaration'
23
24 ntParameters = 'Parameters'
25 ntExpressions = 'Expressions'
26
27 ntLetRec = 'Let Rec'
28 ntLetStar = 'Let Star'
29 ntLet = 'Let'
30
31 ntLetStatement = 'Let Statement'
32 ntLetStarStatement = 'Let* Statement'
33 ntLetRecStatement = 'Letrec Statement'
34
35 ntLetAssignment = 'Let Assignment'
36 ntLetRecAssignment = 'Letrec Assignment'
37 ntLetStarAssignment = 'Letstar Assignment'
38
39 ntAssignment = 'Assignment'
40
41 class Tree_Node:
42     def __init__(self, type, children, leaf, linenumber):
43         self.type = type
44         # save the line number of the node so run-time
45         # errors can be indicated
46         self.linenumber = linenumber
47         if children:
48             self.children = children
49         else:
50             self.children = [ ]
51             self.leaf = leaf
52 # end expression data type #

```

9.6.2 Camille Parser Generator with Tree Builder

The following code is a PLY parser generator for the Camille language.⁴ The grammar used in the parser specification is for a version of Camille used in Chapter 11. Notice that this specification contains actions to construct an abstract-syntax tree using the previous definition, used later for interpretation in Chapters 10–11:

```

118 class ParserException(Exception):
119     def __init__(self, message):
120         self.message = message
121
122     def p_error(t):
123         if (t != None):
124             raise ParserException("Syntax error: Line %d " % (t.lineno))
125         else:
126             raise ParserException("Syntax error near: Line %d" %
127                                   (lexer.lineno - (lexer.lineno > 1)))
128

```

4. The PLY lexical specification is not shown here; lines 8–72 of the lexical specification shown in Section 3.6.2 can be used here as lines 53–117.

```

129 # begin syntactic specification #
130 def p_program_expr(t):
131     '''programs : program programs
132         | program'''
133     #do nothing
134
135 def p_line_expr(t):
136     '''program : expression'''
137     t[0] = t[1]
138     global global_tree
139     global_tree = t[0]
140
141 def p_primitive_op(t):
142     '''expression : primitive LPAREN expressions RPAREN'''
143     t[0] = Tree_Node( ntPrimitive_op, [t[3]], t[1], t.lineno(1))
144
145 def p_primitive(t):
146     '''primitive : PLUS
147         | MINUS
148         | INC1
149         | MULT
150         | DEC1
151         | ZERO
152         | EQV'''
153     t[0] = Tree_Node(ntPrimitive, None, t[1], t.lineno(1))
154
155 def p_expression_number(t):
156     '''expression : NUMBER'''
157     t[0] = Tree_Node(ntNumber, None, t[1], t.lineno(1))
158
159 def p_expression_identifer(t):
160     '''expression : IDENTIFIER'''
161     t[0] = Tree_Node(ntIdentifer, None, t[1], t.lineno(1))
162
163 def p_expression_let(t):
164     '''expression : LET let_statement IN expression'''
165     t[0] = Tree_Node(ntLet, [t[2], t[4]], None, t.lineno(1))
166
167 def p_expression_let_star(t):
168     '''expression : LETSTAR letstar_statement IN expression'''
169     t[0] = Tree_Node(ntLetStar, [t[2], t[4]], None, t.lineno(1))
170
171 def p_expression_let_rec(t):
172     '''expression : LETREC letrec_statement IN expression'''
173     t[0] = Tree_Node(ntLetRec, [t[2], t[4]], None, t.lineno(1))
174
175 def p_expression_condition(t):
176     '''expression : IF expression expression ELSE expression'''
177     t[0] = Tree_Node(ntIfElse, [t[2], t[3], t[5]], None, t.lineno(1))
178
179 def p_expression_function_decl(t):
180     '''expression : FUN LPAREN parameters RPAREN expression
181         | FUN LPAREN RPAREN expression'''
182     if len(t)==6:
183         t[0] = Tree_Node(ntFuncDecl, [t[3], t[5]], None, t.lineno(1))
184     else:
185         t[0] = Tree_Node(ntFuncDecl, [t[4]], None, t.lineno(1))
186
187 def p_expression_function_call(t):
188     '''expression : LPAREN expression arguments RPAREN
189         | LPAREN expression RPAREN '''
190     if len(t)== 5:
191         t[0] = Tree_Node(ntFuncCall, [t[3]], t[2], t.lineno(1))

```

```

192     else:
193         t[0] = Tree_Node(ntFuncCall, None, t[2], t.lineno(1))
194
195     def p_expression_rec_func_decl(t):
196         '''rec_func_decl : FUN LPAREN parameters RPAREN expression
197                        | FUN LPAREN RPAREN expression'''
198         if len(t)==6:
199             t[0] = Tree_Node(ntRecFuncDecl, [t[3], t[5]], None, t.lineno(1))
200         else:
201             t[0] = Tree_Node(ntRecFuncDecl, [t[4]], None, t.lineno(1))
202
203     def p_parameters(t):
204         '''parameters : IDENTIFIER
205                        | IDENTIFIER COMMA parameters'''
206         if len(t) == 4:
207             t[0] = Tree_Node(ntParameters, [t[1], t[3]], None, t.lineno(1))
208         elif len(t) == 2:
209             t[0] = Tree_Node(ntParameters, [t[1]], None, t.lineno(1))
210
211     def p_arguments(t):
212         '''arguments : expression
213                        | expression COMMA arguments'''
214         if len(t) == 2:
215             t[0] = Tree_Node(ntArguments, [t[1]], None, t.lineno(1))
216         elif len(t) == 4:
217             t[0] = Tree_Node(ntArguments, [t[1], t[3]], None, t.lineno(1))
218
219     def p_expressions(t):
220         '''expressions : expression
221                        | expression COMMA expressions'''
222         if len(t) == 4:
223             t[0] = Tree_Node(ntExpressions, [t[1], t[3]], None, t.lineno(1))
224         elif len(t) == 2:
225             t[0] = Tree_Node(ntExpressions, [t[1]], None, t.lineno(1))
226
227     def p_let_statement(t):
228         '''let_statement : let_assignment
229                        | let_assignment let_statement'''
230         if len(t) == 3:
231             t[0] = Tree_Node(ntLetStatement, [t[1], t[2]], None, t.lineno(1))
232         else:
233             t[0] = Tree_Node(ntLetStatement, [t[1]], None, t.lineno(1))
234
235     def p_letstar_statement(t):
236         '''letstar_statement : letstar_assignment
237                        | letstar_assignment letstar_statement'''
238         if len(t) == 3:
239             t[0] = Tree_Node(ntLetStarStatement, [t[1], t[2]], None,
240                                     t.lineno(1))
241         else:
242             t[0] = Tree_Node(ntLetStarStatement, [t[1]], None, t.lineno(1))
243
244     def p_letrec_statement(t):
245         '''letrec_statement : letrec_assignment
246                        | letrec_assignment letrec_statement'''
247         if len(t) == 3:
248             t[0] = Tree_Node(ntLetRecStatement, [t[1], t[2]], None, t.lineno(1))
249         else:
250             t[0] = Tree_Node(ntLetRecStatement, [t[1]], None, t.lineno(1))
251
252     def p_let_assignment(t):
253         '''let_assignment : IDENTIFIER EQ expression'''
254         t[0] = Tree_Node(ntLetAssignment, [t[3]], t[1], t.lineno(1))

```

TreeNode	type: node type (e.g., ntNumber)	TreeNode	type: ntIdentifier
	leaf: primary data associated with node		leaf: x
	children: list of child nodes		children: []
	linenumber: line number in which the node occurs		linenumber: l

Figure 9.2 (left) Visual representation of `TreeNode` Python class. (right) A value of type `TreeNode` for an identifier.

```

255
256 def p_letstar_assignment(t):
257     '''letstar_assignment : IDENTIFIER EQ expression'''
258     t[0] = Tree_Node(ntLetStarAssignment, [t[3]], t[1], t.lineno(1))
259
260 def p_letrec_assignment(t):
261     '''letrec_assignment : IDENTIFIER EQ rec_func_decl'''
262     t[0] = Tree_Node(ntLetRecAssignment, [t[3]], t[1], t.lineno(1))
263 # end syntactic specification #

```

This Camille parser generator in PLY is the same as that shown in Section 3.6.2, but contains actions to build the abstract-syntax tree (AST) in the pattern-action rules. Specifically, the Camille parser builds an AST in which each node contains the node type, a leaf, a list of children, and a line number. The `TreeNode` structure is shown on the left side of Figure 9.2. For all number (`ntNumber`), identifier (`ntIdentifier`), and primitive operator (`ntPrimitive`) node types, the value of the token is stored in the leaf of the node (shown on the right side of Figure 9.2). In the `p_line_expr` function (lines 135–139), notice that the final abstract-syntax tree is assigned to the global variable `global_tree` (line 139) so that it can be referenced by the function that invokes the parser—namely, the following `concrete2abstract` function, which is the Python analog of the `concrete2abstract` Racket Scheme function given in Section 9.5:

```

264 global_tree = ""
265
266 def concrete2abstract(s, parser):
267     pattern = re.compile("[^ \t]+")
268     if pattern.search(s):
269         try:
270             parser.parse(s)
271             global global_tree
272             return global_tree
273         except Exception as e:
274             print ("Unknown Error occurred "
275                  "(this is normally caused by a syntax error)")
276             raise e
277     return None
278
279 def main_func():
280     parser = yacc.yacc()
281     interactiveMode = False
282
283     if len(sys.argv) == 1:
284         interactiveMode = True

```

```

285
286     if interactiveMode:
287         program = ""
288         try:
289             prompt = 'Camille> '
290             while True:
291                 line = input(prompt)
292                 if (line == "" and program != ""):
293                     print(concrete2abstract(line,parser))
294                     lexer.lineno = 1
295                     program = ""
296                     prompt = 'Camille> '
297                 else:
298                     if (line != ""):
299                         program += (line + '\n')
300                         prompt = ''
301
302             except EOFError as e:
303                 sys.exit(0)
304
305             except Exception as e:
306                 print(e)
307                 sys.exit(-1)
308         else:
309             try:
310                 with open(sys.argv[1], 'r') as script:
311                     file_string = script.read()
312                     print(concrete2abstract(file_string,parser))
313                     sys.exit(0)
314             except Exception as e:
315                 print(e)
316                 sys.exit(-1)
317
318     main_func()

```

Examples:

```

$ python3.8 camilleAST.py
Camille> let a=5 in a
<camilleTreeStruct.Tree_Node object at 0x104c6d820>
Camille> let a = 5 in a
<camilleTreeStruct.Tree_Node object at 0x104c6dac0>
Camille> let a=2 in let b=3 in a
<camilleTreeStruct.Tree_Node object at 0x104c6dfd0>
Camille> let f = fun (y, z) +(y,-(z,5)) in (f 2, 28)
<camilleTreeStruct.Tree_Node object at 0x104c6da30>

```

Notice that facilities to convert between concrete and abstraction representations of programs (e.g., the `concrete2abstract` function) are unnecessary in a homoiconic language. Since programs written in a homoiconic language are directly expressed as data objects in that language, they are already in an easily manipulable format. (See also the `occurs-free?` and `occurs-bound?` functions in Section 6.6.)

Programming Exercises for Sections 9.5 and 9.6

Exercise 9.6.1 Consider the following definition of a data type expression in Racket Scheme:

```
(define-datatype expression expression?
  (literal-expression
    (literal_tag number?))
  (variable-expression
    (identifier symbol?))
  (conditional-expression
    (clauses (list-of expression?)))
  (lambda-expression
    (identifiers (list-of symbol?))
    (body expression?))
  (application-expression
    (operator expression?)
    (operands (list-of expression?))))
```

The following function `list-of` used in the definition of the data type is defined in Section 5.10.3 and repeated here:

```
(define list-of
  (lambda (predicate)
    (letrec ((list-of-helper
              (lambda (lst)
                (or (null? lst)
                    (and (pair? lst)
                        (predicate (car lst))
                        (list-of-helper (cdr lst)))))))
      list-of-helper)))
```

This function is also built into the `#lang eopl` language of *DrRacket*.

Define a function `abstract2concrete` that converts an *abstract-syntax representation* of a λ -calculus expression (using the expression data type given here) into a *concrete-syntax* (i.e., list-and-symbol) representation of it.

Exercise 9.6.2 Define a function `abstract2concrete` that converts an *abstract-syntax representation* of an expression (using the `TreeNode` data type given in Section 9.6.1) into a *concrete-syntax* (i.e., a string) representation of it. The function `abstract2concrete` maps a value of the `TreeNode` data type of a Camille expression into a concrete-syntax representation (in this case, a string) of it. To test the correctness of your `abstract2concrete` function, replace lines 293 and 312 in `main_func` with:

```
print(abstract2concrete(concrete2abstract(program, parser)))
```

Examples:

```
$ python3.8 camilleAST.py
Camille> let a = 5 in a
let a = 5 in a
Camille> let a=2 in let b=3 in a
let a = 2 in let b = 3 in a
Camille> let f = fun (y, z) + (y, -(z, 5)) in (f 2, 28)
let f = fun(y, z) + (y, -(z, 5)) in (f 2, 28)
Camille>
```

9.7 Data Abstraction

Data abstraction involves the conception and use of a data structure as:

- an *interface*, which is *implementation-neutral* and contains function *declarations*;
- an *implementation*, which contains function *definitions*; and
- an *application*, which is also *implementation-neutral* and contains *invocations* to functions in the implementation; the application is sometimes called the *main program* or *client code*.

The underlying implementation can change without disrupting the client code as long as the contractual signature of each function declaration in the interface remains unchanged. In this way, the implementation is *hidden* from the application. A data type developed this way is called an *abstract data type* (ADT). Consider a list abstract data type. One possible representation for the list used in the implementation might be an array or vector. Another possible representation might be a linked list. (Note that Church Numerals are a representation of numbers in λ -calculus; see Programming Exercise 5.2.2.) A goal of a type system is to support the definition of abstract data types that have the properties and behavior of primitive types. One advantage of using an ADT is that the application is independent of the representation of the data structure used in the implementation. In turn, any implementation of the interface can be substituted without requiring modifications to the client application. In Section 9.8, we demonstrate a variety of possible representations for an environment ADT, all of which satisfy the requirements for the interface of the abstract data type and, therefore, maintain the integrity of the independence between the representation and the application.

9.8 Case Study: Environments

Recall from Chapter 6 that a *referencing environment* is a mapping that associates variable names (or symbols) with their current bindings at any point in a program in an implementation of a programming language (e.g., $\{(a, 4), (b, 2), (c, 3), (x, 5)\}$). Consider an interface specification of an environment, where formally an environment expressed in the mathematical form $\{(s_1, v_1), (s_2, v_2), \dots, (s_n, v_n)\}$ is a *mapping* (or a set of pairs) from the *domain*—the finite set of Scheme symbols—to the *range*—the set of all Scheme values:

$$\begin{aligned} \text{(empty-environment)} &= [\emptyset] \\ \text{(apply-environment } [f] s) &= f(s) \\ \text{(extend-environment ' } (s_1, s_2, \dots, s_n) \text{ ' } (v_1, v_2, \dots, v_n) [f]) &= [g], \end{aligned}$$

where $g(s') = v_i$ if $s' = s_i$ for some i , $1 \leq i \leq n$, and $f(s')$ otherwise; and $[v]$ means “the representation of data v .”

The environment $\{(a, 4), (b, 2), (c, 3), (x, 5)\}$ may be constructed and accessed with the following client code:

```
> (define simple-environment
  (extend-environment '(a b) '(1 2)
    (extend-environment '(c d e) '(3 5 5)
      (empty-environment))))
> (apply-environment simple-environment 'e)
5
```

Here the *constructors* are `empty-environment` and `extend-environment`, which each *create* an environment. The *observer*, which *extracts* from an environment, is `apply-environment`.

9.8.1 Choices of Representation

We consider the following representations for an environment:

- data structure representation (e.g., lists)
- abstract-syntax representation (ASR)
- closure representation (CLS)

We have already discussed list and abstract-syntax representations—though not for representing environments. (We briefly discussed a list representation for an environment in Chapter 6.) We will leave abstract-syntax representations of environments and list representations of environments in Racket Scheme as exercises (Programming Exercises 9.8.3 and 9.8.4, respectively) and focus on a closure representation of abstract data types here. Specifically, we discuss a closure representation of an environment because it is not only perhaps the most interesting of these representations, but also probably the least familiar for readers.

9.8.2 Closure Representation in Scheme

Often the set of *values* of a data type can be advantageously *represented* as a set of *functions*, particularly when the abstract data type has multiple constructors but only a single observer. Moreover, languages with first-class functions, such as Scheme, facilitate use of a closure representation. Representing a data structure as a function—here, a closure—is a non-intuitive use of functions, because we do not typically think of data as code.⁵

Analogous to our cognitive shift from thinking imperatively to thinking functionally in the conception of a program, here we must consider how we might represent an environment (which we think of as a data structure) as a function (which we think of as code). This cognitive shift is natural because an environment, like a function, is a mapping. However, representing, for example, a stack as a function is less natural (Programming Exercise 9.8.1). The most natural closure representation for the environment is a Scheme closure that accepts a symbol and returns its associated value. With such a representation, we can define the interface functionally in the following implementation:

5. In the von Neumann architecture, we think of and represent code as data; in other words, code and data are represented uniformly in main memory.


```

#lang eopl
;;; closure representation of environment
(define empty-environment
  (lambda ()
    (lambda (identifier)
      (eopl:error 'apply-environment
        "No binding for ~s" identifier))))
(define extend-environment
  (lambda (identifiers values environ)
    (lambda (identifier)
      (let ((position (list-find-position identifier identifiers)))
        (cond
          ((number? position) (list-ref values position))
          (else (apply-environment environ identifier)))))))
(define apply-environment
  (lambda (environ identifier)
    (environ identifier)))
(define list-find-position
  (lambda (identifier los)
    (list-index
      (lambda (identifier1) (eqv? identifier1 identifier)) los)))
(define list-index
  (lambda (predicate ls)
    (cond
      ((null? ls) #f)
      ((predicate (car ls)) 0)
      (else (let ((list-index-r
                    (list-index predicate (cdr ls))))
                (cond
                  ((number? list-index-r) (+ list-index-r 1))
                  (else #f)))))))

```

Getting acclimated to the reality that the data structure *is* a function can be a cognitive challenge. One way to get accustomed to this representation is to reify the function representing an environment every time one is created or extended and unpack it every time one is applied (i.e., accessed). For instance, let us step through the evaluation of the following application code:

```

1 > (define simple-environment
2   (extend-environment '(a b) '(1 2)
3   (extend-environment '(c d e) '(3 4 5)
4   (empty-environment))))
5
6 > (apply-environment simple-environment 'e)
7 5

```

First, the expression `(empty-environment)` (line 4) is evaluated and returns

```

(lambda (symbol)
  (eopl:error 'apply-environment "No binding for ~s" symbol))

```

Here, `eopl:error` is a facility for printing error messages in the *Essentials of Programming Languages* language. Thus, we have

```

1 (define simple-environment
2   (extend-environment '(a b) '(1 2)
3   (extend-environment '(c d e) '(3 4 5)
4   (lambda (symbol)

```

```

5          (eopl:error 'apply-environment
6            "No binding for ~s" symbol))))))

```

Next, the expression on lines 3–6 is evaluated and returns

```

(lambda (symbol)
  (let ((position (list-find-position symbol '(c d e))))
    (cond
      ((number? position)
       (list-ref '(3 4 5) position))
      (else (apply-environment
              (lambda (symbol)
                (eopl:error 'apply-environment
                  "No binding for ~s"
                  symbol))
              symbol))))))

```

Thus, we have

```

1  (define simple-environment
2    (extend-environment '(a b) '(1 2)
3      (lambda (symbol)
4        (let ((position
5              (list-find-position symbol '(c d e))))
6          (cond
7            ((number? position) (list-ref '(3 4 5) position))
8            (else (apply-environment
9                  (lambda (symbol)
10                     (eopl:error 'apply-environment
11                       "No binding for ~s" symbol))
12                     symbol)))))))

```

Next, the expression on lines 2–12 is evaluated and returns

```

(lambda (symbol)
  (let ((position (list-find-position symbol '(a b))))
    (cond
      ((number? position) (list-ref '(1 2) position))
      (else (apply-environment
              (lambda (symbol)
                (let ((position
                      (list-find-position symbol '(c d e))))
                  (cond
                    ((number? position)
                     (list-ref '(3 4 5) position))
                    (else (apply-environment
                          (lambda (symbol)
                            (eopl:error
                              'apply-environment
                              "No binding for ~s"
                              symbol))
                          symbol))))
              symbol))))))

```

Thus, we have

```

1  (define simple-environment
2    (lambda (symbol)
3      (let ((position (list-find-position symbol '(a b))))

```

```

4      (cond
5        ((number? position) (list-ref '(1 2) position))
6        (else (apply-environment
7              (lambda (symbol)
8                (let ((position
9                      (list-find-position symbol '(c d e))))
10                 (cond
11                   ((number? position)
12                    (list-ref '(3 4 5) position))
13                   (else (apply-environment
14                          (lambda (symbol)
15                            (eopl:error
16                              'apply-environment
17                                "No binding for ~s"
18                                  symbol))
19                            symbol))))))
20              symbol))))))
21

```

The identifiers `list-find-position` and `list-ref` are also expanded to their function bindings, but, for purposes of simplicity of presentation, we omit such expansions as they are not critical to the idea at hand. Finally, the `lambda` expression on lines 2–20 representing the simple environment is stored in the Racket Scheme environment under the symbol `simple-environment`.

To evaluate `(apply-environment simple-environment 'e)`, we must unpack this `lambda` expression representing the simple environment. The expression `(apply-environment simple-environment 'e)` evaluates to

```

(apply-environment
 (lambda (symbol)
   (let ((position (list-find-position symbol '(a b))))
     (cond
      ((number? position) (list-ref '(1 2) position))
      (else (apply-environment
              (lambda (symbol)
                (let ((position
                      (list-find-position symbol '(c d e))))
                  (cond
                   ((number? position)
                    (list-ref '(3 4 5) position))
                   (else (apply-environment
                           (lambda (symbol)
                             (eopl:error
                               'apply-environment
                                "No binding for ~s"
                                  symbol))
                             symbol))))))
              symbol))))))
 'e)

```

Given our definition of the `apply-environment` function, this expression, when evaluated, returns

```

1 ((lambda (symbol)
2   (let ((position (list-find-position symbol '(a b))))
3     (cond
4       ((number? position) (list-ref '(1 2) position))

```

```

5      (else (apply-environment
6
7          (lambda (symbol)
8              (let ((position
9                  (list-find-position symbol '(c d e))))
10                 (cond
11                     ((number? position)
12                      (list-ref '(3 4 5) position))
13                     (else (apply-environment
14                             (lambda (symbol)
15                                 (eopl:error
16                                  'apply-environment
17                                   "No binding for ~s"
18                                    symbol))
19                             symbol))))))
20      symbol))))
21  'e)

```

Since the symbol `e` (line 21) is not found in the list of symbols in the outermost environment `'(a b)` (line 2), this expression, when evaluated, returns

```

(apply-environment
 (lambda (symbol)
   (let ((position (list-find-position symbol '(c d e))))
     (cond
      ((number? position) (list-ref '(3 4 5) position))
      (else (apply-environment
              (lambda (symbol)
                (eopl:error 'apply-environment
                             "No binding for ~s" symbol))
              symbol))))))
 'e)

```

This expression, when evaluated, returns

```

1  ((lambda (symbol)
2    (let ((position (list-find-position symbol '(c d e))))
3      (cond
4        ((number? position) (list-ref '(3 4 5) position))
5        (else (apply-environment
6                (lambda (symbol)
7                  (eopl:error 'apply-environment
8                               "No binding for ~s" symbol))
9                symbol))))))
10  'e)

```

Since the symbol `'e` (line 10) is found in the list of symbols in the intermediate environment `'(c d e)` (line 2) at position 2, this expression, when evaluated, returned `(list-ref '(3 4 5) position)`, which, when evaluated, returns 5. This example brings us face to face with the fact that a program is nothing more than data. In turn, a data structure can be represented as a program.

9.8.3 Closure Representation in Python

Since Python supports first-class closures, we can replicate our closure representation of an environment data structure in Scheme in Python:

```

# begin closure representation of environment #
def empty_environment():
    def raise_IE():
        raise IndexError
    return lambda symbol: raise_IE()

def apply_environment(environment, symbol):
    return environment(symbol)

def extend_environment(symbols, values, environment):
    def tryexcept(symbol):
        try:
            val = values[symbols.index(symbol)]
        except:
            val = apply_environment(environment, symbol)
        return val
    return lambda symbol: tryexcept(symbol)
# end closure representation of environment #

simple_env = extend_environment(["a", "b"], [1, 2],
    extend_environment(["b", "c", "d"], [3, 4, 5],
        empty_environment()))

>>> print (apply_environment(simple_env, "d"))
5
>>> print (apply_environment(simple_env, "b"))
2
>>> print (apply_environment(simple_env, "e"))
No binding for symbol e.

```

We can extract the interface for and the (closure representation) implementation of an ADT from the application code:

1. Identify all of the `lambda` expressions in the application code whose evaluation yields values of the data type. Define a constructor function for each such `lambda` expression. The parameters of the constructor are the free variables of the `lambda` expression. Replace each of these `lambda` expressions in the application code with an invocation of the corresponding constructor.
2. Define an observer function such as `apply-environment`. Identify all the points in the application code, including the bodies of the constructors, where a value of the type is applied. Replace each of these applications with an invocation of the observer function (Friedman, Wand, and Haynes 2001, p. 58).

If we do this, then

- the interface consists of the constructor functions and the observer function
- the application is independent of the representation
- we are free to substitute any other implementation of the interface without breaking the application code (Friedman, Wand, and Haynes 2001, p. 58)

9.8.4 Abstract-Syntax Representation in Python

We can also build abstract-syntax representations (discussed in Section 9.5) of data structures (as in Programming Exercise 9.8.3). The following code is an abstract-syntax representation of the environment in Python (Figure 9.3).

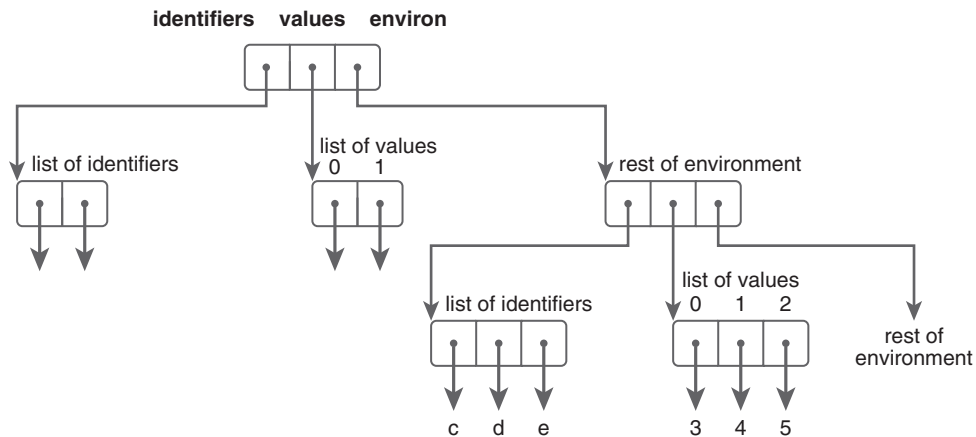


Figure 9.3 An abstract-syntax representation of a named environment in Python.

```
# begin abstract-syntax representation of environment #
class Environment:
    def __init__(self, symbols=None, values=None, environ=None):
        if symbols == None and values == None and environ == None:
            self.flag = "empty-environment-record"
        else:
            self.flag = "extended-environment-record"
            self.symbols = symbols
            self.values = values
            self.environ = environ
    def empty_environment():
        return Environment()
    def extend_environment(symbols, values, environ):
        return Environment(symbols, values, environ)
    def apply_environment(envirion, symbol):
        if envirion.flag == "empty-environment-record":
            return "No binding for symbol " + symbol + "."
        else:
            try:
                return envirion.values[envirion.symbols.index(symbol)]
            except:
                return apply_environment(envirion.environ, symbol)
# end abstract-syntax representation of environment #
simple_env = extend_environment(["a", "b"], [1, 2],
    extend_environment(["b", "c", "d"], [3, 4, 5],
        empty_environment()))
>>> print (apply_environment(simple_env, "d"))
5
>>> print (apply_environment(simple_env, "b"))
2
>>> print (apply_environment(simple_env, "e"))
No binding for symbol e.
```

Programming Exercises for Sections 9.7 and 9.8

Exercise 9.8.1 (Friedman, Wand, and Haynes 2001, Exercise 2.15, p. 58) Consider a *stack* data type with the interface:

$$\begin{aligned}
(\text{empty-stack}) &= [S], \text{ where } (\text{empty-stack? } [S]) = \#t \\
(\text{empty-stack? } [S]) &= \{\#t \text{ if } [S] = (\text{empty-stack}), \text{ and} \\
&\quad \#f \text{ otherwise}\} \\
(\text{push } (\text{pop } [S]) \ e) &= [S] \\
(\text{pop } (\text{push } [S] \ e)) &= [S] \\
(\text{top } (\text{push } [S] \ e)) &= e
\end{aligned}$$

where $[v]$ means “the representation of data v .”

Example client code:

```
> (top (pop (push "hello" (push 1 (push 2 (push (+ 1 2)
                                                (empty-stack)))))))
1
```

Implement this interface in Scheme using a *closure representation* for the stack. The functions `empty-stack` and `push` are the *constructors*, and the functions `pop`, `top`, and `empty-stack?` are the *observers*. Therefore, the closure representation of the stack must take only a single atom argument and use it to determine which *observation* to make. Call this parameter message. The messages can be the atoms `'empty-stack?`, `'top`, or `'pop`. The implementation requires approximately 20 lines of code.

Exercise 9.8.2 Solve Programming Exercise 9.8.1 using `lambda` expressions in Python.

Example client code:

```
>>> print top(pop(push("hello", push(1, push(2, push(1+2,
                                                    empty_stack()))))))
1
```

The remaining programming exercises deal with the implementation of a variety of representations (e.g., `abstract-syntax`, `list`, and `closure`) for environments. Tables 9.3 and 9.4 summarize the representations and languages used in these programming exercises.

Exercise 9.8.3 (Friedman, Wand, and Haynes 2001) Define and implement in Racket Scheme an `abstract-syntax` representation of the environment shown in Section 9.8 (Figure 9.4).

(a) Define a grammar in EBNF (i.e., a concrete syntax) that defines a language of environment expressions in the following form:

```
(extend-environment symbols_n values_n
  (extend-environment symbols_{n-1} values_{n-1}
    ...
    (extend-environment symbols_i values_i
      ...
      (extend-environment symbols_2 values_2
        (extend-environment symbols_1 values_1
          (empty-environment))))))
```

Programming Exercise/Section	Representation	Environment	Language	Figure
PE 9.8.3	ASR	named	Racket Scheme	9.4
Section 9.8.4	ASR	named	Python	9.3
PE 9.8.4.c	LOLR	named	(Racket) Scheme	9.5
PE 9.8.5.a	LOLR	named	Python	9.7
Section 9.8.2	CLS	named	(Racket) Scheme	—
Section 9.8.3	CLS	named	Python	—
PE 9.8.8	ASR	nameless	Racket Scheme	9.9
PE 9.8.9	ASR	nameless	Python	9.10
PE 9.8.4.d	LOVR	nameless	(Racket) Scheme	9.6
PE 9.8.5.b	LOLR	nameless	Python	9.8
PE 9.8.6	CLS	nameless	(Racket) Scheme	—
PE 9.8.7	CLS	nameless	Python	—

Table 9.3 Summary of the Programming Exercises in This Chapter Involving the Implementation of a Variety of Representations for an Environment (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation; and PE = programming exercise.)

	Named	Nameless
(Racket) Scheme	CLS (Section 9.8.2) ASR (Figure 9.4; PE 9.8.3) LOLR (Figure 9.5; PE 9.8.4.c)	CLS (PE 9.8.6) ASR (Figure 9.9; PE 9.8.8) LOVR (Figure 9.6; PE 9.8.4.d)
Python	CLS (Section 9.8.3) ASR (Section 9.8.4; Figure 9.3) LOLR (Figure 9.7; PE 9.8.5.a)	CLS (PE 9.8.7) ASR (Figure 9.10; PE 9.8.9) LOLR (Figure 9.8; PE 9.8.5.b)

Table 9.4 The Variety of Representations of Environments in Racket Scheme and Python Developed in This Chapter (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation; and PE = programming exercise.)

Specifically, complete the following grammar definition:

```

<environment> ::=
<environment> ::=

```

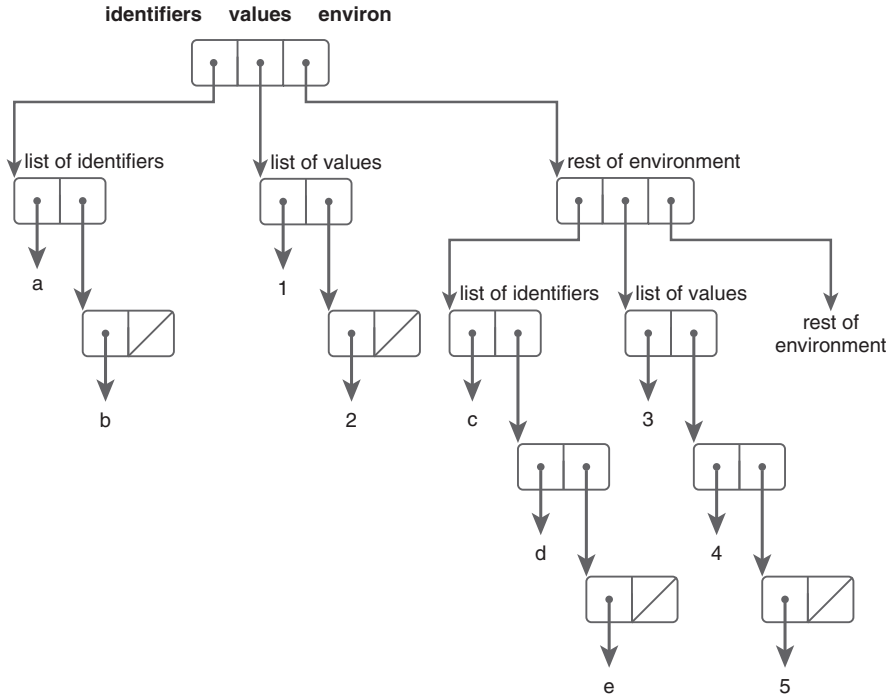



Figure 9.4 An abstract-syntax representation of a named environment in Racket Scheme using the structure of Programming Exercise 9.8.3.

- (b) Annotate that grammar (i.e., concrete syntax) with abstract syntax as shown at the beginning of Section 9.5 for λ -calculus; in other words, represent it as an abstract syntax.
- (c) Define the `environment` data type using `(define-datatype ...)`. You may use the function `list-of`, which is given in Programming Exercise 9.6.1.
- (d) Define the implementation of this environment; that is, define the `empty-environment`, `extend-environment`, and `apply-environment` functions. Use the function `rib-find-position` in your implementation:

```
(define list-find-position
  (lambda (symbol los)
    (list-index (lambda (symbol1) (eqv? symbol1 symbol)) los)))

(define list-index
  (lambda (predicate ls)
    (cond
      ((null? ls) #f)
      ((predicate (car ls)) 0)
      (else (let ((list-index-r
                    (list-index predicate (cdr ls))))
              (cond
                ((number? list-index-r) (+ list-index-r 1))
                (else #f)))))))

(define rib-find-position list-find-position)
```

Programming Exercise	Representation	Figure	Example of Representation
9.8.4.a	LOLR (rib: list of 2 lists)	—	(((a b) (1 2)) ((c d e) (3 4 5)))
9.8.4.b	LOLR (rib: list of lists and vector)	—	(((a b) #(1 2)) ((c d e) #(3 4 5)))
9.8.4.c	LOLR (rib: list of pair of lists and vector)	9.5	(((a b) . #(1 2)) ((c d e) . #(3 4 5)))
9.8.4.d	LOVR (rib: vector)	9.6	(#(1 2) #(3 4 5))

Table 9.5 List-of-Lists/Vectors Representations of an Environment Used in Programming Exercise 9.8.4

Exercise 9.8.4 (Friedman, Wand, and Haynes 2001) In this programming exercise you implement a list representation of an environment in Scheme and make three progressive improvements to it (Table 9.5). Start with the solution to Programming Exercise 9.8.3.a.

- (a) Implement the grammar defined in Programming Exercise 9.8.3.a. In this representation, the empty environment is represented by an empty list and constructed from the `empty-environment` function. A non-empty environment is represented by a list-of-lists and constructed from the `extend-environment` function, where the `car` of the list is a list representing the outermost environment (created by `extend-environment`) and the `cdr` is the list representing the next inner environment.

Example client code:

```
> (define abcd-environ
  (extend-environment '(a b) '(1 2)
    (extend-environment '(b c d) '(3 4 5)
      (empty-environment))))
> abcd-environ
( ((a b) (1 2)) ((b c d) (3 4 5)) )
```

This is called the *ribcage representation* (Friedman, Wand, and Haynes 2001). The environment is represented by a list of lists. The lists contained in the environment list are called *ribs*. The `car` of each rib is a list of symbols, and the `cadr` of each rib is the corresponding list of values. Define the implementation of this environment; that is, define the `empty-environment` and `extend-environment` functions. Use the functions `list-find-position` and `list-index`, shown in Chapter 10, in your implementation. Also, use the following definition:

```
(define rib-find-position list-find-position)
```

We call this particular implementation of the ribcage representation the *list-of-lists representation* (LOLR) of a named environment.

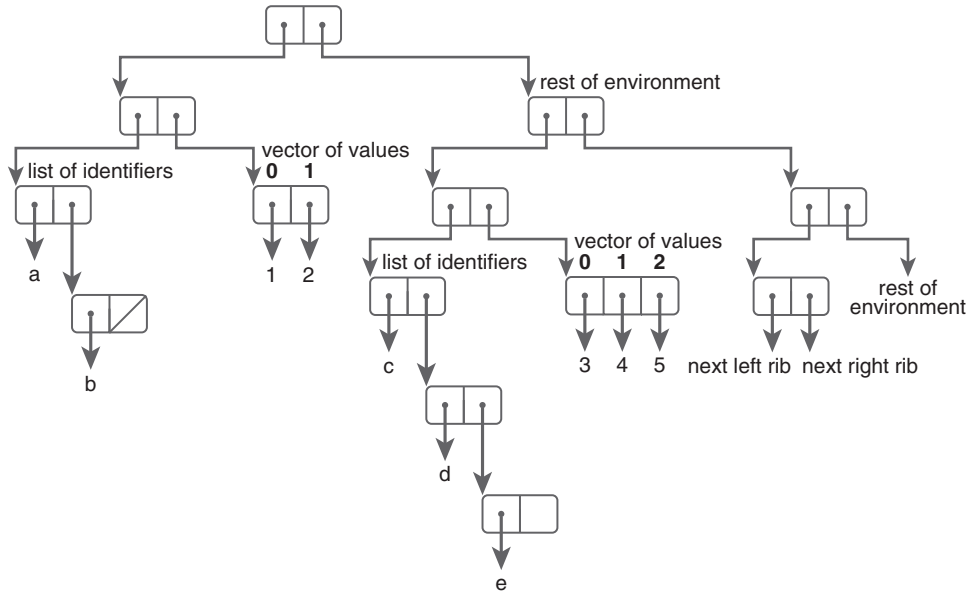


Figure 9.5 A list-of-lists representation of a named environment in Scheme using the structure of Programming Exercise 9.8.4.c.

- (b) Improve the efficiency of access in the solution to (a) by using a vector for the value of each rib instead of a list:

```
> abcd-environ
( ((a b) #(1 2)) ((b c d) #(3 4 5)) )
```

Lookup in a list through `(list-ref ...)` requires linear time, whereas lookup in a vector through `(vector-ref ...)` requires constant time. The `list->vector` function can be used to convert a list to a vector.

- (c) Improve the efficiency of access in the solution to (b) by changing the representation of a rib from a list of two elements to a single pair—so that the values of each rib can be accessed simply by taking the `cdr` of the rib rather than the `car` of the `cdr` (Figure 9.5):

```
> abcd-environ
( ((a b) . #(1 2)) ((b c d) . #(3 4 5)) )
```

- (d) If lookup in an environment is based on *lexical distance* information, then we can eliminate the symbol list from each rib in the representation and represent environments simply as a list of vectors (Figure 9.6)—so that the values of each rib can be accessed simply by taking the `cdr` of the rib:

```
> abcd-environ
( #(1 2) #(3 4 5) )
```

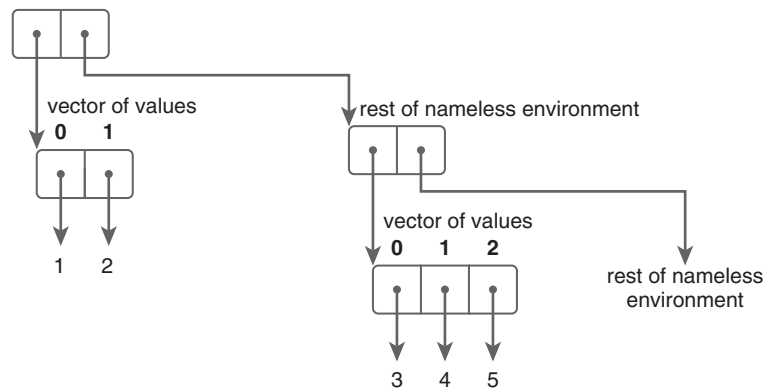


Figure 9.6 A list-of-vectors representation of a nameless environment in Scheme using the structure of Programming Exercise 9.8.4.d.

Improve the solution to (c) to incorporate this optimization. Use the following interface for the nameless environment:

```
(define empty-nameless-environment
  (lambda ()
    ...))
(define extend-nameless-environment
  (lambda (values environ)
    ...))
(define apply-nameless-lexical-environment
  (lambda (environ depth position)
    ...))
```

We call this particular implementation of the ribcage representation the *list-of-vectors representation* (LOVR) of a nameless environment.

Exercise 9.8.5 In this programming exercise, you build two different ribcage representations of the environment in Python (Table 9.6).

(a) (*list-of-lists representation of a named environment*) Complete Programming Exercise 9.8.4.a in Python (Figure 9.7). Since Python does not support function names containing a hyphen, replace each hyphen in the function names in the environment interface with an underscore, as shown in the closure

Programming Exercise	Representation	Figure	Example of Representation
9.8.5.a	LOLR (rib: list of 2 lists)	9.7	[[[a b] [1 2]] [[c d e] [3 4 5]]]
9.8.5.b	LOLR (rib: list of values)	9.8	[[1 2] [3 4 5]]

Table 9.6 List-of-Lists Representations of an Environment Used in Programming Exercise 9.8.5

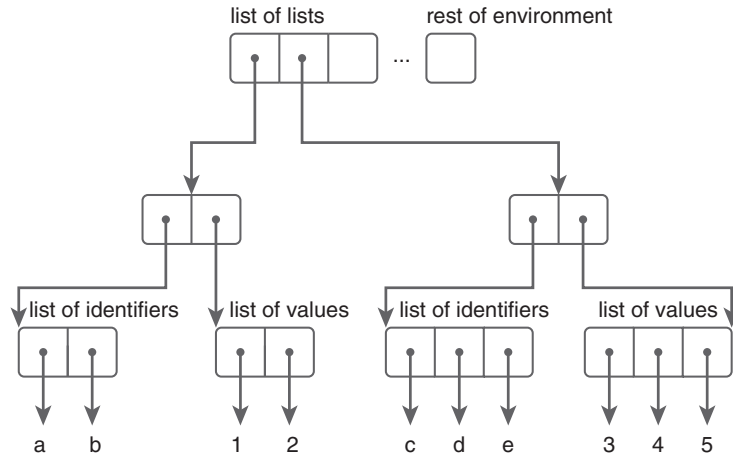


Figure 9.7 A list-of-lists representation of a named environment in Python using the structure of Programming Exercise 9.8.5.a.

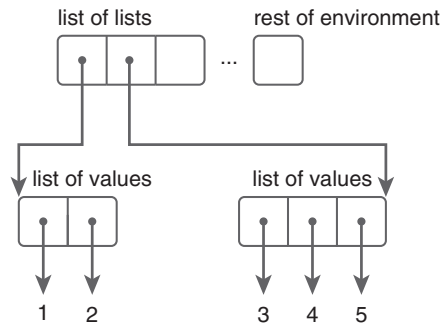


Figure 9.8 A list-of-lists representation of a nameless environment in Python using the structure of Programming Exercise 9.8.5.b.

representation of an environment in Python shown in Section 9.8.4. Also, note that *lists* in Python are used and accessed as if they were *vectors*, rather than like lists in Scheme, ML, or Haskell. In particular, unlike lists used in functional programming, the individual elements of lists in Python can be directly accessed through an integer index in constant time.

- (b) (Friedman, Wand, and Haynes 2001, Exercise 3.25, p. 90) (*list-of-lists representation of a nameless environment*) Build a list-of-lists (i.e., ribcage) representation of a *nameless* environment (Figure 9.8) with the following interface:

```
def empty_nameless_environment()
def extend_nameless_environment (values, environment)
def apply_nameless_environment (environment, depth, position)
```

In other words, complete Programming Exercise 9.8.4.d in Python using a list-of-lists representation (Figure 9.8), instead of a list-of-vectors representation.

In this representation of a nameless environment, the lexical address of a variable reference v is $(depth, position)$; it indicates where to find (and retrieve) the value bound to the identifier used in a reference (i.e., at rib $depth$ in position $position$). Thus, invoking the function `apply_nameless_environment` with the parameters `environment`, `depth`, and `position` retrieves the value at the $(depth, position)$ address in the environment.

Exercise 9.8.6 (*closure representation of a nameless environment in Scheme*) Complete Programming Exercise 9.8.4.d (a nameless environment), but this time use a closure representation, instead of a ribcage representation, for the environment. The closure representation of a named environment in Scheme is given in Section 9.8.2.

Exercise 9.8.7 (*closure representation of a nameless environment in Python*) Complete Programming Exercise 9.8.5.b (a nameless environment), but this time use a closure representation, instead of a ribcage representation, for the environment. The closure representation of a named environment in Python is given in Section 9.8.3.

Exercise 9.8.8 (*abstract-syntax representation of a nameless environment in Racket Scheme*) Complete Programming Exercise 9.8.4.d (a nameless environment), but this time use an abstract-syntax representation, instead of a ribcage representation, for the environment (Figure 9.9). The abstract-syntax representation of a named environment in Racket Scheme is developed in Programming Exercise 9.8.3.

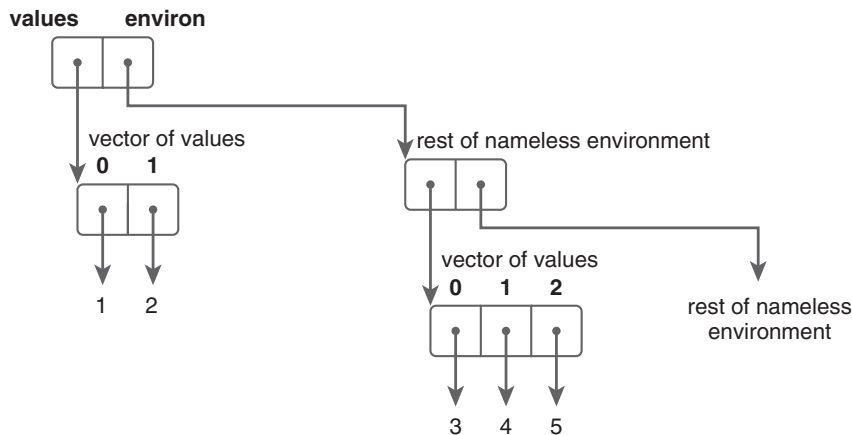


Figure 9.9 An abstract-syntax representation of a nameless environment in Racket Scheme using the structure of Programming Exercise 9.8.8.

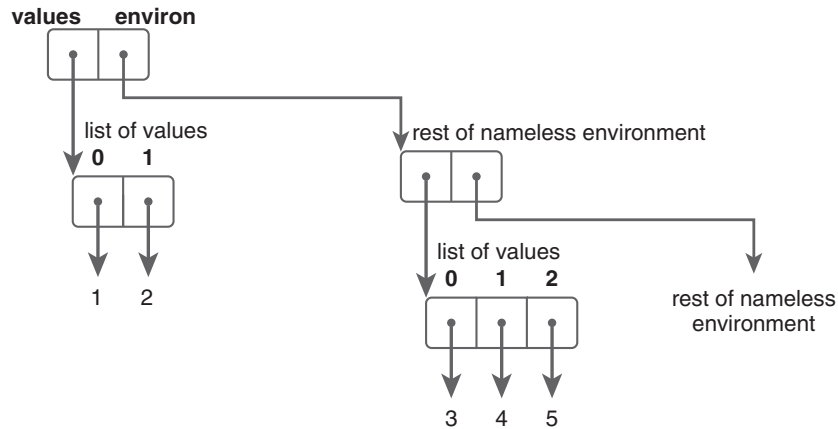


Figure 9.10 An abstract-syntax representation of a nameless environment in Python using the structure of Programming Exercise 9.8.9.

Exercise 9.8.9 (*abstract-syntax representation of a nameless environment in Python*) Complete Programming Exercise 9.8.5.b (a nameless environment), but this time use an abstract-syntax representation, instead of a ribcage representation, for the environment (Figure 9.10). The abstract-syntax representation of a named environment in Python is given in Section 9.8.4 and shown in Figure 9.3.

9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis

We are now ready to draw some comparisons between ML and Haskell.

9.9.1 ML Summary

ML is a statically scoped, programming language that supports primarily functional programming with a safe type system, type inference, an eager evaluation strategy, parametric polymorphism, algebraic data types, pattern matching, automatic memory management through garbage collection, a rich and expressive polymorphic type and module system, and some imperative features. ML integrates functional features from Lisp, rule-based programming (i.e., pattern matching) from Prolog, data abstraction from Smalltalk, and has a more readable syntax than Lisp. As a result, ML is a useful general-purpose programming language.

9.9.2 Haskell Summary

Haskell is a fully curried, statically scoped, (nearly) pure functional programming language with a lazy evaluation parameter-passing strategy, safe type system, type inference, parametric polymorphism, algebraic data types, pattern matching,

automatic memory management through garbage collection, and a rich and expressive polymorphic type and class system.

9.9.3 Comparison of ML and Haskell

Table 9.7 compares the main concepts and features of ML and Haskell. The primary difference between these two languages is that ML uses eager evaluation (i.e., call-by-value) while Haskell uses lazy evaluation (i.e., call-by-name). Eager evaluation means that all subexpressions are always evaluated. These parameter-passing evaluation strategies are discussed in Chapter 12. Unlike Haskell, not all built-in functions in ML are curried. However, the higher-order functions `map`, `foldl`, and `foldr`, which are useful in creating new functions, are curried in ML. ML and Haskell share a similar syntax, though the syntax in Haskell is terser than that in ML. The other differences mentioned in Table 9.7 are mostly syntactic. Haskell is also (nearly) purely functional, in that it has no imperative features or provisions for side effects, even for I/O. Haskell uses the mathematical notion of a monad for conducting I/O while remaining faithful to functional purity. The following expressions succinctly summarize ML and Haskell in relation to each other and to Lisp:

Haskell	=	ML	+	Lazy Evaluation	-	Side Effects
ML	=	Lisp	-	Homoiconicity	+	Safe Type System
Haskell	=	Lisp	-	Homoiconicity	+	Safe Type System
			-	Side Effects	+	Lazy Evaluation

9.9.4 Applications

The features of ML are ideally applied in language-processing systems, including compilers and theorem provers (Appel 2004). Haskell is also being increasingly used for application development in a commercial setting. Examples of applications developed in Haskell include a revision control system and a window manager for the X Window System. Galois is a software development and computer science research company that has used Haskell in multiple projects.⁶

ML and Haskell are also used for artificial intelligence (AI) applications. Traditionally, Prolog, which is presented in Chapter 14, has been recognized as a language for AI, particularly because it has a built-in theorem-proving algorithm called *resolution* and implements the associated techniques of *unification* and *backtracking*, which make resolution practical in a computer system. As a result, the semantics of Prolog are more complex than those of languages such as Scheme, C, and Java. A Prolog program consists of a set of facts and rules. An ML or Haskell program involving a series of function definitions using pattern-directed invocation has much the same appearance. (The built-in list data structures in Prolog and ML/Haskell are nearly identical.) Moreover, the *pattern-directed invocation* built into ML and Haskell is similar to the rule system in Prolog, albeit

6. <https://galois.com/about/haskell/>

Concept	ML	Haskell
lists	homogeneous	homogeneous
cons	<code>::</code>	<code>:</code>
append	<code>@</code>	<code>++</code>
integer equality	<code>=</code>	<code>==</code>
integer inequality	<code><></code>	<code>/=</code>
strings	not a list of characters use <code>explode</code>	a list of Characters
renaming parameters	<code>lst as (X : XS)</code>	<code>lst@ (X : XS)</code>
functional redefinition	permitted	not permitted
pattern-directed invocation	yes, with <code> </code>	yes
parameter passing	call-by-value, strict, applicative-order evaluation	call-by-need, non-strict, normal-order evaluation
functional composition	<code>o</code>	<code>.</code>
infix to prefix	<code>(op operator)</code>	<code>(operator)</code>
sections	not supported	supported, use <code>(operator)</code>
prefix to infix		<code>`operator`</code>
user-defined functions	introduced with <code>fun</code> can be defined at the prompt or in a script	must be defined in a script
anonymous functions	<code>(fn tuple => body)</code>	<code>(\ tuple -> body)</code>
curried form	omit parentheses, commas	omit parentheses, commas
curried	partially	fully
type declaration	<code>:</code>	<code>::</code>
type definition	<code>type</code>	<code>type</code>
data type definition	<code>datatype</code>	<code>data</code>
type variables	prefaced with <code>'</code> written before data type name	not prefaced with <code>'</code> written after data type name
function type	optional, but if used, embedded within function definition	optional, but if used, precedes function definition
type inference/checking	Hindley-Milner	Hindley-Milner
function overloading	not supported	supported through qualified types and type classes
ADTs	module system (structures, signatures, and functors)	class system

Table 9.7 Comparison of the Main Concepts and Features of ML and Haskell

without the semantic complexity associated with unification and backtracking in Prolog.

However, ML and Haskell, unlike Prolog, include currying and curried functions and a powerful type and module system for creating abstract data types. As a result, ML and Haskell are used for AI in applications where Prolog (or Lisp) might have been the only programming language considered in the past. Curry, nearly a superset of Haskell, is an experimental programming language that seeks to marry the functional and logic programming in a single language. Similarly, miniKanren is a family of languages for relational programming. ML (and Prolog) were developed in the early 1970s; Haskell was developed in the early 1990s.

9.9.5 Analysis

Some beginner programmers find the constraints of the safe type system in ML and Haskell to be a source of frustration. Moreover, some find type classes to be a source of frustration in Haskell. However, once these concepts are understood properly, advanced ML and Haskell programmers appreciate the safe, algebraic type systems in ML and Haskell.

The subtle syntax and sophisticated type system of Haskell are a double edged sword—highly appreciated by experienced programmers but also a source of frustration among beginners, since the generality of Haskell often leads to cryptic error messages. (Heeren, Leijen, and van IJzendoorn 2003, p. 62)

An understanding of the branch of mathematics known as *category theory* is helpful for mastering the safe, algebraic type systems in ML and Haskell. Paul Graham (n.d.) has written:

Most hackers I know have been disappointed by the ML family. Languages with static typing would be more suitable if programs were something you thought of in advance, and then merely translated into code. But that's not how programs get written. The inability to have lists of mixed types is a particularly crippling restriction. It gets in the way of exploratory programming (it's convenient early on to represent everything as lists), . . .

9.10 Thematic Takeaways

- A goal of a *type system* is to support *data abstraction* and, in particular, the definition of *abstract data types* that have the properties and behavior of primitive types.
- An inductive *variant record* data type—a union of records—is particularly useful for representing an *abstract-syntax tree* of a computer program.
- Data types and the functions that manipulate them are natural reflections of each other—a theme reinforced in Chapter 5. As a result, programming

languages support the construction (e.g., `define-datatype`) and decomposition (e.g., `cases`) of data types.

- The conception and use of an *abstract data type* data structure are distributed among an implementation-neutral *interface*, an *implementation* containing function *definitions*, and an *application* containing invocations to functions in the implementation.
- The underlying representation/implementation of an abstract data type can change without breaking the application code as long as the contractual signature of each function declaration in the interface remains unchanged. In this way, the implementation is *hidden* from the application.
- A variety of representation strategies for data structures are possible, including list, abstract syntax, and closure representations.
- Well-defined data structures as abstract data types are an essential ingredient in the implementation of a programming language (e.g., interpreters and compilers).
- A programming language with an expressive type system is indispensable for the construction of efficacious and efficient data structures.

9.11 Chapter Summary

Type systems support *data abstraction* and, in particular, the definition of user-defined data types that have the properties and behavior of primitive types. A variety of aggregate (e.g., arrays, records, and unions) and inductive data types (e.g., linked list) can be constructed using the type system of a language. A *type system* of a programming language includes the mechanism for creating new data types from existing types. It should enable the creation of new data types easily and flexibly. The pattern matching built into ML and Haskell supports the decomposition of an (inductive) aggregate data type.

Variant records (i.e., unions of records) and *abstract syntax* are of particular use in data structures for representing computer programs. An *abstract-syntax tree* (AST) is similar to a parse tree, except that it uses *abstract syntax* or an *internal representation* (i.e., it is internal to the system processing it) rather than *concrete syntax*. Specifically, while the structure of a parse tree depicts how a sentence (in concrete syntax) conforms to a grammar, the structure of an abstract-syntax tree illustrates how the sentence is represented internally, typically with an inductive, variant record data type.

Data abstraction involves factoring the conception and use of a data structure into an *interface*, *implementation*, and *application*. The implementation is hidden from the application, meaning that a variety of representations can be used for the data structure in the implementation without requiring changes to the application since both conform to the interface. A data structure created in this way is called an *abstract data type*. A goal of a type system is to support the definition of abstract data types that have the properties and behavior of primitive types. A variety of representation strategies for data structures are possible, including abstract-syntax

and closure representations. This chapter prepares us for designing efficacious and efficient data structures for the interpreters we build in Part III (Chapters 10–12).

9.12 Notes and Further Reading

The closure representation of an environment in Section 9.8.2 is from Friedman, Wand, and Haynes (2001); where it is referred to as a *procedural representation*, with minor modifications in presentation here. The concept of a *ribcage representation* of an environment is also articulated in Friedman, Wand, and Haynes (2001). We adopt the notation $[v]$ from Friedman, Wand, and Haynes (2001) to indicate “the representation of data v .” The original version of ML theoretically expressed by A. J. Robin Milner in 1978 (Milner 1978) used a slightly different syntax than Standard ML, used here, and did not support pattern matching and constructor algebras. For more information on the ML type system, we refer the reader to Ullman (1997, Chapter 6). For reflections on and a critique of Standard ML, see MacQueen (1993) and Appel (1993), respectively. Idris is a programming language for type-driven development with similar features to ML and Haskell. Type systems are being applied to the areas of networking and computer security (Wright 2010).

PART III

INTERPRETER

IMPLEMENTATION

Chapters 10–11 and Sections 12.2, 12.4, and 12.6–12.7 are inspired by Friedman, Wand, and Haynes (2001, Chapter 3). The primary difference between the two approaches is in implementation language. We use Python to build environment-passing interpreters while Friedman, Wand, and Haynes (2001) uses Scheme. Appendix A provides an introduction to the Python programming language.

We recommend that readers begin with online Appendix D, which is a guide to getting started with Camille and includes details of its syntax and semantics, how to acquire access to the Camille Git repository necessary for using Camille, and the pedagogical approach to using the language. Online Appendix E provides the individual grammars for the progressive versions of Camille in one central location.

Chapter 10

Local Binding and Conditional Evaluation

The interpreter for a computer language is just another program.

— Hal Abelson in Foreword to *Essentials of Programming Languages* (Friedman, Wand, and Haynes 2001)

Les yeux sont les interprètes du coeur, mais il n’y a que celui qui y a intérêt qui entend leur langage.

(Translation: The eyes are the interpreters of the heart, but only those who have an interest can hear their language.)

— Blaise Pascal

THIS book is about programming language concepts. One approach to learning language concepts is to implement them by building interpreters for computer languages. Interpreter implementation also provides the *operational semantics* for the interpreted programs. In this and the following two chapters we put into practice the language concepts we have encountered in Chapters 1–9.

10.1 Chapter Objectives

- Introduce the essentials of interpreter implementation.
- Explore the implementation of *local binding*.
- Explore the implementation of *conditional evaluation*.

10.2 Checkpoint

Thus far in this course of study of programming languages, we have explored:

- (Chapter 2) Language definition methods (i.e., *grammars*). We have also used these methods as a model to define data structures and implement functions that access them.
- (Chapter 5) Recursive, functional programming in λ -calculus and Scheme (and ML and Haskell in online Appendices B and C, respectively).
- (Chapter 6) *Binding* (as a general programming language concept) and (static and dynamic) *scoping*.
- (Chapter 8) *Partial function application*, *currying*, and *higher-order functions* as a way to create powerful and reusable programming abstractions.
- (Chapter 9) *Data types* and *type systems*:
 - definition (with `class` in Python; with `define-datatype` in Racket Scheme; with `type` and `datatype` in ML; and with `type` and `data` in Haskell)
 - *pattern matching* and *pattern-directed invocation* (with `cases` in Scheme, and built into ML and Haskell)
- (Chapter 9) *Data abstraction* and *abstract data types*:
 - the concepts of *interface*, *implementation*, and *application*
 - multiple representations (*list*, *abstract syntax*, and *closure*) for defining an implementation for organizing data structures in an interpreter, especially an environment

We now use these fundamentals to build (data-driven, environment-passing) interpreters, in the style of `occurs-free?` from Chapter 6, and `concrete2abstract` and `abstract2concrete` from Chapter 9 (Section 9.6 and Programming Exercise 9.6.2). We progressively add language concepts and features, including conditional evaluation, local binding, (recursive) functions, a variety of parameter-passing mechanisms, statements, and other concepts as we move through Chapters 10–12.

Camille is a programming language inspired by Friedman, Wand, and Haynes (2001), which is intended for learning the concepts and implementation of computer languages through the development of a series of interpreters for it written in Python (Perugini and Watkin 2018). In particular, in Chapters 10–12 we implement a variety of an environment-passing interpreters for Camille—in the tradition of Friedman, Wand, and Haynes (2001)—in Python.

There are multiple benefits of incrementally implementing language interpreters. First, we are confronted with one of the most fundamental truths of computing: “the interpreter for a computer language is just another program” (Friedman, Wand, and Haynes 2001, Foreword, p. vii, Hal Abelson). Second, once a language interpreter is established as just another program, we realize quickly that implementing a new concept, construct, or feature in a computer language involves adding code at particular points in that program. Third, we learn the causal relationship between a language and its interpreter. In other words, we realize that an interpreter for a language explicitly defines the semantics of the language that it interprets. The consequences of this realization

are compelling: We can be mystified by the drastic changes we can effect in the semantics of implemented language by changing only a few lines of code in the interpreter—sometimes as little as one line (e.g., using dynamic scoping rather than static scoping, or using lazy evaluation as opposed to eager evaluation).

We use Python as the implementation language in the construction of these interpreters. Thus, an understanding of Python is requisite for the construction of interpreters in Python in Chapters 10–12. We refer readers to Appendix A for an introduction to the Python programming language.

Online Appendix D is a guide to getting started with Camille and includes details of its syntax and semantics, how to acquire access to the Camille Git repository necessary for using Camille, and the pedagogical approach to using the language. The Camille Git repository is available at <https://bitbucket.org/camilleinterpreter/camille-interpreter-in-python-release/src/master/>. Its structure and contents are described in online Appendix D and at <https://bitbucket.org/camilleinterpreter/camille-interpreter-in-python-release/src/master/PAPER/paper.md>. Online Appendix E provides the individual grammars for the progressive versions of Camille in one central location.

10.3 Overview: Learning Language Concepts Through Interpreters

We start by implementing only primitive operations in this chapter. Then, we develop an `evaluate-expression` function that accepts an expression and an environment as arguments, evaluates the passed expression in the passed environment, and returns the result. This function, which is at the heart of any interpreter, constitutes a large conditional structure based on the type of expression passed (e.g., a variable reference or function definition).

Adding support for a new concept or feature to the language typically involves adding a new grammar rule (in `camilleparse.py`) and/or primitive (in `camillelib.py`), adding a new field to the abstract-syntax representation of an expression (in `camilleinterpreter.py`), and adding a new case to the `evaluate_expr` function (in `camilleinterpreter.py`).

Next, we add support for conditional evaluation and local binding. Support for local binding requires a lookup environment, which leads to the possibility of testing a variety of representations for that environment (as discussed in Chapter 9), as long as it adheres to the well-defined interface used by `evaluate_expr`. Later, in Chapter 11, we add support for non-recursive functions, which raises the issue of how to represent a function—there are a host of options from which to choose. At this point, we can also explore implementing dynamic scoping as an alternative to the default static scoping. This amounts to little more than storing the calling environment, rather than the lexically enclosing environment, in the representation of the function. Next, we implement recursive functions, also in Chapter 11, which require a modified environment. At this point, we will have implemented Camille v2.1, which only supports functional

programming, and explored the use of multiple configuration options for both aspects of the design of the interpreter as well as the semantics of implemented concepts (see Table 10.3 later in this chapter).

Next, we start slowly to morph Camille, in Chapter 12, through its interpreter, into a language with imperative programming features by adding provisions for side effect (e.g., through variable assignment). Variable assignment requires a modification to the representation of the environment. Now, the environment must store references to expressed values, rather than the expressed values themselves. This raises the issue of implicit versus explicit dereferencing, and naturally leads to exploring a variety of parameter-passing mechanisms, such as pass-by-reference or pass-by-name/lazy evaluation. Finally, in Chapter 12, we close the loop on the imperative approach by eliminating the need to use recursion for repetition by recalibrating the language, through its interpreter, to be a statement-oriented, rather than expression-oriented, language. This involves adding support for statement blocks, while loops, and I/O operations.

10.4 Preliminaries: Interpreter Essentials

Building an interpreter for a computer language involves defining the following elements:

1. **A Read-Eval-Print Loop (REPL):** a user interface that reads program strings and passes them to the front end of the interpreter
2. **A Front End:** a source code parser that translates a string representing a program into an abstract-syntax representation—usually a tree—of the program, sometimes referred to as bytecode
3. **An Interpreter:**¹ an expression evaluation function or loop that traverses and interprets an abstract-syntax representation of the program
4. **Supporting Data Types/Structures and Libraries:** a suite of abstract data types (e.g., an environment, closure, and reference) and associated functions to support the evaluation of expressions

We present each of the first three of these components in Section 10.6. We first encounter the need for supporting data types (in this case, an environment) and libraries in Section 10.7.

10.4.1 Expressed Values Vis-à-Vis Denoted Values

The set of values that a programming language manipulates fall into two categories:

1. The component of a language implementation that accepts an abstract-syntax tree and evaluates it is called an *interpreter*—see Chapter 4 and the rightmost component labeled “Interpreter” in Figure 10.1. However, we generally refer to the entire language implementation as *the interpreter*. To the programmer of the source program being interpreted, the entire language implementation (Figure 4.1) is the interpreter rather than just the last component of it.

- *Expressed values* are the possible (return) values of expressions (e.g., numbers, characters, and strings in Java or Scheme).
- *Denoted values* are values bound to variables (e.g., references to locations containing expressed values in Java or Scheme).

10.4.2 Defined Language Vis-à-Vis Defining Language

When building an interpreter, we think of two languages:

- The *defined programming language* (or *source language*) is the language specified (or operationalized) by the interpreter.
- The *defining programming language* (or *host language*) is the language in which we implement the interpreter (for the defined language).

Here, our defined language is Camille and our defining language is Python.

10.5 The Camille Grammar and Language

Here is our first Camille grammar:

```

<program> ::= <expression>

<expression> ::= ntNumber
               <number>

<expression> ::= ntPrimitive_op
               <primitive> ({<expression>}+(<.>))

<primitive> ::= ntPrimitive
               + | - | * | incl | decl | zero? | eqv?

```

At this point, the language only has support for numbers and primitive operations. Sample expressions in Camille are:

```

32
+ (33, 1)
incl (2)
decl (4)
decl (- (33, 1))
+ (incl (2), - (6, 4))
+ (- (35, 33), incl (8))

```

Currently, in Camille,

```

expressed value = integer
denoted value  = integer

```

Thus,

expressed value = denoted value = integer

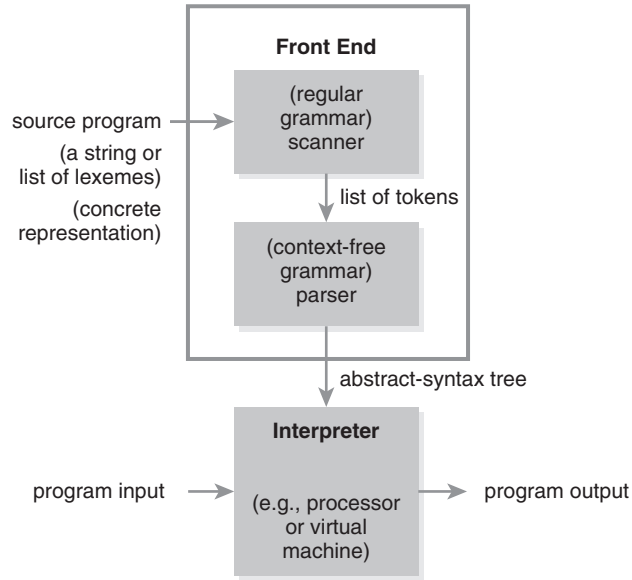


Figure 10.1 Execution by interpretation.

10.6 A First Camille Interpreter

10.6.1 Front End for Camille

Language processing starts with a program to convert Camille program text (i.e., a string) into an abstract-syntax tree. In other words, we need a scanner and a parser, referred to as a *front end* (shown on the left-hand side of Figure 10.1), which can accept a string, verify that it is a sentence in Camille, and translate it into an abstract-syntax representation. Recall from Chapter 3 that scanning culls out the lexemes, determines whether all are valid, and returns a list of tokens. Parsing determines whether the list of tokens is in the correct order and, if so, structures this list into an abstract-syntax tree. A parser generator is a program that accepts lexical and syntactic specifications and automatically generates a scanner and parser from them. We use the PLY (Python Lex-Yacc) parser generator for Python introduced in Chapter 3 (i.e., the Python analog for `lex` and `yacc` in C). The following code is a generator in PLY for the front end of Camille:

```

1 import re
2 import sys
3 import operator
4 import traceback
5 import ply.lex as lex
6 import ply.yacc as yacc
7 from collections import defaultdict
8
9 # begin lexical specification #

```

```

10 tokens = ('NUMBER', 'PLUS', 'WORD', 'MINUS', 'MULT', 'DEC1', 'INC1',
11           'ZERO', 'LPAREN', 'RPAREN', 'COMMA', 'EQV', 'COMMENT')
12
13 keywords = ('incl', 'decl', 'zero?', 'eqv?')
14
15 keyword_lookup = {'incl' : 'INC1', 'decl' : 'DEC1',
16                  'zero?' : 'ZERO', 'eqv?' : 'EQV' }
17
18 t_PLUS      = r'\+'
19 t_MINUS     = r'\-'
20 t_MULT      = r'\*'
21 t_LPAREN    = r'\('
22 t_RPAREN    = r'\)'
23 t_COMMA     = r','
24 t_ignore    = " \t"
25
26 def t_WORD(t):
27     r'[A-Za-z_][A-Za-z_0-9*?!]*'
28     pattern = re.compile("^[A-Za-z_][A-Za-z_0-9*?!]*$")
29
30     # if the identifier is a keyword, parse it as such
31     if t.value in keywords:
32         t.type = keyword_lookup[t.value]
33     # otherwise it is a syntax error
34     else:
35         print("Runtime error: Unknown word %s %d" %
36               (t.value[0], t.lexer.lineno))
37         sys.exit(-1)
38     return t
39
40 def t_NUMBER(t):
41     r'[-?\d+]'
42     # try to convert the string to an int, flag overflows
43     try:
44         t.value = int(t.value)
45     except ValueError:
46         print("Runtime error: number too large %s %d" %
47               (t.value[0], t.lexer.lineno))
48         sys.exit(-1)
49     return t
50
51 def t_COMMENT(t):
52     r'---.*'
53     pass
54
55 def t_newline(t):
56     r'\n'
57     t.lexer.lineno = t.lexer.lineno + 1
58
59 def t_error(t):
60     print("Unrecognized token %s on line %d." % (t.value.rstrip(),
61                                                  t.lexer.lineno))
62
63 lexer = lex.lex()
64 # end lexical specification #
65
66 # begin syntactic specification
67 class ParserException(Exception):
68     def __init__(self, message):
69         self.message = message
70
71 def p_error(t):
72     if (t != None):
73         raise ParserException("Syntax error: Line %d " % (t.lineno))

```

```

73     else:
74         raise ParserException("Syntax error near: Line %d" %
75                               (lexer.lineno - (lexer.lineno > 1)))
76
77 def p_program_expr(t):
78     '''programs : program programs
79                 | program'''
80     #do nothing
81
82 def p_line_expr(t):
83     '''program : expression'''
84     t[0] = t[1]
85     print(evaluate_expr(t[0]))
86
87 def p_primitive_op(t):
88     '''expression : primitive LPAREN expressions RPAREN'''
89     t[0] = Tree_Node(ntPrimitive_op, [t[3]], t[1], t.lineno(1))
90
91 def p_primitive(t):
92     '''primitive : PLUS
93                 | MINUS
94                 | INC1
95                 | MULT
96                 | DEC1
97                 | ZERO
98                 | EQV'''
99     t[0] = Tree_Node(ntPrimitive, None, t[1], t.lineno(1))
100
101 def p_expression_number(t):
102     '''expression : NUMBER'''
103     t[0] = Tree_Node(ntNumber, None, t[1], t.lineno(1))
104
105 def p_expressions(t):
106     '''expressions : expression
107                   | expression COMMA expressions'''
108     if len(t) == 4:
109         t[0] = Tree_Node(ntExpressions, [t[1], t[3]], None, t.lineno(1))
110     elif len(t) == 2:
111         t[0] = Tree_Node(ntExpressions, [t[1]], None, t.lineno(1))
112 # end syntactic specification
113
114 def parser_feed(s, parser):
115     pattern = re.compile("[^ \t]+")
116     if pattern.search(s):
117         try:
118             parser.parse(s)
119         except InterpreterException as e:
120             print("Line %s: %s" % (e.lineno, e.message))
121             if (e.additional_information != None):
122                 print("Additional information:")
123                 print(e.additional_information)
124         except ParserException as e:
125             print(e.message)
126         except Exception as e:
127             print("Unknown Error occurred "
128                   "(this is normally caused by a Python syntax error)")
129     raise e

```

Lines 9–63 and 65–112 constitute the lexical and syntactic specifications, respectively. Comments in Camille programs begin with the lexeme `---` (i.e., three consecutive dashes) and continue to the end of the line. Multi-line comments

are not supported. Comments are ignored by the scanner (lines 51–53). Recall from Chapter 3 that the `lex.lex()` (line 62) generates a scanner. Similarly, the function `yacc.yacc()` generates a parser and is called in the interpreter from the REPL definition (Section 10.6.4). Notice that the `p_line_expr` function (lines 82–85) has changed slightly from the version shown on lines 135–139 in the parser generator listing in Section 9.6.2. In particular, lines 138–139 in the original definition

```
135 def p_line_expr(t):
136     '''program : expression'''
137     t[0] = t[1]
138     global global_tree
139     global_tree = t[0]
```

are replaced with line 85 in the current definition:

```
82 def p_line_expr(t):
83     '''program : expression'''
84     t[0] = t[1]
85     print (evaluate_expr(t[0]))
```

Rather than assign the final abstract-syntax tree to the global variable `global_tree` (line 139) so that it can be referenced by a function that invokes the parser (e.g., the `concrete2abstract` function), now we pass the tree to the interpreter (i.e., the `evaluate_expr` function) on line 85.

For details on PLY, see <https://www.dabeaz.com/ply/>. The use of a scanner/parser generator facilitates this incremental development approach, which leads to a more malleable interpreter/language. Thus, the lexical and syntactic specifications given here can be used as is, and the scanner and parser generated from them can be considered black boxes.

10.6.2 Simple Interpreter for Camille

A simple interpreter for Camille follows:

```
130 # begin implementation of primitive operations
131 def eqv(op1, op2):
132     return op1 == op2
133
134 def decl1(op):
135     return op - 1
136
137 def incl(op):
138     return op + 1
139
140 def isZero(op):
141     return op == 0
142 # end implementation of primitive operations
143
144 # begin expression data type #
145
146 # list of node types
147 ntPrimitive = 'Primitive'
148 ntPrimitive_op = 'Primitive Operator'
```



```

149 ntNumber = 'Number'
150 ntExpressions = 'Expressions'
151
152 class Tree_Node:
153     def __init__(self, type, children, leaf, linenumber):
154         self.type = type
155         #save the line number of the node so run-time
156         #errors can be indicated
157         self.linenumber = linenumber
158         if children:
159             self.children = children
160         else:
161             self.children = [ ]
162         self.leaf = leaf
163 # end expression data type #
164
165 # begin interpreter #
166 class InterpreterException(Exception):
167     def __init__(self, linenumber, message,
168                 additional_information=None, exception=None):
169         self.linenumber = linenumber
170         self.message = message
171         self.additional_information = additional_information
172         self.exception = exception
173
174 primitive_op_dict = { "+" : operator.add, "-" : operator.sub,
175                      "*" : operator.mul, "decl" : decl1,
176                      "incl" : incl1, "zero?" : isZero,
177                      "eqv?" : eqv }
178 primitive_op_dict = defaultdict(lambda: -1, primitive_op_dict)
179
180 def evaluate_operands(operands):
181     return map(lambda x : evaluate_operand(x), operands)
182
183 def evaluate_operand(operand):
184     return evaluate_expr(operand)
185
186 def apply_primitive(prim, arguments):
187     return primitive_op_dict[prim.leaf](*arguments)
188
189 def printtree(expr):
190     print(expr.leaf)
191     for child in expr.children:
192         printtree(child)
193
194 def evaluate_expr(expr):
195     try:
196         if expr.type == ntPrimitive_op:
197             # expr leaf is mapped during parsing to
198             # the appropriate binary operator function
199             arguments = list(evaluate_operands(expr.children))[0]
200             return apply_primitive(expr.leaf, arguments)
201
202         elif expr.type == ntNumber:
203             return expr.leaf
204
205         elif expr.type == ntExpressions:
206             ExprList = []
207             ExprList.append(evaluate_expr(expr.children[0]))
208
209             if len(expr.children) > 1:
210                 ExprList.extend(evaluate_expr(expr.children[1]))
211             return ExprList

```

```

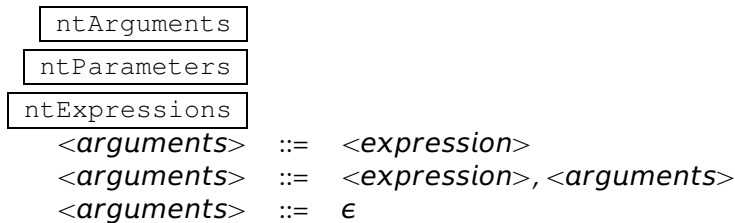
212     else:
213         raise InterpreterException(expr.linenumber,
214                                   "Invalid tree node type %s" % expr.type)
215     except InterpreterException as e:
216         # Raise exception to the next level until
217         # we reach the top level of the interpreter.
218         # Exceptions are fatal for a single tree,
219         # but other programs within a single file may
220         # otherwise be OK.
221         raise e
222     except Exception as e:
223         # We want to catch the Python interpreter exception and
224         # format it such that it can be used
225         # to debug the Camille program.
226         print(traceback.format_exc())
227         raise InterpreterException(expr.linenumber,
228                                   "Unhandled error in %s" % expr.type, str(e), e)
229 # end interpreter #

```

This segment of code contains both the definitions of the abstract-syntax tree data structure (lines 144–163) and the `evaluate_expr` function (lines 194–228). Notice that for each variant (lines 147–150) of the `TreeNode` data type (lines 152–162) that represents a Camille expression, there is a corresponding action in the `evaluate_expr` function (lines 194–228). Each variant in the `TreeNode` variant record² has a case in the `evaluate_expr` function. This interpreter is the component on the right-hand side of Figure 4.1, replicated here as Figure 10.1.

10.6.3 Abstract-Syntax Trees for Arguments Lists

We briefly discuss how the arguments to a primitive operator are represented in the abstract-syntax tree and evaluated. The following rules are used to represent the list of arguments to a primitive operator (or a function, which we encounter in Chapter 11):



Since all primitive operators in Camille accept arguments, the rule *<arguments> ::= ε* applies to (forthcoming) user-defined functions that may or may not accept arguments (as discussed in Chapter 11).

Consider the expression `*(7, x)` and its abstract-syntax tree presented in Figure 10.2. The top half of each node represents the `type` field of the `TreeNode`, the bottom right quarter of each node represents one member of the `children`

2. Technically, it is not a variant record as strictly defined, but rather a data type with fixed fields, where one of the fields, the `type` flag, indicates the interpretation of the fields.

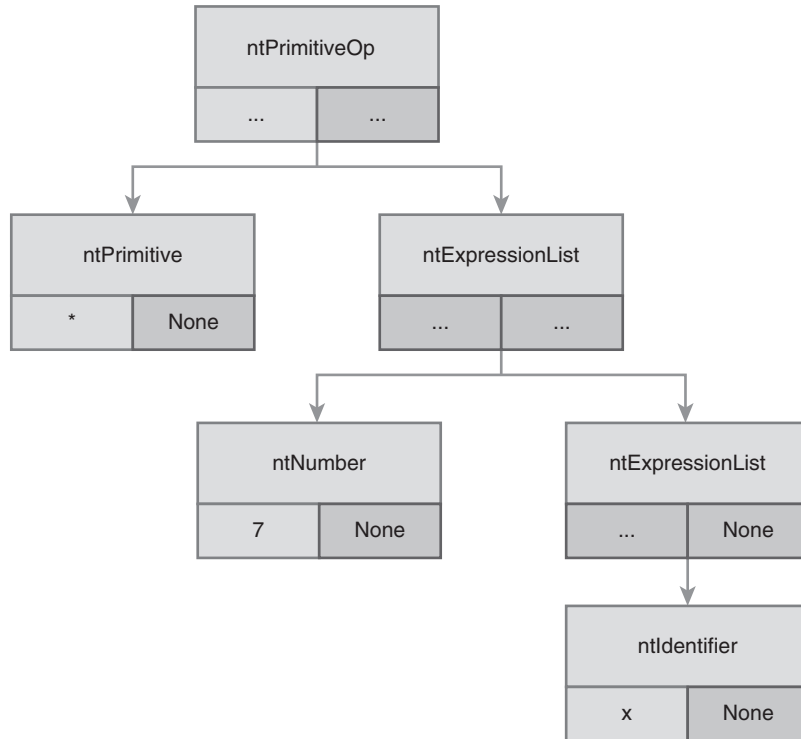


Figure 10.2 Abstract-syntax tree for the Camille expression $*(7, x)$.

list, and bottom left quarter of each node represents the `leaf` field. The `ntExpressionList` variant of `TreeNode` represents an argument list.

The `ntExpressionList` variant of an abstract-syntax tree constructed by the parser is flattened into a Python list by the interpreter for subsequent processing. A post-order traversal of the `ntExpressionList` variant is conducted, with the values in the leaf nodes being inserted into a Python list in the order in which they appear in the application of the primitive operator in the Camille source code. Each leaf is evaluated using `evaluate_expr` and its value is inserted into the Python list. Lines 205–211 of the `evaluate_expr` function (replicated here) demonstrate this process:

```

205 elif expr.type == ntExpressions:
206     ExprList = []
207     ExprList.append(evaluate_expr(expr.children[0]))
208
209     if len(expr.children) > 1:
210         ExprList.extend(evaluate_expr(expr.children[1]))
211     return ExprList

```

If a child exists, it becomes the next `ntExpressionList` node to be (recursively) traversed (line 210). This flattening process continues until a `ntExpressionList` node without a child is reached. The list returned by the recursive call to

`evaluate_expr` is appended to the list created with the leaf of the node (line 210).

10.6.4 REPL: Read-Eval-Print Loop

To make this interpreter operable (i.e., to test it), we need an interface for entering Camille expressions and running programs. The following is a *read-eval-print loop* (REPL) interface to the Camille interpreter:

```

230 # begin REPL
231 def main_func():
232     parser = yacc.yacc()
233     interactiveMode = False
234
235     if len(sys.argv) == 1:
236         interactiveMode = True
237
238     if interactiveMode:
239         program = ""
240         try:
241             prompt = 'Camille> '
242             while True:
243                 line = input(prompt)
244                 if (line == "" and program != ""):
245                     parser_feed(program, parser)
246                     lexer.lineno = 1
247                     program = ""
248                     prompt = 'Camille> '
249                 else:
250                     if (line != ""):
251                         program += (line + '\n')
252                         prompt = ''
253
254             except EOFError as e:
255                 sys.exit(0)
256
257             except Exception as e:
258                 print(e)
259                 sys.exit(-1)
260         else:
261             try:
262                 with open(sys.argv[1], 'r') as script:
263                     file_string = script.read()
264                     parser_feed(file_string, parser)
265                     sys.exit(0)
266             except Exception as e:
267                 print(e)
268                 sys.exit(-1)
269
270     main_func()
271 # end REPL

```

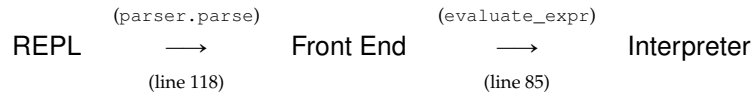
The function `yacc.yacc()` invoked on line 232 generates a parser and returns an object (here, named `parser`) that contains a function (named `parse`). This function accepts a string (representing a Camille program) and parses it (line 118 in the parser generator listing).

This REPL supports two ways of running Camille programs: *interactively* and *non-interactively*. In interactive mode (lines 238–259), the function `main_func`

prints the prompt, reads a string from standard input (line 243), and passes that string to the parser (line 245). In non-interactive mode (lines 261–268), the prompt for input is not printed. Instead, the REPL receives one or more Camille programs in a single source code file passed as a command-line argument (line 262), reads it as a string (line 263), and passes that string to the parser (line 264).

10.6.5 Connecting the Components

The following diagram depicts how the components of the interpreter are connected.



The REPL reads a string and passes it to the front end (`parser.parse`; line 118). The front end parses that string, while concomitantly building an abstract-syntax representation/tree for it, and passes that tree to the interpreter (`evaluate_expr`—the entry point of the interpreter; line 85). The interpreter traverses the tree to evaluate the program that the tree represents. Notice that this diagram is an instantiated view of Figure 10.1 with respect to the components of the Camille interpreter presented here.

10.6.6 How to Run a Camille Program

A bash script named `run` is available for use with each version of the Camille interpreter:

```
#!/usr/bin/env bash
python3.8 camilleinterpreter.py $1
```

Interactive mode is invoked by executing `run` without any command-line argument. The following is an interactive session with the Camille interpreter:

```
$ ./run
Camille> 32
32
Camille> +(33,1)
34
Camille> incl(2)
3
Camille> decl(4)
3
Camille> decl(-(33,1))
31
Camille> +(incl(2),-(6,4))
5
Camille> +(-(35,33),incl(7))
10
```

Non-interactive mode is invoked by passing the `run` script a single source code filename representing one or more Camille programs:

```
$ cat tests.cam
32
--- add a comment
+(33,1)
incl(2)
decl(4)
decl(-(33,1))
+(incl(2),-(6,4))
+(-(35,33),incl(7))
$ ./run tests.cam
32
34
3
3
31
5
10
```

In both interactive and non-interactive modes, Camille programs must be separated by a blank line—which explains the blank lines after each input expression in these transcripts from the Camille interpreter. We use this blank line after each program to support both the evaluation of multi-line programs at the REPL (in interactive mode) and the evaluation of multiple programs in a single source code file (in non-interactive mode).

10.7 Local Binding

To support local binding, we require syntactic and operational support for identifiers. Syntactically, to support local binding of values to identifiers in Camille, we add the following rules to the grammar:

	ntIdentifier
<code><expression></code>	<code>::= <identifier></code>
<code><expression></code>	<code>::= <let_expression></code>
	ntLet
<code><let_expression></code>	<code>::= let <let_statement> in <expression></code>
	ntLetStatement
<code><let_statement></code>	<code>::= <let_assignment></code>
<code><let_statement></code>	<code>::= <let_assignment> <let_statement></code>
	ntLetAssignment
<code><let_assignment></code>	<code>::= <identifier> = <expression></code>

We must also add the `let` and `in` keywords to the generator of the scanner on lines 10–16 at the beginning of Section 10.6.1. The following are the corresponding pattern-action rules in the `PLY` parser generator:

```

def p_expression_identifier(t):
    '''expression : IDENTIFIER'''
    t[0] = Tree_Node(ntIdentifier, None, t[1], t.lineno(1))

def p_expression_let(t):
    '''expression : LET let_statement IN expression'''
    t[0] = Tree_Node(ntLet, [t[2], t[4]], None, t.lineno(1))

def p_let_statement(t):
    '''let_statement : let_assignment | let_assignment let_statement'''
    if len(t) == 3:
        t[0] = Tree_Node(ntLetStatement, [t[1], t[2]], None, t.lineno(1))
    else:
        t[0] = Tree_Node(ntLetStatement, [t[1]], None, t.lineno(1))

def p_let_assignment(t):
    '''let_assignment : IDENTIFIER EQ expression'''
    t[0] = Tree_Node(ntLetAssignment, [t[3]], t[1], t.lineno(1))

```

We also must augment the `t_WORD` function in the lexical analyzer generator so that it can recognize locally bound identifiers:

```

1  def t_WORD(t):
2      r'[A-Za-z_][A-Za-z_0-9*?!]*'
3      pattern = re.compile("^[A-Za-z_][A-Za-z_0-9*?!]*$")
4
5      # if the identifier is a keyword, parse it as such
6      if t.value in keywords:
7          t.type = keyword_lookup[t.value]
8      # otherwise it might be a variable so check that
9      elif pattern.match(t.value):
10         t.type = 'IDENTIFIER'
11     # otherwise it is a syntax error
12     else:
13         print("Runtime error: Unknown word %s %d" %
14               (t.value[0], t.lexer.lineno))
15         sys.exit(-1)
16     return t

```

Lines 8–10 are the new lines of code inserted into the middle (between lines 32 and 33) of the original definition of the `t_WORD` function defined on lines 26–38 at the beginning of Section 10.6.1.

To bind values to identifiers, we require a data structure in which to store the values so that they can be retrieved using the identifier—in other words, we need an environment. The following is the closure representation of an environment in Python from Section 9.8 (repeated here for convenience):

```

# begin closure representation of environment #
def empty_environment():
    def raise_IE():
        raise IndexError
    return lambda symbol: raise_IE()

def apply_environment(environment, symbol):
    return environment(symbol)

def extend_environment(symbols, values, environment):
    def tryexcept(symbol):

```

```

    try:
        val = values[symbols.index(symbol)]
    except:
        val = apply_environment(environment, symbol)
    return val
return lambda symbol: tryexcept(symbol)
# end closure representation of environment #

simple_env = extend_environment(["a","b"], [1,2],
    extend_environment(["b","c","d"], [3,4,5],
        empty_environment()))

>>> print (apply_environment(simple_env, "d"))
5
>>> print (apply_environment(simple_env, "b"))
2
>>> print (apply_environment(simple_env, "e"))
No binding for symbol e.

```

Now that we have an environment, we need to modify the signatures of `evaluate_expr`, `evaluate_operands`, and `evaluate_operand` so that they can accept an environment `environ` as an argument:

```

1  # begin interpreter #
2  def evaluate_operands(operands, environ):
3      return map(lambda x : evaluate_operand(x, environ), operands)
4
5  def evaluate_operand(operand, environ):
6      return evaluate_expr(operand, environ)
7
8  def apply_primitive(prim, arguments):
9      return primitive_op_dict[prim.leaf](*arguments)
10
11 def printtree(expr):
12     print(expr.leaf)
13     for child in expr.children:
14         printtree(child)
15
16 def evaluate_expr(expr, environ):
17     if expr.type == ntPrimitive_op:
18         # expr leaf is mapped during parsing to
19         # the appropriate binary operator function
20         arguments = list(evaluate_operands(expr.children, environ))[0]
21         return apply_primitive(expr.leaf, arguments)
22
23     elif expr.type == ntNumber:
24         return expr.leaf
25
26     elif expr.type == ntIdentifier:
27         try:
28             return apply_environment(environ, expr.leaf)
29         except:
30             raise InterpreterException(expr.linenum,
31                 "Unbound identifier '%s'" % expr.leaf)
32
33     elif expr.type == ntLet:
34
35         temp = evaluate_expr(expr.children[0], environ) # assignment
36         identifiers = []
37         arguments = []
38         for name in temp:
39             identifiers.append(name)

```



```

40         arguments.append(temp[name])
41
42     temp = evaluate_expr(expr.children[1],
43                          extend_environment(identifiers, arguments, environ))
44     return temp
45
46 elif expr.type == ntLetStatement:
47     # perform assignment
48     temp = evaluate_expr(expr.children[0], environ)
49     # perform subsequent assignment(s) if there are any (recursive)
50     if len(expr.children) > 1:
51         temp.update(evaluate_expr(expr.children[1], environ))
52     return temp
53
54 elif expr.type == ntLetAssignment:
55     return { expr.leaf : evaluate_expr(expr.children[0], environ) }
56
57 elif expr.type == ntExpressions:
58     ExprList = []
59     ExprList.append(evaluate_expr(expr.children[0], environ))
60
61     if len(expr.children) > 1:
62         ExprList.extend(evaluate_expr(expr.children[1], environ))
63     return ExprList
64 else:
65     raise InterpreterException(expr.linenumber,
66                                "Invalid tree node type %s" % expr.type)
67 # end interpreter #

```

Lines 33–44 of the `evaluate_expr` function access the `ntLet` variant of the abstract-syntax tree of type `TreeNode` and evaluate the `let` expression it represents. In particular, line 35 evaluates the right-hand side of the `=` sign in each binding, and lines 42–43 evaluate the body of the `let` expression (line 42) in an environment extended with the newly created bindings (line 43). Notice that we build support for local binding in Camille from first principles—specifically, by defining an environment.

We briefly discuss how the bindings in a `let` expression are both represented in the abstract-syntax tree and evaluated. The abstract-syntax tree that describes a `let` expression is similar to the abstract-syntax tree that describes an argument list.³ Figure 10.3 presents a simplified version of an abstract-syntax tree that represents a `let` expression. Again, the top half of each node represents the `type` field of the `TreeNode`, the bottom right quarter of each node represents one member of the `children` list, and bottom left quarter of each node represents the `leaf` field.⁴

Consider the `ntLet`, `ntLetStatement`, and `ntLetAssignment` cases in the `evaluate_expr` function:

```

33 elif expr.type == ntLet:
34
35     temp = evaluate_expr(expr.children[0], environ) # assignment
36     identifiers = []
37     arguments = []
38     for name in temp:

```

3. The same approach is used in the abstract-syntax tree for `let*` (Programming Exercise 10.6) and `letrec` expressions (Section 11.3).

4. This figure is also applicable for `let*` and `letrec` expressions.

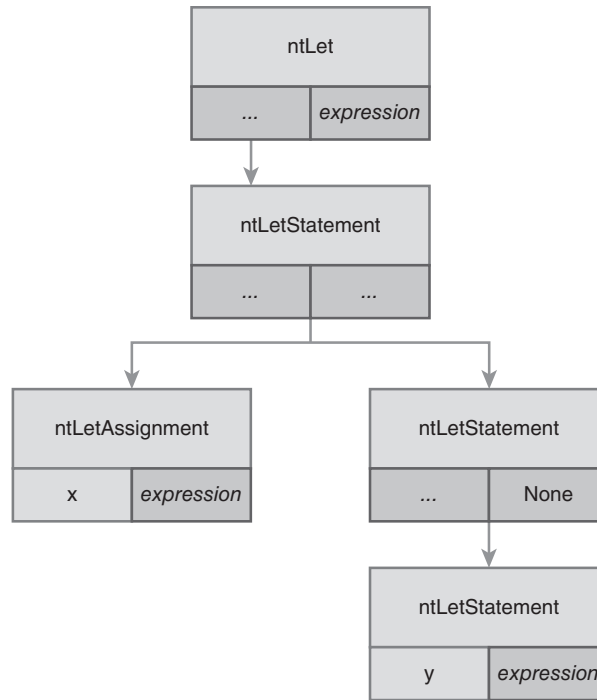


Figure 10.3 Abstract-syntax tree for the Camille expression

`let x = 1 y = 2 in *(x,y).`

```

39     identifiers.append(name)
40     arguments.append(temp[name])
41
42     temp = evaluate_expr(expr.children[1],
43                          extend_environment(identifiers, arguments, environ))
44     return temp
45
46 elif (expr.type == ntLetStatement):
47     # perform assignment
48     temp = evaluate_expr(expr.children[0], environ)
49     # perform subsequent assignment(s) if there are any (recursive)
50     if len(expr.children) > 1:
51         temp.update(evaluate_expr(expr.children[1], environ))
52     return temp
53
54 elif expr.type == ntLetAssignment:
55     return { expr.leaf : evaluate_expr(expr.children[0], environ) }

```

A subtree for the `ntLetStatement` variant of an abstract-syntax tree for a `let` expression is traversed in the same fashion as a parameter/argument list is traversed—in a post-order fashion (lines 46–52). The `ntLet` (lines 33–44) and `ntLetAssignment` (lines 54–55) cases of `evaluate_expr` require discussion. The `ntLetAssignment` case (lines 54–55) creates a single-element Python dictionary (line 55) containing a name–value pair defined within the `let` expression. Once all `ntLetStatement` nodes are processed, a Python

dictionary containing all name–value pairs is returned to the `ntLet` case. The Python dictionary is then split into two lists: one containing only names (line 39) and another containing only values (line 40). These values are placed into an environment (line 43). The body of the `let` expression is then evaluated using this new environment (line 42).

It is also important to note that the last line of the `p_line_expr` function in the parser generator, `print(evaluate_expr(t[0]))` (line 85 of the listing at the beginning of Section 10.6.1), needs to be replaced with `print(evaluate_expr(t[0], empty_environment()))` so that an empty environment is passed to the `evaluate_expr` function with the AST of the program.

Example expressions in this version of Camille⁵ with their evaluated results follow:

```
Camille> let
           a=32
           b=33
       in
           -(b,a)
1
Camille> --- demonstrates a scope hole
let
    a=32
in
    let
        --- shadows the a on line 9
        a = -(a,16)
    in
        decl1(a)
15
Camille> let a = 9 in i
Line 1: Unbound identifier 'i'
```

10.8 Conditional Evaluation

To support conditional evaluation in Camille, we add the following rules to the grammar and corresponding pattern-action rules to the PLY parser generator:

`<expression>` ::= `<conditional_expression>`

`<conditional_expression>` ::= `ntIfElse` `if <expression> <expression> else <expression>`

```
def p_expression_condition(t):
    '''expression : IF expression ELSE expression'''
    t[0] = Tree_Node(ntIfElse, [t[2], t[3], t[5]], None, t.lineno(1))
```

We must also add the `if` and `else` keywords to the generator of the scanner on lines 10–16 of the listing at the beginning of Section 10.6.1.

5. Camille version 1.1(named CLS).

The following code segment of the `evaluate_expr` function accesses the `ntIfElse` variant of the abstract-syntax tree of type `TreeNode` and evaluates the conditional expression it represents:

```

1  def evaluate_expr(expr, environ):
2      try:
3          if expr.type == ntPrimitive_op:
4              ...
5              ...
6              ...
7          elif expr.type == ntIfElse:
8              if evaluate_expr(expr.children[0], environ):
9                  return evaluate_expr(expr.children[1], environ)
10             else:
11                 return evaluate_expr(expr.children[2], environ)

```

Notice that we implement conditional evaluation in Camille using the support for conditional evaluation in Python (i.e., `if ... else`; lines 7–10). In addition, we avoid adding a boolean type (for now) by associating 0 with false and anything else with true (as in the C programming language). Example expressions in this version of Camille with their evaluated results follow:

```

Camille> if incl(0) 32 else 33
32
Camille> if decl(-(33,32)) 32 else 33
33

```

10.9 Putting It All Together

The following interpreter for Camille supports both local binding and conditional evaluation:⁶

```

1  import re
2  import sys
3  import operator
4  import traceback
5  import ply.lex as lex
6  import ply.yacc as yacc
7  from collections import defaultdict
8
9  # begin closure representation of environment #
10 def empty_environment():
11     def raise_IE():
12         raise IndexError
13     return lambda symbol: raise_IE()
14
15 def apply_environment(environment, symbol):
16     return environment(symbol)
17
18 def extend_environment(symbols, values, environment):
19     def tryexcept(symbol):
20         try:
21             val = values[symbols.index(symbol)]

```

6. Camille version 1.2(named CLS).


```

85         "eqv?" : eqv }
86
87 primitive_op_dict = defaultdict(lambda: -1, primitive_op_dict)
88
89 def evaluate_operands(operands, environ):
90     return map(lambda x : evaluate_operand(x, environ), operands)
91
92 def evaluate_operand(operand, environ):
93     return evaluate_expr(operand, environ)
94
95 def apply_primitive(prim, arguments):
96     return primitive_op_dict[prim.leaf](*arguments)
97
98 def printtree(expr):
99     print(expr.leaf)
100    for child in expr.children:
101        printtree(child)
102
103 def evaluate_expr(expr, environ):
104     try:
105         if expr.type == ntPrimitive_op:
106             # expr leaf is mapped during parsing to
107             # the appropriate binary operator function
108             arguments = list(evaluate_operands(expr.children,
109                                                 environ))[0]
110             return apply_primitive(expr.leaf, arguments)
111
112         elif expr.type == ntNumber:
113             return expr.leaf
114
115         elif expr.type == ntIdentifier:
116             try:
117                 return apply_environment(environ, expr.leaf)
118             except:
119                 raise InterpreterException(expr.linenumber,
120                                             "Unbound identifier '%s'" % expr.leaf)
121
122         elif expr.type == ntIfElse:
123             if evaluate_expr(expr.children[0], environ):
124                 return evaluate_expr(expr.children[1], environ)
125             else:
126                 return evaluate_expr(expr.children[2], environ)
127
128         elif expr.type == ntLet:
129
130             # assignment
131             temp = evaluate_expr(expr.children[0], environ)
132             identifiers = []
133             arguments = []
134             for name in temp:
135                 identifiers.append(name)
136                 arguments.append(temp[name])
137
138             # evaluation
139             temp = evaluate_expr(expr.children[1],
140                                 extend_environment(identifiers, arguments,
141                                                    environ))
142             return temp
143
144         elif (expr.type == ntLetStatement):
145             # perform assignment
146             temp = evaluate_expr(expr.children[0], environ)
147             # perform subsequent assignment(s)

```

```

148         # if there are any (recursive)
149         if len(expr.children) > 1:
150             temp.update(evaluate_expr(expr.children[1], environ))
151         return temp
152
153     elif expr.type == ntLetAssignment:
154         return { expr.leaf : evaluate_expr(expr.children[0],
155                                             environ) }
156
157     elif expr.type == ntExpressions:
158         ExprList = []
159         ExprList.append(evaluate_expr(expr.children[0], environ))
160
161         if len(expr.children) > 1:
162             ExprList.extend(evaluate_expr(expr.children[1],
163                                           environ))
164         return ExprList
165     else:
166         raise InterpreterException(expr.linenumber,
167                                   "Invalid tree node type %s" % expr.type)
168 except InterpreterException as e:
169     # Raise exception to the next level until
170     # we reach the top level of the interpreter.
171     # Exceptions are fatal for a single tree,
172     # but other programs within a single file may
173     # otherwise be OK.
174     raise e
175 except Exception as e:
176     # we want to catch the Python interpreter exception and
177     # format it such that it can be used
178     # to debug the Camille program
179     print(traceback.format_exc())
180     raise InterpreterException(expr.linenumber,
181                               "Unhandled error in %s" % expr.type, str(e), e)
182 # end interpreter #
183
184 # begin lexical specification #
185
186 tokens = ('NUMBER', 'PLUS', 'WORD', 'MINUS', 'MULT', 'DECL',
187          'INCL', 'ZERO', 'LPAREN', 'RPAREN', 'COMMA',
188          'IDENTIFIER', 'LET', 'EQ',
189          'IN', 'IF', 'ELSE', 'EQV', 'COMMENT')
190
191 keywords = ('if', 'else', 'incl', 'decl',
192            'in', 'let', 'zero?', 'eqv?')
193
194 keyword_lookup = {'if' : 'IF', 'else' : 'ELSE',
195                  'incl' : 'INCL', 'decl' : 'DECL', 'in' : 'IN',
196                  'let' : 'LET', 'zero?' : 'ZERO',
197                  'eqv?' : 'EQV' }
198
199 t_PLUS      = r'\+'
200 t_MINUS     = r'\-'
201 t_MULT      = r'\*'
202 t_LPAREN    = r'\('
203 t_RPAREN    = r'\)'
204 t_COMMA     = r','
205 t_EQ        = r'='
206 t_ignore    = " \t"
207
208 def t_WORD(t):
209     r'[A-Za-z_][A-Za-z_0-9*?!]*'
210     pattern = re.compile("^([A-Za-z_][A-Za-z_0-9*?!]*$)")

```

```

211
212     # if the identifier is a keyword, parse it as such
213     if t.value in keywords:
214         t.type = keyword_lookup[t.value]
215     # otherwise it might be a variable so check that
216     elif pattern.match(t.value):
217         t.type = 'IDENTIFIER'
218     # otherwise it is a syntax error
219     else:
220         print("Runtime error: Unknown word %s %d" %
221               (t.value[0], t.lexer.lineno))
222         sys.exit(-1)
223     return t
224
225 def t_NUMBER(t):
226     r'[-?\d+]'
227     # try to convert the string to an int, flag overflows
228     try:
229         t.value = int(t.value)
230     except ValueError:
231         print("Runtime error: number too large %s %d" %
232               (t.value[0], t.lexer.lineno))
233         sys.exit(-1)
234     return t
235
236 def t_COMMENT(t):
237     r'---.*'
238     pass
239
240 def t_newline(t):
241     r'\n'
242     #continue to next line
243     t.lexer.lineno = t.lexer.lineno + 1
244
245 def t_error(t):
246     print("Unrecognized token %s on line %d." % (t.value.rstrip(),
247                                                  t.lexer.lineno))
248
249 lexer = lex.lex()
250 # end lexical specification #
251 # begin syntactic specification #
252 class ParserException(Exception):
253     def __init__(self, message):
254         self.message = message
255
256 def p_error(t):
257     if (t != None):
258         raise ParserException("Syntax error: Line %d " % (t.lineno))
259     else:
260         raise ParserException("Syntax error near: Line %d" %
261                               (lexer.lineno - (lexer.lineno > 1)))
262
263 def p_program_expr(t):
264     '''programs : program programs
265                | program'''
266     # do nothing
267
268 def p_line_expr(t):
269     '''program : expression'''
270     t[0] = t[1]
271     print(evaluate_expr(t[0], empty_environment()))
272
273 def p_primitive_op(t):

```



```

274     '''expression : primitive LPAREN expressions RPAREN'''
275     t[0] = Tree_Node(ntPrimitive_op, [t[3]], t[1], t.lexer.lineno)
276
277     def p_primitive(t):
278         '''primitive : PLUS
279                     | MINUS
280                     | INC1
281                     | MULT
282                     | DEC1
283                     | ZERO
284                     | EQV'''
285         t[0] = Tree_Node(ntPrimitive, None, t[1], t.lexer.lineno)
286
287     def p_expression_number(t):
288         '''expression : NUMBER'''
289         t[0] = Tree_Node(ntNumber, None, t[1], t.lexer.lineno)
290
291     def p_expression_identifier(t):
292         '''expression : IDENTIFIER'''
293         t[0] = Tree_Node( ntIdentifier, None, t[1], t.lexer.lineno)
294
295     def p_expression_let(t):
296         '''expression : LET let_statement IN expression'''
297         t[0] = Tree_Node(ntLet, [t[2], t[4]], None, t.lexer.lineno)
298
299     def p_expression_condition(t):
300         '''expression : IF expression expression ELSE expression'''
301         t[0] = Tree_Node(ntIfElse, [t[2], t[3], t[5]], None,
302                               t.lexer.lineno)
303
304     def p_expressions(t):
305         '''expressions : expression
306                     | expression COMMA expressions'''
307         if len(t) == 4:
308             t[0] = Tree_Node(ntExpressions, [t[1], t[3]], None,
309                               t.lexer.lineno)
310         elif len(t) == 2:
311             t[0] = Tree_Node(ntExpressions, [t[1]], None, t.lexer.lineno)
312
313     def p_let_statement(t):
314         '''let_statement : let_assignment
315                     | let_assignment let_statement'''
316         if len(t) == 3:
317             t[0] = Tree_Node(ntLetStatement, [t[1], t[2]], None,
318                               t.lexer.lineno)
319         else:
320             t[0] = Tree_Node(ntLetStatement, [t[1]], None, t.lexer.lineno)
321
322     def p_let_assignment(t):
323         '''let_assignment : IDENTIFIER EQ expression'''
324         t[0] = Tree_Node(ntLetAssignment, [t[3]], t[1], t.lexer.lineno)
325 # end syntactic specification #
326
327     def parser_feed(s,parser):
328         pattern = re.compile("[^ \t]+")
329         if pattern.search(s):
330             try:
331                 parser.parse(s)
332             except InterpreterException as e:
333                 print ( "Line %s: %s" % (e.linenummer, e.message))
334                 if ( e.additional_information != None ):
335                     print("Additional information:")
336                     print(e.additional_information)

```

```

337     except ParseException as e:
338         print(e.message)
339     except Exception as e:
340         print ("Unknown Error occurred "
341               "(This is normally caused by "
342               "a Python syntax error.)")
343         raise e
344
345 # begin REPL
346 def main_func():
347     parser = yacc.yacc()
348     interactiveMode = False
349
350     if len(sys.argv) == 1:
351         interactiveMode = True
352
353     if interactiveMode:
354         program = ""
355         try:
356             prompt = 'Camille> '
357             while True:
358                 line = input(prompt)
359                 if (line == "" and program != ""):
360                     parser_feed(program, parser)
361                     lexer.lineno = 1
362                     program = ""
363                     prompt = 'Camille> '
364                 else:
365                     if (line != ""):
366                         program += (line + '\n')
367                         prompt = ''
368
369         except EOFError as e:
370             sys.exit(0)
371
372     except Exception as e:
373         print(e)
374         sys.exit(-1)
375     else:
376         try:
377             with open(sys.argv[1], 'r') as script:
378                 file_string = script.read()
379                 parser_feed(file_string, parser)
380                 sys.exit(0)
381         except Exception as e:
382             sys.exit(-1)
383
384 main_func()
385 # end REPL

```

Programming Exercises for Chapter 10

Table 10.1 summarizes some of the details of the exercises here.

Exercise 10.1 Reimplement the interpreter given in this chapter for Camille 1.2.a to use the abstract-syntax representation of a named environment given in Section 9.8.4. This is Camille 1.2(named ASR).

Programming Exercise	Camille	Description	Start from	Representation of Environment
10.1	1.2(named ASR)	let, if/else	1.2	Named ASR
10.2	1.2(named LOLR)	let, if/else	1.2	Named LOLR
10.3	1.2(nameless ASR)	let, if/else	1.2	Nameless ASR
10.4	1.2(nameless LOLR)	let, if/else	1.2	Nameless LOLR
10.5	1.2(nameless CLS)	let, if/else	1.2	Nameless CLS
10.6	1.3	let, let*, if/else	1.2	CLS ASR LOLR

Table 10.1 New Versions of Camille, and Their Essential Properties, Created in the Chapter 10 Programming Exercises. (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

Exercise 10.2 Reimplement the interpreter given in this chapter for Camille 1.2 to use the list-of-lists representation of a named environment developed in Programming Exercise 9.8.5.a. This is Camille 1.2(named LOLR).

Programming Exercises 10.3–10.5 involve building Camille interpreters using nameless environments that are accessed through lexical addressing. These interpreters require an update to the definition of the `p_line_expr` function shown at the end of of Section 10.6.1 and repeated here:

```

82 def p_line_expr(t):
83     '''program : expression'''
84     t[0] = t[1]
85     print (evaluate_expr(t[0]))

```

We must replace line 85 with lines 85 and 86 in the following new definition:

```

82 def p_line_expr(t):
83     '''program : expression'''
84     t[0] = t[1]
85     lexical_addresser(t[0], 0, [])
86     print (evaluate_expr(t[0], empty_nameless_environment()))

```

Exercise 10.3 Reimplement the interpreter for Camille 1.2 to use the abstract-syntax representation of a nameless environment developed in Programming Exercise 9.8.9. This is Camille 1.2(nameless ASR).

Exercise 10.4 Reimplement the interpreter for Camille 1.2 to use the list-of-lists representation of a nameless environment developed in Programming Exercise 9.8.5.b. This is Camille 1.2(nameless LOLR).

Exercise 10.5 Reimplement the interpreter given in this chapter for Camille 1.2 to use the closure representation of a nameless environment developed in Programming Exercise 9.8.7. This is Camille 1.2(nameless CLS).

Exercise 10.6 Implement `let*` in Camille (with the same semantics it has in Scheme). For instance:

```

Camille> let*
          a = 3
          b = +(a, 4)
        in
          +(a, b)
10

```

This is Camille 1.3.

10.10 Thematic Takeaways

- A theme throughout this chapter (and in Chapters 11 and 12) is that to add a new feature or concept to Camille, we typically add:
 - a new production rule to the grammar
 - a new variant to the abstract-syntax representation of the `TreeNode` variant record representing a Camille expression
 - a new case to `evaluate_expr` corresponding to the new variant
 - any necessary and supporting data types/structures and libraries
- When adding a concept/feature to a defined programming language, we can either rely on support for that concept/feature in the defining language or implement the particular concept/feature manually (i.e., from first principles). For instance, we implemented conditional evaluation in Camille using the support for conditional evaluation found in Python (i.e., `if/else`). In contrast, we built support for local binding in Camille from scratch by defining an environment.

10.11 Chapter Summary

The main elements of an interpreter language implementation are:

- a read-eval-print loop user interface (e.g., `main_func`)
- a front end (i.e., scanner and parser, e.g., `parser.parse`)
- an abstract-syntax data type (e.g., the expression data type `TreeNode`)
- an interpreter (e.g., the `evaluate_expr` function)
- supporting data types/structures and libraries (e.g., `environment`)

Figure 10.4 and Table 10.2 indicate the dependencies between the versions of Camille developed in this chapter, including the programming exercises. Table 10.3 summarizes the concepts and features implemented in the progressive versions of Camille developed in this chapter, including the programming exercises. Table 10.4 outlines the configuration options available in Camille for aspects of the design of the interpreter (e.g., choice of representation of referencing environment).

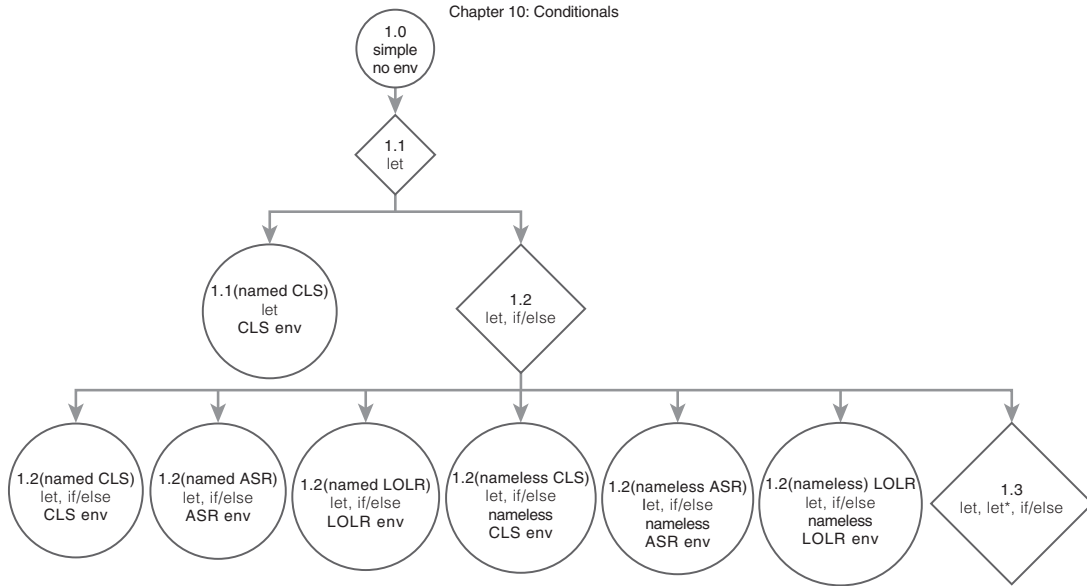


Figure 10.4 Dependencies between the Camille interpreters developed in this chapter. The semantics of a directed edge $a \rightarrow b$ are that version b of the Camille interpreter is an extension of version a (i.e., version b subsumes version a). (Key: circle = instantiated interpreter; diamond = abstract interpreter; ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

Version	Extends	Description
Chapter 10: Local Binding and Conditional Evaluation		
1.0	N/A	simple, no environment
1.1	1.0	let, named CLS ASR LOLR environment
1.1(named CLS)	1.1	let, named CLS environment
1.2	1.1	let, if/else
1.2(named CLS)	1.2	let, if/else, named CLS environment
1.2(named ASR)	1.2	let, if/else, named ASR environment
1.2(named LOLR)	1.2	let, if/else, named LOLR environment
1.2(nameless CLS)	1.2	let, if/else, nameless CLS environment
1.2(nameless ASR)	1.2	let, if/else, nameless ASR environment
1.2(nameless LOLR)	1.2	let, if/else, nameless LOLR environment
1.3	1.2	let, let*, (named nameless) (CLS ASR LOLR) environment

Table 10.2 Versions of Camille (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

	Version of Camille	1.0	1.1	1.2	1.3
Concepts/Data Structures	Expressed Values	integers	integers	integers	integers
	Denoted Values	integers	integers	integers	integers
	Representation of Environment	N/A	ASR CLS LOLR	ASR CLS LOLR	ASR CLS LOLR
	Local Binding	×	↑ let ↑	↑ let ↑	↑ let , let* ↑
	Conditionals	×	×	↓ if/else ↓	↓ if/else ↓
	Scoping	N/A	lexical	lexical	lexical

Table 10.3 Concepts and Features Implemented in Progressive Versions of Camille. The symbol ↓ indicates that the concept is supported through its implementation in the defining language (here, Python). The Python keyword included in each cell, where applicable, indicates which Python construct is used to implement the feature in Camille. The symbol ↑ indicates that the concept is implemented manually. The Camille keyword included in each cell, where applicable, indicates the syntactic construct through which the concept is operationalized. (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation. Cells in boldface font highlight the enhancements across the versions.)

Interpreter Design Options	
Type of Environment	Representation of Environment
named	abstract syntax
nameless	list of lists
	closure

Table 10.4 Configuration Options in Camille

10.12 Notes and Further Reading

The Camille programming language was first introduced and described in Perugini and Watkin (2018) (where it was called CHAMELEON), which also addresses its syntax and semantics, the educational aspects involved in the implementation of a variety of interpreters for it, its malleability, and student feedback to inspire its use for teaching languages. Online Appendix D is a guide to getting started with Camille; it includes details of its syntax and semantics, how to acquire access to the Camille Git repository necessary for using Camille, and the pedagogical approach to using the language.

Chapter 10 (as well as Chapter 11 and Sections 12.2, 12.4, and 12.6–12.7) is inspired by Friedman, Wand, and Haynes (2001, Chapter 3). Our contribution is the use of Python to build EOPL-style interpreters.

Chapter 11

Functions and Closures

The `eval-apply` cycle exposes the essence of a computer language.

— H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs* (1996)

WE continue our progressive development of the Camille programming language and interpreters for it in this chapter by adding support for functions and closures to Camille.

11.1 Chapter Objectives

- Describe the implementation of non-recursive and recursive functions through closures.
- Explore circular environment structures for supporting recursive functions.
- Explore representational strategies for closures.
- Explore representational strategies for circular environment structures for supporting recursive functions.

11.2 Non-recursive Functions

We begin by adding support for *non-recursive functions*—that is, functions that cannot make a call to themselves in their body.

11.2.1 Adding Support for User-Defined Functions to Camille

We desire user-defined functions to be first-class entities in Camille. This means that a function can be (1) the return value of an expression (altering the expressed values) and (2) bound to an identifier and stored in the environment of the interpreter (altering the denoted values). Adding user-defined, first-class functions to Camille alters the expressed and denoted values of the language:

expressed value = integer $\cup$ closure
 denoted value = integer $\cup$ closure

Thus,

expressed value = denoted value = integer $\cup$ closure

Recall that in Chapter 10 we had

expressed value = denoted value = integer

To support functions in Camille, we add the following rules to the grammar and corresponding pattern-action rules to the PLY parser generator:

`<expression>` ::= `<nonrecursive_function>`
`<expression>` ::= `<function_call>`

`ntFuncDecl`

`<nonrecursive_function>` ::= fun (`{<identifier>}*()`) `<expression>`

`ntFuncCall`

`<function_call>` ::= (`<expression>` `{<expression>}*()`)

```
def p_expression_function_decl(t):
    '''expression : FUN LPAREN parameters RPAREN expression
                  | FUN LPAREN RPAREN expression'''
    if len(t)==6:
        t[0] = Tree_Node(ntFuncDecl, [t[3], t[5]], None, t.lineno(1))
    else:
        t[0] = Tree_Node(ntFuncDecl, [t[4]], None, t.lineno(1))

def p_expression_function_call(t):
    '''expression : LPAREN expression arguments RPAREN
                  | LPAREN expression RPAREN '''
    if len(t)== 5:
        t[0] = Tree_Node(ntFuncCall, [t[3]], t[2], t.lineno(1))
    else:
        t[0] = Tree_Node(ntFuncCall, None, t[2], t.lineno(1))
```

The following example Camille expressions show functions with their evaluated results:

```
Camille> let --- identity function
          identity = fun (x) x
          in
          (identity 1)

1

Camille> let --- squaring function
          square = fun (x) *(x,x)
          in
          (square 2)
```

```

Camille> let
          area = fun (width,height) *(width,height)
          in
            (area 2,3)

6

```

To support functions, we must first determine the value to be stored in the environment for a function. Consider the following expression:

```

1  let
2    a = 1
3  in
4    let
5      f = fun (x) +(x,a)
6      a = 2
7    in
8      (f a)

```

What value should be inserted into the environment and mapped to the identifier f (line 5)? Alternatively, what value should be retrieved from the environment when the identifier f is evaluated (line 7)? The identifier f must be evaluated when f is applied (line 7). Thus, we must determine the information necessary to store in the environment to represent the value of a user-defined function. The necessary information that must be stored in a function value depends on which data is required to evaluate that function when it is applied (or invoked). To determine this, let us examine what must happen to invoke a function.

Assuming the use of *lexical scoping* (to bind each reference to a declaration), when a function is applied, the body of the function must be evaluated in an environment that binds the formal parameters to the arguments and binds the free variables in the body of the function to their values *at the time the function was created* (i.e., *deep binding*). In the Camille expression previously shown, when f is called, its body must be evaluated in the environment

$\{(x,2), (a,1)\}$ (i.e., static scoping)

and not in the environment

$\{(x,2), (a,2)\}$ (i.e., dynamic scoping)

Thus, we must call

evaluate_expr (+(x,a), (x,2), (a,1))

and not call

evaluate_expr (+(x,a), (x,2), (a,2))

Thus,

```

Camille> let
          a = 1
          in
            let

```

```

    f = fun (x) + (x, a)
    a = 2
in
    (f a)

3

```

For a function to retain the bindings of its free variables *at the time it was created*, it must be a *closed* package and completely independent of *the environment in which it is called*. This package is called a *closure* (as discussed in Chapters 5 and 6).

11.2.2 Closures

A closure must contain:

- the list of formal parameters¹
- the body of the function (an expression)
- the bindings of its free variables (an environment)

We say that this function is *closed over* or *closed in* its *creation* environment. A closure resembles an *object* from object-oriented programming—both have state and behavior. A closure consists of a pair of (expression, environment) pointers. Thus, we can think of a closure as a `cons` cell, which also contains two pointers (Section 5.5.1). In turn, we can think of a function value as an abstract data type (ADT) with the following *interface*:

- `make_closure`: a constructor that builds or packages a closure
- `apply_closure`: an observer that applies a closure

where the following equality holds:

```

apply_closure (make_closure(arglist, body, environ), arguments) =
evaluate_expr(body, extend_environment(parameters, arguments, environ))

```

When a function is called, the body of the function is evaluated in an environment that binds the formal parameters to the arguments and binds the free variables in the body of the function to their values at the time the function was created.

Let us build an abstract-syntax representation in Python for Camille closures (Figure 11.1):

Closure	parameters: list of parameter names
	body: root <code>TreeNode</code> of function
	environ: environment in which the function is evaluated

Figure 11.1 Abstract-syntax representation of our `Closure` data type in Python.

1. Recall, from Section 5.4.1, the distinction between *formal* and *actual* parameters or, in other words, the difference between *parameters* and *arguments*.

```

class Closure:
    def __init__(self, parameters, body, environ):
        self.parameters = parameters
        self.body = body
        self.environ = environ

def is_closure(cls):
    return isinstance(cls, Closure)

def make_closure(parameters, body, environ):
    return Closure(parameters, body, environ)

def apply_closure(cls, arguments):
    return evaluate_expr(cls.body,
        extend_environment(cls.parameters, arguments, cls.environ))

```

We can also represent a (expressed and denoted) closure value in Camille as a Python closure:

```

def make_closure(parameters, body, environ):
    return lambda arguments: evaluate_expr(body,
        extend_environment(parameters, arguments, environ))

def apply_closure(cls, arguments):
    return cls(arguments)

def is_closure(cls):
    return callable(cls)

```

Using either of these representations for Camille closures, the following equality holds:

```

apply_closure (make_closure(arglist, expr.children[1], environ), arguments) =
evaluate_expr(cls.body, extend_environment(cls.parameters, arguments, cls.environ))

```

Figures 11.2 and 11.3 illustrate how closures are stored in abstract-syntax and list-of-lists representations, respectively, of named environments.

11.2.3 Augmenting the `evaluate_expr` Function

With this foundation in place, only minor modifications to the Camille interpreter are necessary to support first-class functions:

```

1  def evaluate_expr(expr, environ):
2      if ...:
3          ...
4          ...
5          ...
6      elif ...:
7          ...
8
9      elif expr.type == ntFuncDecl:
10         if (len(expr.children) == 2):
11             arglist = evaluate_expr(expr.children[0], environ)
12             body = expr.children[1]
13         else:

```

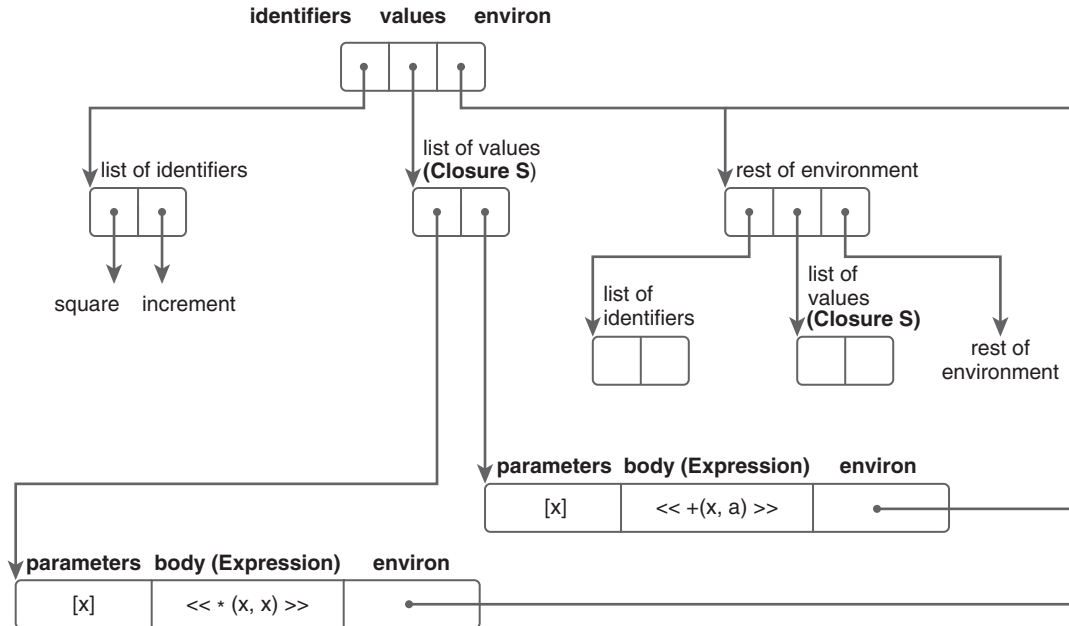


Figure 11.2 An abstract-syntax representation of a non-recursive, named environment (Section 9.8.4).

```

14     arglist = []
15     body = expr.children[0]
16     return make_closure(arglist, body, environ)
17
18 elif expr.type == ntParameters:
19     ParamList = []
20     ParamList.append(expr.children[0])
21
22     if len(expr.children) > 1:
23         ParamList.extend(evaluate_expr(expr.children[1], environ))
24     return ParamList
25
26 elif expr.type == ntArguments:
27     ArgList = []
28     ArgList.append(evaluate_expr(expr.children[0], environ))
29
30     if len(expr.children) > 1:
31         ArgList.extend(evaluate_expr(expr.children[1], environ))
32     return ArgList
33
34 elif expr.type == ntFuncCall:
35     cls = evaluate_expr(expr.leaf, environ)
36     if len(expr.children) != 0:
37         arguments = evaluate_expr(expr.children[0], environ)
38     else:
39         arguments = []
40
41     if is_closure(cls):
42         return apply_closure(cls, arguments)
43     else:

```

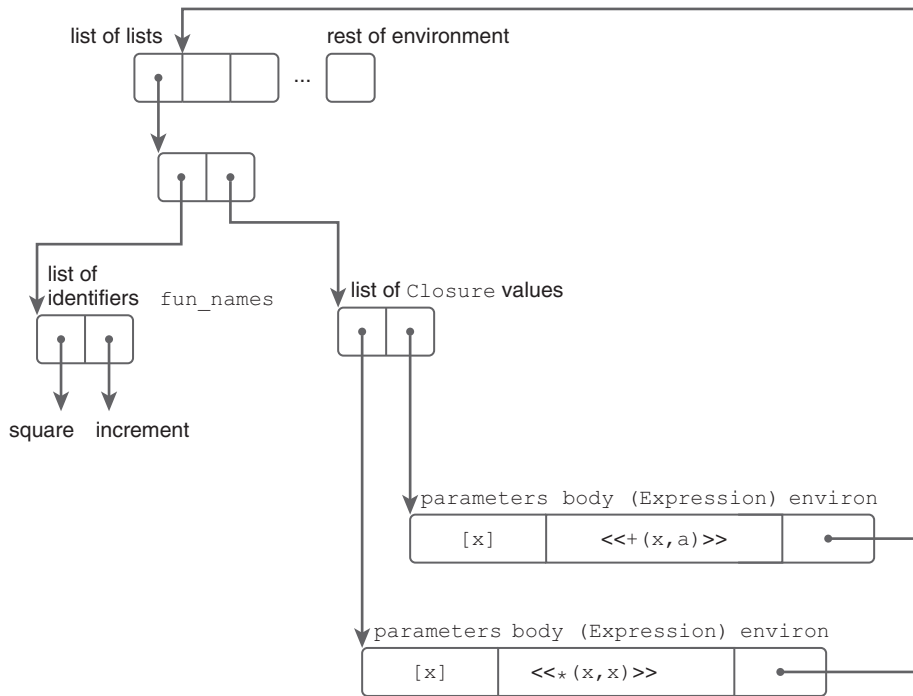


Figure 11.3 A list-of-lists representation of a non-recursive, named environment using the structure of Programming Exercise 9.8.5.a.

```

44      # Error: function is not a closure;
45      # attempt to apply a non-function
46      raise InterpreterException(expr.linenumber,
47      "'%' is not a function" % expr.leaf.leaf)

```

Example expressions in this version of Camille with their evaluated results follow:

```

Camille> let f = fun (x) x in (f 1)

1
Camille> let f = fun (x) *(x,x) in (f 2)

4
Camille> let f = fun (width,height) *(width,height) in (f 2,3)

6
Camille> let a = 1 in let f = fun (x) +(x,a) a = 2 in (f a)

3

```

Consider the Camille rendition (and its output) of the Scheme program shown at the start of Section 6.11 to demonstrate deep, shallow, and ad hoc binding:

```

Camille> let
  y = 3

```

```

in
  let
    x = 10
    --- create closure here: deep binding
    f = fun (x) *(y, +(x,x))
  in
    let
      y = 4
    in
      let
        y = 5
        x = 6
        --- create closure here: shallow binding
        g = fun (x, y) *(y, (x y))
      in
        let
          y = 2
        in
          --- create closure here: ad hoc binding
          (g f,x)
216

```

This result (216) demonstrates that Camille implements deep binding to resolve nonlocal references in the body of first-class functions.

Note that this version of Camille does *not* support recursion:

```

Camille> let
  sum = fun (x) if zero?(x) 0 else +(x, (sum decl(x)))
in
  (sum 5)

Runtime Error: Line 2: Unbound Identifier 'sum'

```

However, we can simulate recursion with `let` as done in the definition of the function `length` in Section 5.9.3:

```

Camille> let
  sum = fun (s, x)
    if zero?(x)
      0
    else
      +(x, (s s,decl(x)))
in
  (sum sum, 5)

15

```

11.2.4 A Simple Stack Object

Through an extension of the prior idea, even though it does not have support for object-oriented programming, Camille can be used to build object-oriented abstractions. For instance, the following Camille program simulates the implementation of a simple stack class with two constructors (`new_stack` and `push`) and three observers/messages (`empty_stack?`, `top`, and `pop`). The output of this program is 3. The stack object is represented as a Camille closure:

```

let
  --- constructor
  new_stack = fun ()
    fun(msg)
      if eqv?(msg, 1)
        -1 --- error: cannot top an empty stack
      else
        if eqv?(msg, 2)
          -2 --- error: cannot pop an empty stack
        else
          1 --- represents true: stack is empty

  --- constructor
  push = fun (elem, stack)
    fun (msg)
      if eqv?(msg,1) elem
      else if eqv?(msg,2) stack
      else 0

  --- observers
  emptystack? = fun (stack) (stack 0)
  top = fun (stack) (stack 1)
  pop = fun (stack) (stack 2)
in
  let
    simplestack = (new_stack)
  in
    (top (push 3, (push 2, (push 1, simplestack))))

```

Conceptual Exercises for Section 11.2

Exercise 11.2.1 What is the difference between a *closure* and a *function*? Explain.

Exercise 11.2.2 User-defined functions are typically implemented with a run-time stack of activation records. Where is the run-time stack in the user-defined Camille functions implemented in this section? Explain.

Exercise 11.2.3 As discussed in this section, this version of Camille does not support recursion. However, we simulated recursion by passing a function to itself—so it can call itself. Is there another method of simulating recursion in this non-recursive version of the Camille interpreter? In particular, explore the relationship between *dynamic scoping* and the `let*` expression (Programming Exercise 10.6). Consider the following Camille expression:

```

--- mutually recursive iseven? and isodd? functions
let*
  iseven? = fun(x) if zero?(x) 1 else (isodd? decl(x))
  isodd?  = fun(x) if zero?(x) 0 else (iseven? decl(x))
in
  (isodd? 15)

```

Will this expression evaluate properly using *lexical scoping* in the version of the Camille interpreter supporting only non-recursive functions? Will this expression evaluate properly using *dynamic scoping* in the version of the Camille interpreter supporting only non-recursive functions? Explain.

Programming Exercise	Camille	Description	Start from	Representation of Closures	Representation of Environment
11.2.6	2.0(verify ASR)	verify environment	2.0	ASR CLS	ASR
11.2.7	2.0(verify LOLR)	verify environment	2.0	ASR CLS	LOLR
11.2.8	2.0(verify CLS)	verify environment	2.0	ASR CLS	CLS
11.2.9	2.0(nameless LOLR)	nameless environment	2.0(verify LOLR)	ASR CLS	LOLR
11.2.10	2.0(nameless ASR)	nameless environment	2.0(verify ASR)	ASR CLS	ASR
11.2.11	2.0(nameless CLS)	nameless environment	2.0(verify CLS)	ASR CLS	CLS
11.2.12	2.0(dynamic scoping)	dynamic scoping	2.0	lambda expression	CLS ASR LOLR

Table 11.1 New Versions of Camille, and Their Essential Properties, Created in the Section 11.2.4 Programming Exercises. (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

Programming Exercises for Section 11.2

Table 11.1 summarizes the properties of the new versions of the Camille interpreter developed in the following programming exercises. Figure 11.4 presents the dependencies between the versions of Camille developed thus far, including in these programming exercises.

Exercise 11.2.4 Modify the definition of the `new_counter` function in Python in Section 6.10.2 to incorporate a step on the increment into the counter closure.

Examples:

```
>>> counter1 = new_counter(0,1)
>>> counter2 = new_counter(1,2)
>>> counter50 = new_counter(100,50)
>>>
>>> print(counter1())
1
>>> print(counter1())
2
>>> print(counter2())
3
>>> print(counter2())
5
>>> print(counter1())
3
>>> print(counter1())
4
>>> print(counter2())
7
>>> print(counter50())
150
>>> print(counter50())
200
>>> print(counter50())
250
>>> print(counter1())
5
```

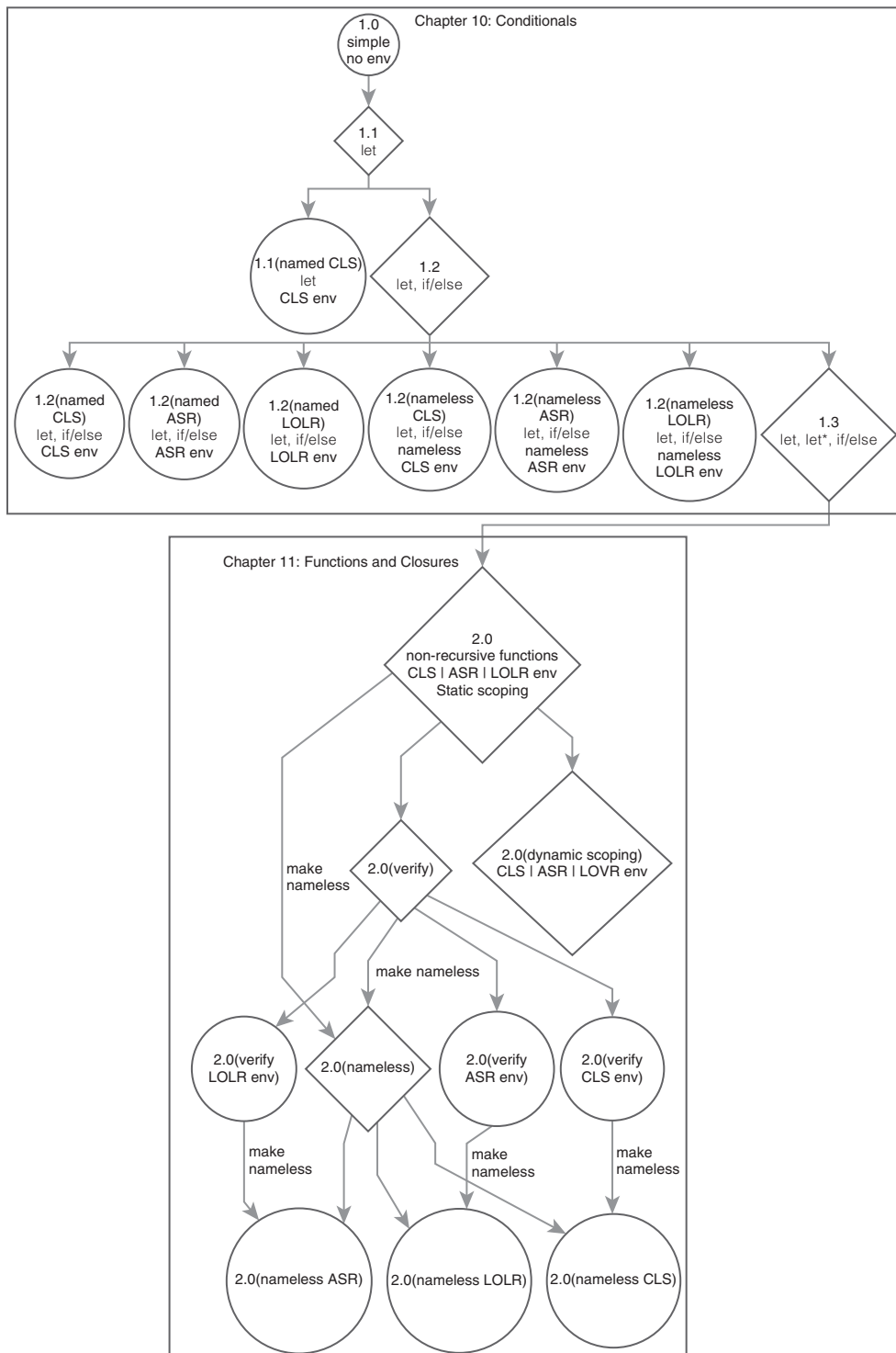


Figure 11.4 Dependencies between the Camille interpreters developed thus far, including those in the programming exercises. The semantics of a directed edge $a \rightarrow b$ are that version b of the Camille interpreter is an extension of version a (i.e., version b subsumes version a). (Key: circle = instantiated interpreter; diamond = abstract interpreter; ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

Exercise 11.2.5 (Friedman, Wand, and Haynes 2001, Exercise 3.23, p. 90) Implement a lexical-address calculator, like that of Programming Exercise 6.5.3, for the version of Camille defined in this section. The calculator must take an abstract-syntax representation of a Camille expression and return another abstract-syntax representation of it. In the new representation, the `leaf` of every `ntIdentifier` parse tree node should be replaced with a `[var, depth, pos]` list, where *(depth, pos)* is the lexical address for this occurrence of the variable *var*, unless the occurrence of `ntIdentifier` is free. Name the top-level function of the lexical-address calculator `lexical_address`, and define it to accept and return an abstract-syntax representation of a Camille program. However, use the generated parser and `concrete2abstract` function in Section 9.6 to build the abstract-syntax representation of the Camille input expression. Use the `abstract2concrete` function to translate the lexically addressed abstract-syntax representation of a Camille program to a string (Programming Exercise 9.6.2). Thus, the program must take a string representation of a Camille expression as input and return another string representation of it where the occurrence of each variable reference *v* is replaced with a `[v, depth, pos]` list, where *(depth, pos)* is the lexical address for this occurrence of the variable *v*, unless the occurrence of *v* is free. If the variable reference *v* is free, print `['v ', 'free']` as shown in line 7 of the following examples.

Examples:

```

1  $ ./run
2  Camille> let a = 5 in a
3
4  let a = 5 in ['a', 0, 0]
5  Camille> let a = 5 in i
6
7  let a = 5 in ['i', 'free']
8  Camille> let a = 2 in let b = 3 in a
9
10 let a = 2 in let b = 3 in ['a', 1, 0]
11 Camille> let f = fun (y,z) +(y,-(z,5)) in (f 2,28)
12
13 let f=fun(y, z) +(['y', 0, 0],-(['z', 0, 1], 5)) in (['f', 0, 0] 2,28)

```

Exercise 11.2.6 (Friedman, Wand, and Haynes 2001, Exercise 3.24, p. 90) Modify the Camille interpreter defined in this section to demonstrate that the value bound to each identifier is found at the position given by its lexical address. Specifically, modify the `evaluate_expr` function so that it accepts the output of the lexical-address calculator function `lexical_address` built in Programming Exercise 11.2.5 and passes both the identifier and the lexical address of each reference to the `apply_environment` function. The function `apply_environment` must look up the value bound to the identifier in the usual way. It must then compare the lexical address to the actual rib (i.e., depth and position) in which the value is found and print an informative message in the format demonstrated in the following examples. If the `leaf` of an `ntIdentifier`

parse tree node is free, print [*v* : free] as shown in line 9. Name the lexical-address calculator function `lexical_address` and invoke it from the `main_func` function (lines 46 and 69):

```

1  ...
2
3  global_tree = ""
4
5  ...
6
7  def p_line_expr(t):
8      '''program : expression'''
9      t[0] = t[1]
10     # save global_tree
11     global global_tree
12     global_tree = t[0]
13
14  ...
15
16  def parser_feed(s,parser):
17      pattern = re.compile("[^ \t]+")
18      if pattern.search(s):
19          try:
20              parser.parse(s)
21          except InterpreterException as e:
22              print ( "Line %s: %s" % (e.linenum, e.message))
23              if ( e.additional_information != None ):
24                  print("Additional information:")
25                  print(e.additional_information)
26          except Exception as e:
27              print ("Unknown Error occurred ")
28              print ("(this is normally caused by a Python syntax error)")
29              raise e
30
31  def main_func():
32      parser = yacc.yacc()
33      interactiveMode = False
34      global global_tree
35      if len(sys.argv) == 1:
36          interactiveMode = True
37
38      if interactiveMode:
39          program = ""
40          try:
41              prompt = 'Camille> '
42              while True:
43                  line = input(prompt)
44                  if (line == "" and program != ""):
45                      parser_feed(program,parser)
46                      lexical_address(global_tree[0], 0, [])
47                      print(evaluate_expr(global_tree[0], empty_environment()))
48                      lexer.lineno = 1
49                      global_tree = []
50                      program = ""
51                      prompt = 'Camille> '
52                  else:
53                      if (line != ""):
54                          program += (line + '\n')
55                          prompt = ''
56

```

```

57         except EOFError as e:
58             sys.exit(0)
59
60         except Exception as e:
61             print(e)
62             sys.exit(-1)
63     else:
64         try:
65             with open(sys.argv[1], 'r') as script:
66                 file_string = script.read()
67                 parser_feed(file_string, parser)
68                 for tree in global_tree:
69                     lexical_address(tree, 0, [])
70                     print(evaluate_expr(tree, empty_environment()))
71             sys.exit(0)
72         except Exception as e:
73             print(e)
74             sys.exit(-1)
75
76 main_func()

```

Use an abstract-syntax representation of the environment. Thus, you may find it helpful to first complete Programming Exercise 9.8.9. Also, use the following abstract-syntax representation definition of `apply_environment` to verify the correctness of your lexical-address calculator:

```

def apply_environment(envIRON, symbol, depth, position):
    def apply_environment_with_depth(envIRON1, current_depth):
        if envIRON1.flag == "empty-environment-record":
            raise IndexError

        elif envIRON1.flag == "extended-environment-record":
            try:
                pos = envIRON1.symbols.index(symbol)
                value = envIRON1.values[pos]

                print ("Just found the value %s at depth %s = %s and "
                       "position %s = %s." % (value, current_depth,
                                               depth, pos, position))

                return value
            except (IndexError, ValueError):
                return apply_environment_with_depth(envIRON1.envIRON,
                                                       current_depth+1)

        elif envIRON1.flag == \
            "recursively-extended-environment-record":
            try:
                pos = envIRON1.fun_names.index(symbol)
                value = make_closure(envIRON1.parameterlists[pos],
                                     envIRON1.bodies[pos], envIRON1)

                print ("Just found the value %s at depth %s = %s and "
                       "position %s = %s." % (value, current_depth,
                                               depth, pos, position))

                return value
            except:
                return apply_environment(envIRON1.envIRON, symbol)

    return apply_environment_with_depth(envIRON, 0)

```

Examples:

```

1  $ ./run
2  Camille> let a = 5 in a
3
4  Just found the value 5 at depth 0 = 0 and position 0 = 0.
5  5
6  let a = 5 in [0,0]
7  Camille> let a = 5 in i
8
9  [i : free]
10 (3, "Unbound identifier 'i'")
11 Camille> let a = 2 in let b = 3 in a
12
13 Just found the value 2 at depth 1 = 1 and position 0 = 0.
14 2
15 Camille> let f = fun (y,z) +(y,-(z,5)) in (f 2,28)
16
17 Just found the value <function make_closure.<locals>.<lambda> at
18   0x1085b9378> at depth 0 = 0 and position 0 = 0.
19 Just found the value 2 at depth 0 = 0 and position 0 = 0.
20 Just found the value 28 at depth 0 = 0 and position 1 = 1.
21 25

```

Exercise 11.2.7 Complete Programming Exercise 11.2.6, but this time use a list-of-lists representation of an environment from Programming Exercise 9.8.5.a.

Exercise 11.2.8 Complete Programming Exercise 11.2.6, but this time use a closure representation of an environment from Section 9.8.3.

Exercise 11.2.9 Since lexically bound identifiers are superfluous in the abstract-syntax tree processed by an interpreter, we can completely replace each lexically bound identifier with its lexical-address. In this exercise, you build an interpreter that supports functions and uses a list-of-lists representation of a nameless environment. In other words, extend Camille 2.0(named LOLR) built in Programming Exercise 11.2.7 to use a completely nameless environment. Alternatively, extend Camille 1.2(nameless LOLR) built in Programming Exercise 10.4 with functions.

- (a) Modify your solution to Programming Exercise 11.2.5 so that its output for a reference contains only the lexical address, not the identifier. That is, replace the leaf of each `ntIdentifier` node with a `[depth, pos]` list, where (*depth*, *pos*) is the lexical address for this occurrence of the identifier, unless the occurrence of `ntIdentifier` is free. If the leaf of an `ntIdentifier` node is free, print `[free]` as shown in line 7 of the following examples.

Examples:

```

1  $ ./run
2  Camille> let a = 5 in a
3
4  let a = 5 in [0,0]
5  Camille> let a = 5 in i
6
7  let a = 5 in [free]

```

```

8 Camille> let a = 2 in let b = 3 in a
9
10 let a = 2 in let b = 3 in [1,0]
11 Camille> let f = fun (y,z) + (y,-(z,5)) in (f 2,28)
12
13 let f = fun(y, z) + ([0,0], -([0,1], 5)) in ([0,0] 2, 28)
14 Camille>

```

- (b) (Friedman, Wand, and Haynes 2001, Exercise 3.25, p. 90) Build a list-of-lists (i.e., ribcage) representation of a *nameless* environment (Figure 11.5) with the following interface:

```

def empty_nameless_environment()
def extend_nameless_environment (values, environment)
def apply_nameless_environment (environment, depth, position)

```

In other words, solve Programming Exercise 9.8.5.b.

In this representation of a nameless environment, the lexical address of a variable reference *v* is (*depth*, *position*) and indicates from where to find (and retrieve) the value bound to the identifier used in a reference (i.e., at rib *depth* in position *position*). Thus, invoking the function `apply_nameless_environment` with the parameters `environment`, `depth`, and `position` retrieves the value at the (`depth`, `position`) address in the environment.

- (c) Adapt the `evaluate_expr`, `make_closure`, and `apply_closure` functions of the version of Camille defined in this section to use a LOLR of a nameless environment. Handle free identifiers as follows:

```

Camille> a
Lexical Address error: Unbound Identifier 'a'

```

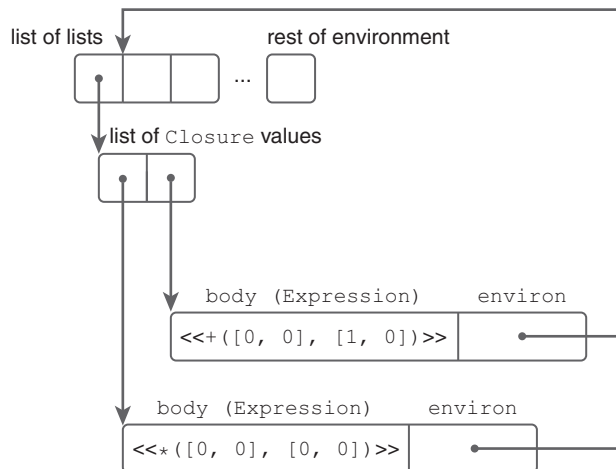


Figure 11.5 A list-of-lists representation of a non-recursive, nameless environment.

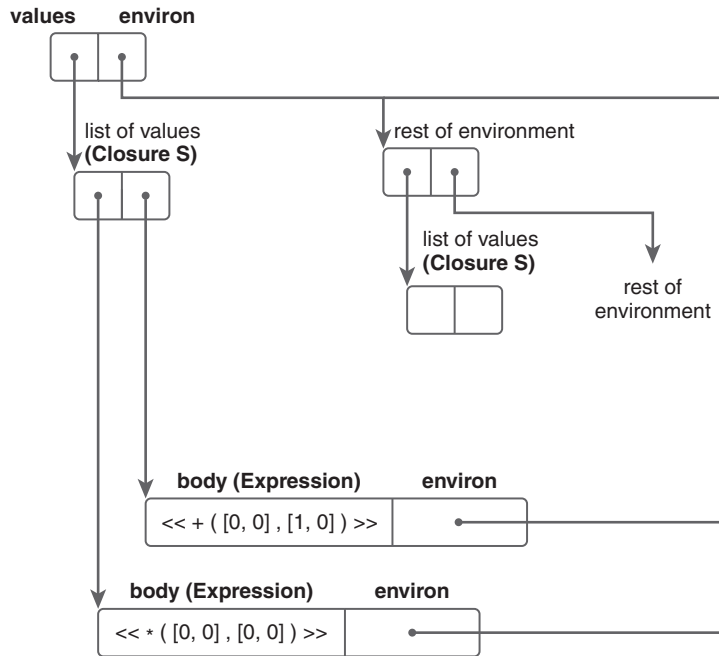


Figure 11.6 An abstract-syntax representation of a non-recursive, nameless environment using the structure of Programming Exercise 9.8.9.

Name the lexical-address calculator function `lexical_address` and invoke it from the `main_func` function in lines 46 and 69 as shown in Programming Exercise 11.2.6.

Exercise 11.2.10 Complete Programming Exercise 11.2.9, but this time use an abstract-syntax representation of a nameless environment (Figure 11.6). In other words, modify Camille 2.0(verify ASR) as built in Programming Exercise 11.2.6 to use a completely nameless environment. Alternatively, extend Camille 1.2(nameless ASR) as built in Programming Exercise 10.3 with functions. Start by solving Programming Exercise 9.8.9 (i.e., developing an abstract-syntax representation of a nameless environment).

Exercise 11.2.11 Complete Programming Exercise 11.2.9, but this time use a closure representation of a nameless environment. In other words, modify Camille 2.0(verify CLS) as built in Programming Exercise 11.2.8 to use a completely nameless environment. Alternatively, extend Camille 1.2(nameless CLS) as built in Programming Exercise 10.5 with functions. Start by solving Programming Exercise 9.8.7 (i.e., developing a closure representation of a nameless environment).

Exercise 11.2.12 (Friedman, Wand, and Haynes 2001, Exercise 3.30, p. 91) Modify the Camille interpreter defined in this section to use *dynamic scoping* to bind

references to declarations. For instance, in the Camille function `f` shown here, the reference to the identifier `s` in the expression `*(t, s)` on line 5 is bound to 15, not 10; thus, the return value of the call to `(f s)` on line 8 is 225 (under dynamic scoping), not 150 (under static/lexical scoping).

Example:

```

1  Camille> let
2              s = 10
3          in
4              let
5                  f = fun (t) *(t, s)
6                  s = 15
7              in
8                  (f s)
9
10 225

```

Represent user-defined functions with `lambda` expressions in Python of the form `lambda arguments, environ: ...`. Rather than creating a closure when a function is defined, create a closure when a function is called and pass to it the environment in which it is called. Do these user-defined functions with `lambda` expressions have any free variables? Can this non-recursive, dynamic scoping version of the Camille interpreter evaluate a recursive function?

Note that you must *not* use the (Python closure or abstract syntax) closure data type, interface, and implementation given in this section to solve this exercise. Rather, you must represent user-defined Camille functions with a Python `lambda` expression of the form `lambda arguments, environ: ...`.

11.3 Recursive Functions

We now add support for *recursive functions*—that is, functions that can make a call to themselves in their body.

11.3.1 Adding Support for Recursion in Camille

To support recursion in Camille, we add the following rules to the grammar and corresponding pattern-action rules to the PLY parser generator:

`<expression>` ::= `<letrec_expression>`

`ntLetRec`

`<letrec_expression>` ::= `letrec <letrec_statement> in <expression>`

`ntLetRecStatement`

`<letrec_statement>` ::= `<letrec_assignment>`

`<letrec_statement>` ::= `<letrec_assignment> <letrec_statement>`

`ntLetRecAssignment`

`<letrec_assignment>` ::= `<identifier> = <recursive_function>`

ntRecFuncDecl

<recursive_function> ::= fun ({<identifier>}*(¹)) <expression>

```
def p_expression_let_rec(t):
    '''expression : LETREC letrec_statement IN expression'''
    t[0] = Tree_Node(ntLetRec, [t[2], t[4]], None, t.lineno(1))

def p_letrec_statement(t):
    '''letrec_statement : letrec_assignment
                        | letrec_assignment letrec_statement'''
    if len(t) == 3:
        t[0] = Tree_Node(ntLetRecStatement, [t[1], t[2]], None,
                        t.lineno(1))
    else:
        t[0] = Tree_Node(ntLetRecStatement, [t[1]], None, t.lineno(1))

def p_letrec_assignment(t):
    '''letrec_assignment : IDENTIFIER EQ rec_func_decl'''
    t[0] = Tree_Node(ntLetRecAssignment, [t[3]], t[1], t.lineno(1))

def p_expression_rec_func_decl(t):
    '''rec_func_decl : FUN LPAREN parameters RPAREN expression
                    | FUN LPAREN RPAREN expression'''
    if len(t) == 6:
        t[0] = Tree_Node(ntRecFuncDecl, [t[3], t[5]], None, t.lineno(1))
    else:
        t[0] = Tree_Node(ntRecFuncDecl, [t[4]], None, t.lineno(1))
```

Example expressions in this version of Camille follow:

```
Camille> letrec      --- recursive squaring function
    square = fun (n) if eqv?(n,1) 1
                  else decl(+((square -(n,1)), *(2,n)))
    in
    (square 2)

4

Camille> letrec      --- factorial function
    fact = fun(x) if zero?(x) 1 else *(x, (fact decl(x)))
    in
    (fact 5)

120

Camille> letrec --- mutually recursive iseven? and isodd? functions
    iseven? = fun(x) if zero?(x) 1 else (isodd? decl(x))
    isodd?  = fun(x) if zero?(x) 0 else (iseven? decl(x))
    in
    (isodd? 15)

1
```

11.3.2 Recursive Environment

To support recursion, we must modify the environment. Specifically, we must ensure that the environment stored in the closure of a recursive

function contains the function itself. To do so, we add a new function `extend_environment_recursively` to the environment interface. Three possible representations of a recursive environment are a closure, abstract syntax, and a list-of-lists.

Closure Representation of Recursive Environment

The closure representation of a recursive environment is the same as the closure representation of a non-recursive environment except for the following definition of the `extend_environment_recursively` function:

```

1  def extend_environment_recursively (fun_names, parameterlists,
2                                     bodies, environ):
3      recursive_envIRON = lambda identifier: tryexcept(identifier)
4      def tryexcept(identifier):
5          try:
6              position = fun_names.index(identifier)
7              return make_closure(parameterlists[position],
8                                  bodies[position], recursive_envIRON)
9          except:
10             val = apply_environment(envIRON, identifier)
11             return val
12
13     return recursive_envIRON

```

The recursive environment is initially created as a Python closure or `lambda` expression (line 3). As usual with a closure representation of an environment, that Python closure is invoked when `apply_environment` is called. At that time, the closure for the recursive function is created (lines 7–8), and contains the recursive environment (line 8) originally created (line 3). Thus, the environment containing the recursive function is found in the closure representing the recursive function.

The relationship between the `apply_environment(envIRON, symbol)` and `extend_environment_recursively(fun_names, parameterlists, bodies, environ)` functions is specified as follows:

1. If `name` is one of the names in `fun_names`, and `parameters` and `body` are the corresponding formal parameter list and function body in `parameterlists` and `bodies`, respectively, then

`apply_environment(e', name) = make_closure(parameters, body, e')`

where `e'` is

`extend_environment_recursively(fun_names, parameterlists, bodies, environ)`

2. Else,

`apply_environment(e', name) = apply_environment(envIRON, name)`

Abstract-Syntax Representation of Recursive Environment

To create an abstract-syntax representation of a recursive environment, we augment the abstract-syntax representation of a non-recursive environment with a new set of fields for a recursively-extended-environment-record:

```

1  class Environment:
2      def __init__(self, symbols=None, values=None, fun_names=None,
3                  parameterlists=None, bodies=None, environ=None):
4          if symbols == None and values == None and fun_names == None and \
5              parameterlists == None and bodies == None and environ==None:
6              self.flag = "empty-environment-record"
7          elif fun_names == None and parameterlists == None and \
8              bodies == None:
9              self.flag = "extended-environment-record"
10             self.symbols = symbols
11             self.values = values
12             self.environ = environ
13         elif symbols == None and values == None:
14             self.flag = "recursively-extended-environment-record"
15             self.fun_names = fun_names
16             self.parameterlists = parameterlists
17             self.bodies = bodies
18             self.environ = environ

```

We must also add a new function `extend_environment_recursively` to the interface and augment the definition `apply_environment` in the implementation to handle the new `recursively-extended-environment-record` (lines 30–36):

```

19  def extend_environment_recursively(fun_names1,
20                                   parameterlists1, bodies1,
21                                   environ1):
22      return Environment(fun_names=fun_names1,
23                        parameterlists=parameterlists1,
24                        bodies=bodies1, environ=environ1)
25
26  def apply_environment(envIRON, symbol):
27      if environ.flag == "empty-environment-record":
28          ...
29      elif environ.flag == "extended-environment-record":
30          ...
31      elif environ.flag == "recursively-extended-environment-record":
32          try:
33              position = environ.fun_names.index(symbol)
34              return make_closure(environ.parameterlists[position],
35                                environ.bodies[position], environ)
36          except:
37              return apply_environment(environ.environ, symbol)

```

The circular structure of the abstract-syntax representation of a recursive environment is presented in Figure 11.7. In this figure, `<<if zero?(x) then 1 else (odd decl(x))>>` represents the abstract-syntax representation of a Camille expression (i.e., `TreeNode`). In general, in this chapter, `<<x>>` represents the abstract-syntax representation of `x`. Notice that the environment contained in the closure of each recursive function is the

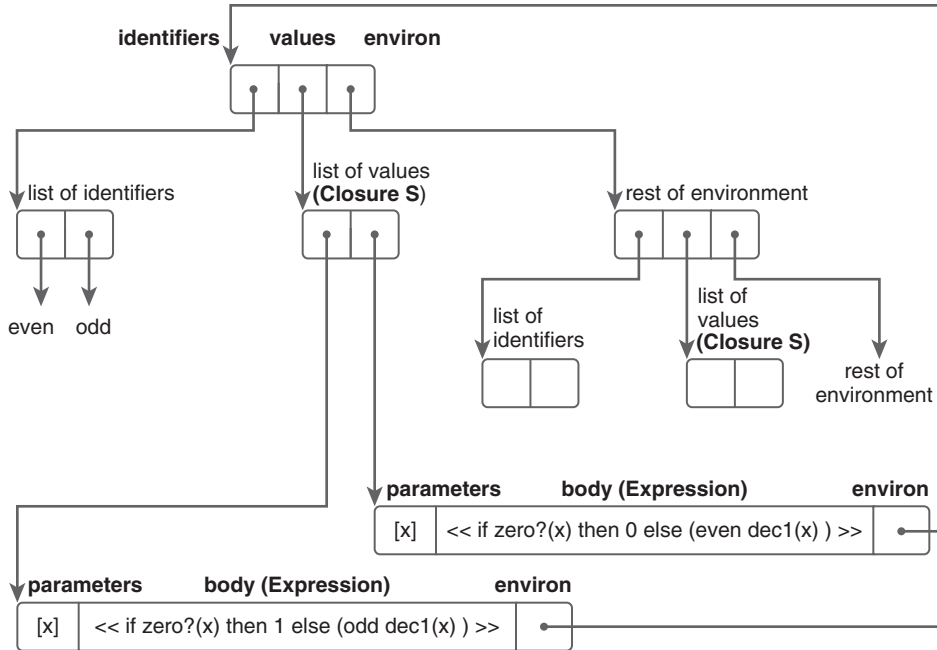


Figure 11.7 An abstract-syntax representation of a circular, recursive, named environment.

environment containing the closure, not the environment in which the closure is created.

List-of-Lists Representation of Recursive Environment

In the closure and abstract-syntax representations of a recursive environment just described, a new closure is built each time a function is retrieved from the environment (i.e., when `apply_environment` is called). This is unnecessary (and inefficient) since the environment for the closure being repeatedly built is always the same. If we use a list-of-lists (i.e., *ribcage*) representation of a recursive environment, we can build each closure only once, in the `extend_environment_recursively` function, when the recursive function is encountered:

```
def extend_environment_recursively(fun_names, parameterlists,
                                  bodies, environ):
    closures = []
    recenv = extend_environment(fun_names, closures, environ)

    for paramlist, body in zip(parameterlists, bodies):
        closures.append(make_closure(paramlist, body, recenv))
    return recenv
```

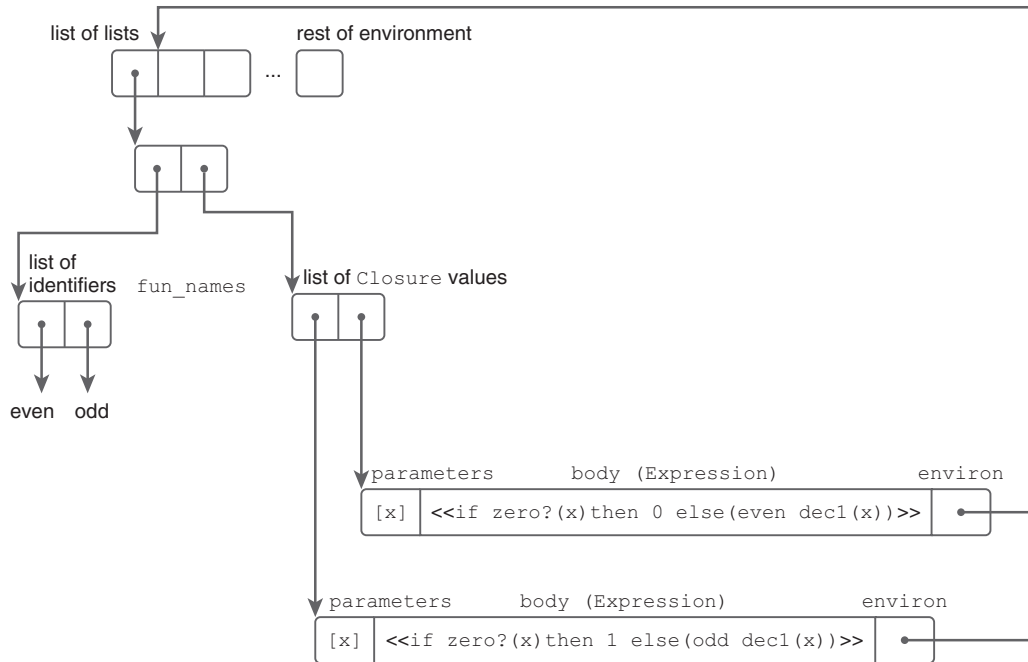


Figure 11.8 A list-of-lists representation of a circular, recursive, named environment.

Everything else from the list-of-lists representation of a non-recursive environment remains the same in the list-of-lists representation of a recursive environment. The circular structure of the list-of-lists representation of a recursive environment is shown in Figure 11.8.

11.3.3 Augmenting `evaluate_expr` with New Variants

The final modification we must make to support recursive functions is an augmentation of the `evaluate_expr` function to process the new variants of `TreeNode` that we added to support recursion—that is, `ntLetRec`, `ntLetRecStatement`, `ntLetRecAssignment`, and `ntRecFuncDecl`.

We start by discussing how the bindings in a `letrec` expression are represented in the abstract-syntax tree. Subtrees of the `ntLetRecStatement` variant are traversed in the same way as the `ntLetStatement` and `ntLetStarStatement` variants. However, the semantics of these expressions differ in how values are added to the environment. Specifically, `ntLetRecAssignment` returns a list containing three lists: a list of identifiers to which each function is bound, the parameter lists of each function, and the body of each function.

The following augmented definition of `evaluate_expr` describes how a `letrec` expression is evaluated:

```

1  def evaluate_expr (expr, environ):
2      try:
3          ...
4          ...
5          ...
6      elif expr.type == ntLetRec:
7          # assignment
8          FunctionDataList = evaluate_expr(expr.children[0], environ)
9
10         return evaluate_expr(expr.children[1],
11                               extend_environment_recursively(FunctionDataList[0],
12                                                                FunctionDataList[1],
13                                                                FunctionDataList[2],
14                                                                environ)) # evaluation
15
16     elif expr.type == ntLetRecStatement:
17         FunctionData = evaluate_expr(expr.children[0], environ)
18         if len(expr.children) > 1:
19             tempFunctionData = evaluate_expr(expr.children[1], environ)
20             FunctionData[0] = FunctionData[0] + tempFunctionData[0]
21             FunctionData[1] = FunctionData[1] + tempFunctionData[1]
22             FunctionData[2] = FunctionData[2] + tempFunctionData[2]
23
24         return FunctionData
25
26     elif expr.type == ntLetRecAssignment:
27         arglist_body = evaluate_expr(expr.children[0], environ)
28         return [[expr.leaf], arglist_body[0], arglist_body[1]]
29
30     elif expr.type == ntRecFuncDecl:
31         if (len(expr.children) == 2):
32             arglist = evaluate_expr(expr.children[0], environ)
33             body = [expr.children[1]]
34         else:
35             arglist = []
36             body = [expr.children[0]]
37         return [[arglist], body]

```

Conceptual Exercises for Section 11.3

Exercise 11.3.1 Even though the `make-closure` function is called in the definition of the `extend-environment-recursively` for the closure representation of a recursive environment, the closure is still created every time the name of the recursive function is looked up in the environment. Explain.

Exercise 11.3.2 Can a `let*` expression evaluated using *dynamic scoping* achieve the same result (i.e., recursion) as a `letrec` expression evaluated using *lexical scoping*? In other words, does a `let*` expression evaluated using dynamic scoping simulate a `letrec` expression? Explain.

Programming Exercise	Camille	Description	Start from	Representation of Closures	Representation of Environment
11.3.6	2.1(nameless ASR)	<code>letrec</code> nameless environment	2.0(nameless ASR) or 2.1(named ASR)	ASR CLS	ASR
11.3.7	2.1(nameless LOLR)	<code>letrec</code> nameless environment	2.0(nameless LOLR) or 2.1(named LOVR)	ASR CLS	LOLR
11.3.8	2.1(nameless CLS)	<code>letrec</code> nameless environment	2.0(nameless CLS) or 2.1(named CLS)	ASR CLS	CLS
11.3.9	2.1(dynamic scoping)	<code>letrec</code> dynamic scoping	2.0(dynamic scoping) or 2.1	lambda expression	CLS ASR LOLR

Table 11.2 New Versions of Camille, and Their Essential Properties, Created in the Section 11.3.3 Programming Exercises. (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

Programming Exercises for Section 11.3

Table 11.2 summarizes the properties of the new versions of the Camille interpreter developed in the following programming exercises. Figure 11.9 presents the dependencies between the non-recursive and recursive versions of Camille developed thus far, including in these programming exercises.

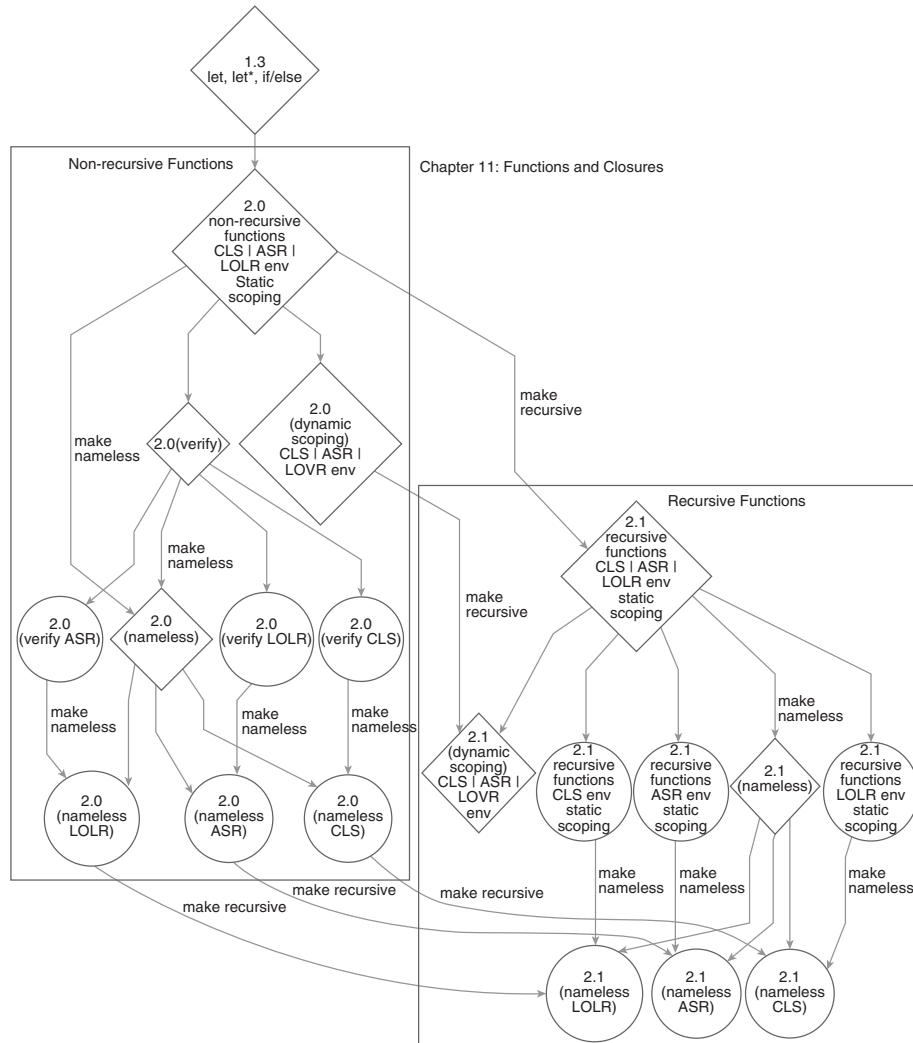
Exercise 11.3.3 Build an abstract-syntax representation of a nameless, recursive environment (Figure 11.10). Complete Programming Exercise 9.8.9, but this time make the abstract-syntax representation of the nameless environment recursive.

Exercise 11.3.4 (Friedman, Wand, and Haynes 2001, Exercise 3.34, p. 95) Build a list-of-lists representation of a nameless, recursive environment (Figure 11.11). Complete Programming Exercise 9.8.5.b or 11.2.9.b, but this time make the list-of-lists representation of the nameless environment recursive.

Exercise 11.3.5 Build a closure representation of a nameless, recursive environment. Complete Programming Exercise 9.8.7, but this time make the closure representation of the nameless environment recursive.

Exercise 11.3.6 (Friedman, Wand, and Haynes 2001) Augment the solution to Programming Exercise 11.2.10 with `letrec`. In other words, extend Camille 2.0(nameless ASR) with `letrec`. Alternatively, modify Camille 2.1(named ASR) to use a nameless environment. Reuse the abstract-syntax representation of a recursive, nameless environment built in Programming Exercise 11.3.3.

Exercise 11.3.7 (Friedman, Wand, and Haynes 2001, Exercise 3.34, p. 95) Augment the solution to Programming Exercise 11.2.9 with `letrec`. In other words, extend Camille 2.0(nameless LOLR) with `letrec`. Alternatively, modify Camille 2.1(named LOLR) to use a nameless environment. Reuse the list-of-lists



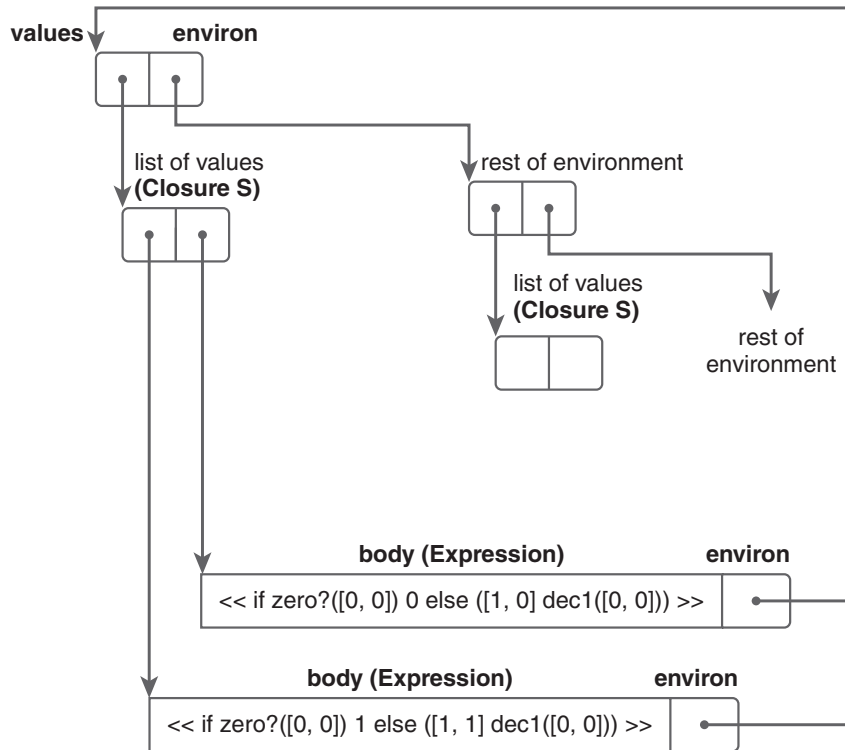


Figure 11.10 An abstract-syntax representation of a circular, recursive, nameless environment using the structure of Programming Exercise 11.3.3.

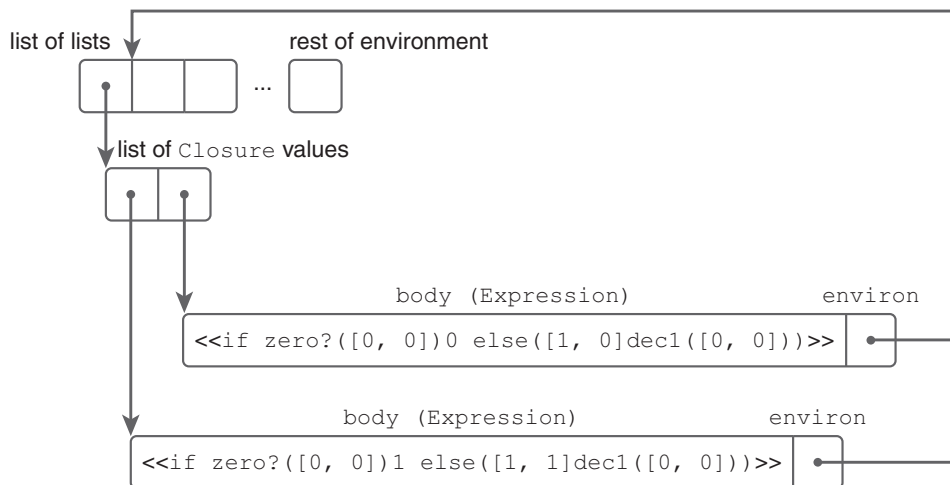


Figure 11.11 A list-of-lists representation of a circular, recursive, nameless environment using the structure of Programming Exercise 11.3.4.

to use a nameless environment. Reuse the closure representation of a recursive, nameless environment built in Programming Exercise 11.3.5.

Exercise 11.3.9 Modify the Camille interpreter defined in this section to use *dynamic scoping* to bind references to declarations. For instance, in the recursive Camille function `pow` shown here the reference to the identifier `s` in the expression `*(s, (pow -(t,1)))` in line 5 is bound to 3, not 2; thus, the return value of the call to `(pow 2)` on line 10 is 9 (under dynamic scoping), not 4 (under static/lexical scoping).

Example:

```

1  Camille> let
2              s = 2
3              in
4              letrec
5                  pow = fun(t) if zero?(t) 1 else *(s, (pow -(t,1)))
6                  in
7                      let
8                          s = 3
9                          in
10                             (pow 2)
11
12  9

```

11.4 Thematic Takeaways

- The interplay of evaluating expressions in an environment and applying functions to arguments is integral to the operation of an interpreter:

```

apply_closure (make_closure(arglist, body, environ), arguments) =
evaluate_expr(body, extend_environment(parameters, arguments, environ))

```

- Non-recursive and recursive, user-defined functions are implemented manually in Camille, with the implementation of a closure ADT.
- We can alter (sometimes drastically) the semantics of the language defined by an interpreter (e.g., from static to dynamic scoping, or from deep to shallow to ad hoc binding) by changing as little as one or two lines of code of the interpreter. This typically involves just changing how and when we pass the environment.
- The careful design of ADTs through interfaces renders the Camille interpreter malleable and flexible. For instance, we can switch the representation of the environment or closures without breaking the Camille interpreters as long as these representations remain faithful to the interface. The Camille interpreters do not rely on particular representations for the supporting ADTs.

- Identifiers as references in computer programs are superfluous to the operation of an interpreter and need not be represented in the abstract-syntax tree produced by a parser and processed by an interpreter; only lexical depth and position are necessary.
- “The interpreter for a computer language is just another [computer] program” (Friedman, Wand, and Haynes 2001, Foreword, p. vii, Hal Abelson).

11.5 Chapter Summary

In this chapter, we implemented non-recursive and recursive, user-defined functions in Camille. In Camille, functions are represented as closures. We built three representations for the closure data type: an abstract-syntax representation (ASR), a closure representation (CLS), and a Python closure representation (i.e., `lambda` expressions in Python; Programming Exercise 11.2.12). When a function is invoked, we pass the values to be bound to the arguments of the function to the closure representing the function. For the ASR and CLS representations of a closure, a pointer to the environment in which the function is defined is stored in the closure (i.e., lexical scoping). For the Python closure representation (i.e., `lambda` expressions in Python), a pointer to the environment in which the function is called is stored in the closure. We continue to see that identifiers as references are superfluous in the abstract-syntax tree processed by an interpreter; only lexical depth and position are necessary. Thus, we developed both named and nameless non-recursive environments, and named and nameless recursive environments (Table 11.3) and interpreters using these environments (Table 11.4). Moreover, we continue to see that deep binding is not lexical scoping and that shallow binding is not dynamic scoping. Deep, shallow, and ad hoc binding are only applicable in languages with first-class functions (e.g., Scheme, Camille).

Figure 11.12 and Table 11.5 present the dependencies between the versions of Camille we have developed. Table 11.6 summarizes the versions of the Camille interpreter we have developed. Note that if closures in Camille are represented as Python closures in version 2.0 of the Camille interpreter, then the (Non-recursive Functions, 2.0) cell in Table 11.6 must contain “`↓ lambda ↓.`” Similarly, if closures in Camille are represented as Python closures in version 2.1 of the Camille interpreter, then the (Recursive Functions, 2.1) cell must contain “`↓ lambda ↓.`”

Table 11.7 outlines the configuration options available in Camille for aspects of both the design of the interpreter (e.g., choice of representation of referencing environment) and the semantics of implemented concepts (e.g., choice of scoping method). As we vary the latter, we get a different version of the language (Table 11.6). Note that the nameless environment is not available for use with the interpreter supporting dynamic scoping.

	Named	Nameless
Non-recursive	CLS (Section 9.8.3) ASR (Section 9.8.4; Figure 11.2) LOLR (Figure 11.3; PE 9.8.5.a)	CLS (PE 9.8.7) ASR (Figure 11.6; PE 9.8.9) LOLR (Figure 11.5; PE 9.8.5.b/11.2.9.b)
Recursive	CLS (Section 11.3.2) ASR (Section 11.3.2; Figure 11.7) LOLR (Section 11.3.2; Figure 11.8)	CLS (PE 11.3.5) ASR (Figure 11.10; PE 11.3.3) LOLR (Figure 11.11; PE 11.3.4)

Table 11.3 Variety of Environments in Python Developed in This Text (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation; PE = programming exercise.)

	Named	Nameless
Non-recursive	CLS (Section 11.2) ASR (Section 11.2) LOLR (Section 11.2)	CLS (PE 11.2.11) ASR (PE 11.2.10/2.0(nameless ASR)) LOLR (PE 11.2.9/2.0(nameless LOLR))
Recursive	CLS (Section 11.3) ASR (Section 11.3) LOLR (Section 11.3)	CLS (PE 11.3.8) ASR (PE 11.3.6/2.1(nameless ASR)) LOLR (PE 11.3.7/2.1(nameless LOLR))

Table 11.4 Camille Interpreters in Python Developed in This Text Using All Combinations of Non-recursive and Recursive Functions, and Named and Nameless Environments. All interpreters identified in this table work with both the CLS and ASR of closures (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation; PE = programming exercise.)

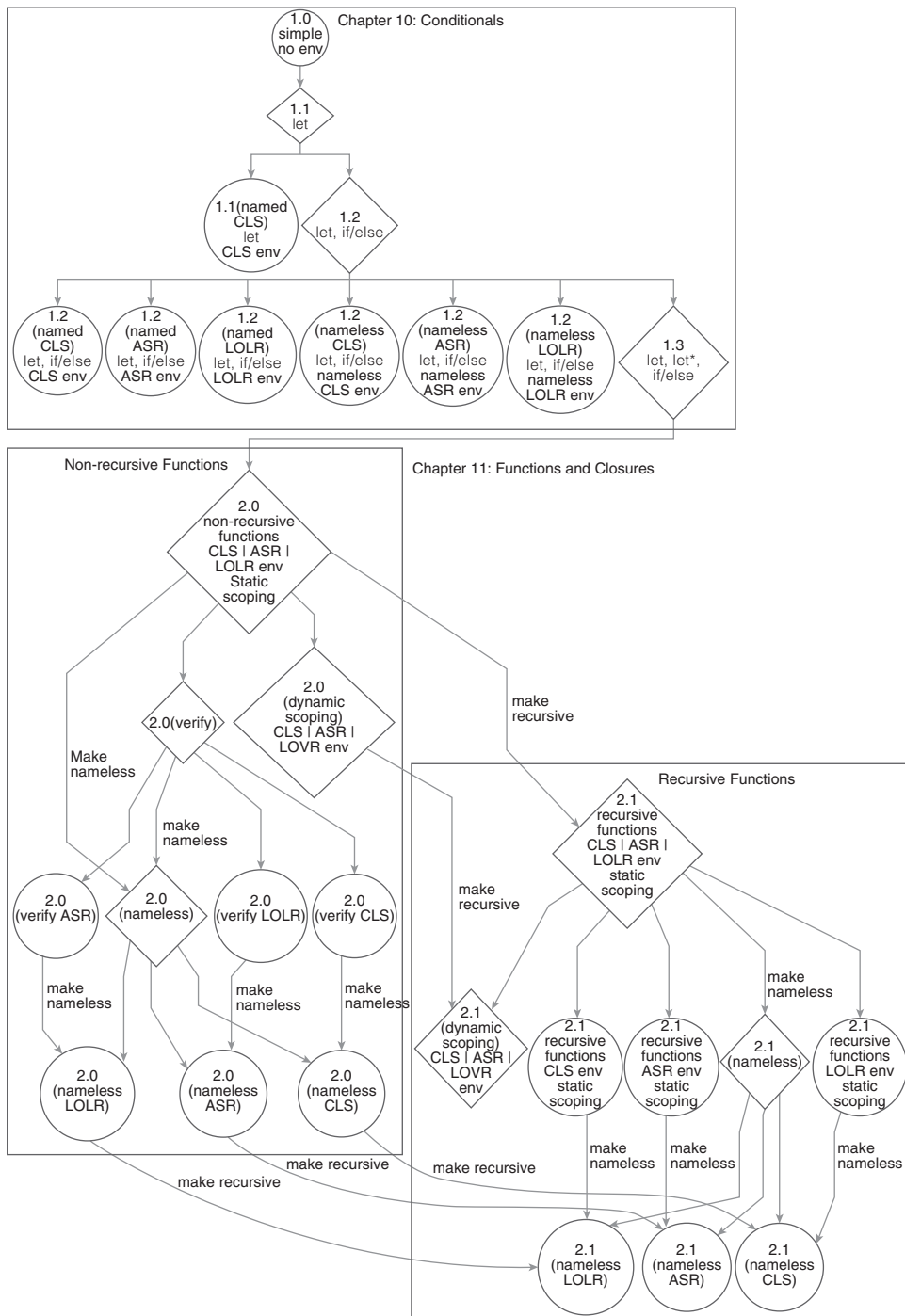


Figure 11.12 Dependencies between the Camille interpreters developed thus far, including those in the programming exercises. The semantics of a directed edge $a \rightarrow b$ are that version b of the Camille interpreter is an extension of version a (i.e., version b subsumes version a). (Key: circle = instantiated interpreter; diamond = abstract interpreter; ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

Version	Extends	Description
Chapter 10: Local Binding and Conditional Evaluation		
1.0	N/A	simple, no environment
1.1	1.0	let, named CLS ASR LOLR environment
1.1(named CLS)	1.1	let, named CLS environment
1.2	1.1	let, if
1.2(named CLS)	1.2	let, if/else, named CLS environment
1.2(named ASR)	1.2	let, if/else, named ASR environment
1.2(named LOLR)	1.2	let, if/else, named LOLR environment
1.2(nameless CLS)	1.2	let, if/else, nameless CLS environment
1.2(nameless ASR)	1.2	let, if/else, nameless ASR environment
1.2(nameless LOLR)	1.2	let, if/else, nameless LOLR environment
1.3	1.2	let*, if/else, (named nameless) (CLS ASR LOLR) environment
Chapter 11: Functions and Closures		
Non-recursive Functions		
2.0	1.2	fun, CLS ASR LOLR environment
2.0(verify ASR)	2.0	fun, verify ASR environment
2.0(nameless ASR)	2.0(verify ASR)	fun, nameless ASR environment
2.0(verify LOLR)	2.0	fun, verify LOLR environment
2.0(nameless LOLR)	2.0(verify LOLR)	fun, nameless LOLR environment
2.0(verify CLS)	2.0	fun, verify CLS environment
2.0(nameless CLS)	2.0(verify CLS)	fun, nameless CLS environment
2.0(dynamic scoping)	2.0	fun, dynamic scoping, (named nameless) (CLS ASR LOLR) environment
Recursive Functions		
2.1	2.0	letrec, named CLS ASR LOLR environment
2.1(named CLS)	2.0	letrec, named CLS environment
2.1(nameless CLS)	2.0(nameless CLS) or 2.1(named CLS)	letrec, nameless CLS environment
2.1(named ASR)	2.0	letrec, named ASR environment
2.1(nameless ASR)	2.0(nameless ASR) or 2.1(named ASR)	letrec, nameless ASR environment
2.1(named LOLR)	2.0	letrec, named LOLR environment
2.1(nameless LOLR)	2.0(nameless LOLR) or 2.1(named LOLR)	letrec, nameless LOLR environment
2.1(dynamic scoping)	2.0(dynamic scoping) or 2.1	letrec, dynamic scoping, (named nameless) (CLS ASR LOLR) environment

Table 11.5 Versions of Camille (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

	Version of Camille					
Concepts / Data Structures	Expressed Values	integers	integers	integers	integers	integers $\cup$ cls
	Denoted Values	integers	integers	integers	integers	integers $\cup$ cls
	Representation of Environment	N/A	ASR CLS LOLR	ASR CLS LOLR	ASR CLS LOLR	ASR CLS LOLR
	Representation of Closures	N/A	N/A	N/A	N/A	ASR CLS
	Local Binding	$\times$	$\uparrow$ let $\uparrow$	$\uparrow$ let $\uparrow$	$\uparrow$ let, let* $\uparrow$	$\uparrow$ let, let* $\uparrow$
	Conditionals	$\times$	$\times$	$\downarrow$ if/else $\downarrow$	$\downarrow$ if/else $\downarrow$	$\downarrow$ if/else $\downarrow$
	Non-recursive Functions	$\times$	$\times$	$\times$	$\times$	$\uparrow$ fun $\uparrow$
	Recursive Functions	$\times$	$\times$	$\times$	$\times$	$\uparrow$ let rec $\uparrow$
	Scoping	N/A	lexical	lexical	lexical	lexical
	Environment Binding to Closure	N/A	N/A	N/A	N/A	deep
	Parameter Passing	N/A	N/A	N/A	N/A	$\uparrow$ by value $\uparrow$

Table 11.6 Concepts and Features Implemented in Progressive Versions of Camille. The symbol $\downarrow$ indicates that the concept is supported through its implementation in the defining language (here, Python). Python keyword included in each cell, where applicable, indicates which Python construct is used to implement the feature in Camille. The symbol $\uparrow$ indicates that the concept is implemented manually. The Camille keyword included in each cell, where applicable, indicates the syntactic construct through which the concept is operationalized. (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation. Cells in boldface font highlight the enhancements across the versions.)

Interpreter Design Options			Language Semantic Options		
Type of Environment	Representation of Environment	Representation of Functions	Scoping Method	Environment Binding	Parameter-Passing Mechanism
named	abstract syntax	abstract syntax	static	deep	by value
nameless	list of lists closure	closure	dynamic		

Table 11.7 Configuration Options in Camille

11.6 Notes and Further Reading

For a book focused on the implementation of functional programming languages, we refer readers to Peyton Jones (1987).

Chapter 12

Parameter Passing

Lazy evaluation is perhaps the most powerful tool for modularization in the functional programmer's repertoire.

— John Hughes in “Why Functional Programming Matters” (1989)

W^E study a variety of *parameter-passing mechanisms* in this chapter. Concomitantly, we add support for a subset of them to Camille, including *pass-by-reference* and *lazy evaluation*. In addition, we reflect on the design decisions we have made and techniques we have used throughout the interpreter implementation process and discuss alternatives.

12.1 Chapter Objectives

- Explore a variety of parameter-passing mechanisms, including *pass-by-value* and *pass-by-reference*.
- Describe *lazy evaluation* (i.e., pass-by-name and pass-by-need) and its implications on programs.
- Discuss the implementation of pass-by-reference and lazy evaluation.

12.2 Assignment Statement

To support an assignment statement in Camille, we add the following rules to the grammar and corresponding pattern-action rules to the PLY parser generator:

ntAssignment

`<expression> ::= assign! <identifier> = <expression>`

```
def p_expression_assign(t):  
    '''expression : ASSIGN IDENTIFIER EQ expression'''  
    t[0] = Tree_Node(ntAssignment, [t[4]], t[2], t.lineno(1))
```

It is helpful to draw a distinction between binding and variable assignment. A *binding* associates a *name* with a *value*. A *variable assignment*, in contrast, is a mutation of the expressed value stored in a memory cell. For instance, an identifier x can be associated with a *reference*, where a reference is an expressed value containing or referring to another expressed value 1 . Mutating the value that the reference contains or to which the reference refers from 1 to 2 does not alter the binding of x to the reference (i.e., x is still bound to the same reference). A reference is called an *L-value* and an expressed value is known as an *R-value*—based on the side of the assignment statement in which each appears.

Variable assignment is helpful for a variety of purposes. For instance, two or more functions can communicate with each other through a shared “global” variable rather than by passing the variable back and forth to each other. This use of variables can reduce the number of parameters that need to be passed in a program. Of course, the use of variable assignment involves side effect, so there is a trade-off between data protection and the overhead of parameter-passing. However, we can use closures to protect that shared state from any unintended outside interference:

```
Camille> let --- hidden state through a lexical closure
          new_counter = fun()
            let
              i = 0
            in
              fun()
                let --- i++;
                  ignored = assign! i = incl(i)
                in
                  i
            in
              let
                counter = (new_counter)
              in
                let
                  ignored1 = (counter)
                  ignored2 = (counter)
                  ignored3 = (counter)
                in
                  (counter)
```

4

Here, the variable i is a private variable representing a counter. The identifier `counter` is bound to a Camille closure. In consequence, it *remembers* values in its lexical parent—here, i —even though the lifetime of that parent has expired (i.e., been popped off the stack).

12.2.1 Use of Nested `lets` to Simulate Sequential Evaluation

Since we do not yet have support for sequential evaluation or statement blocks in Camille (we add it in Section 12.7), we use nested `lets` to simulate sequential evaluation as demonstrated in the following example. The hypothetical Camille expression

```

let
  a = 1
  b = 2
in
  { ignored = assign! a = incl(a); --- a++;
    ignored2 = assign! b = incl(b); --- b++;
    +(a,b) }

```

can be rewritten as an actual Camille expression:

```

Camille> let --- nested lets simulate sequential evaluation
  a = 1
  b = 2
in
  let
    ignored = assign! a = incl(a)
  in
    let
      ignored = assign! b = incl(b)
    in
      +(a,b)

5

```

The identifier `ignored` receives the return value of the two assignment statements. The return value of the assignment statement in C and C++ is the value of the expression on the right-hand side of the assignment operator.

12.2.2 Illustration of Pass-by-Value in Camille

We will modify the Camille interpreter so that parameters to functions are represented as references in the environment. We start by creating a new reference for each parameter in each function call—a parameter-passing mechanism called *pass-by-value*. As a result of the use of this new reference, modifications to the parameter within the body of the function will have no effect on the value of the parameter in the environment in which the parameter was passed as an argument; in other words, assignments will only have “local” effect. For instance, consider the following Camille program:

```

Camille> let ---- pass-by-value with copy of reference for parameter x
  n = 1
in
  let
    increment = fun(x) assign! x = incl(x) --- x++;
  in
    let
      ignored = (increment n)
    in
      n --- returns 1 not 2

1

```

Here, a copy of `n` is passed to and incremented by the function `increment`, so the value of the `n` in the outermost `let` expression is not modified. Similarly, consider a `swap` function in Camille:

```

Camille> let --- swap function: pass-by-value
          a = 3
          b = 4

          swap = fun(x,y)
                  let
                    temp = x
                  in
                    let
                      ignored1 = assign! x = y
                    in
                      assign! y = temp
          in
            let
              ignored2 = (swap a,b)
            in
              -(a, b) --- returns -1, not 1

-1

```

Here, the values of *a* and *b* are not swapped because both are passed to the *swap* function *by value*.

12.2.3 Reference Data Type

To support an assignment statement in Camille, we must add a *Reference* data type, with interface *dereference* and *assignreference* to the interpreter. We use the familiar list-of-values (used in the list-of-lists, ribcage and abstract-syntax representations of an environment) for each rib (Friedman, Wand, and Haynes 2001). References are elements of *lists*, which are assignable using the assignment operator in Python. Again, note that lists in Python are used and accessed as if they were *vectors* rather than lists in Scheme, ML, or Haskell. In particular, unlike lists used in functional programming, the individual elements of lists in Python can be directly accessed through an integer index in constant time. Figure 12.1 depicts an instance of this *Reference* data type in relation to the underlying Python list used in its implementation. The following is an abstract-syntax implementation of a reference data type:

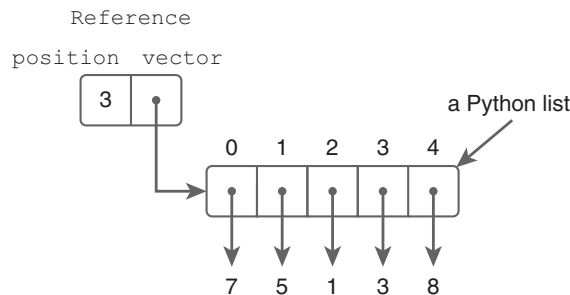


Figure 12.1 A primitive reference to an element in a Python list.

Data from Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 2001. *Essentials of Programming Languages*. 2nd ed. Cambridge, MA: MIT Press.

```

class Reference:
    def __init__(self, position, vector):
        self.position = position
        self.vector = vector

    def primitive_dereference(self):
        return self.vector[self.position]

    def primitive_assignreference(self, value):
        self.vector[self.position] = value

    def dereference(self):
        try:
            return self.primitive_dereference()
        except:
            raise Exception("Illegal dereference.")

    def assignreference(self, value):
        try:
            self.primitive_assignreference(value)
        except:
            raise Exception("Illegal creation of reference.")

```

The function `dereference` here is the analog of the `*` (dereferencing) operator in C/C++ (e.g., `*x`) when preceding a variable reference. However, unlike in C/C++, dereferencing is *implicit* in Camille, akin to referencing Scheme or Java objects. Thus, the function `dereference` is called within the Camille interpreter, but not directly by Camille programmers. In Scheme:

expressed value	=	any possible Scheme value
denoted value	=	reference to any possible Scheme value

so that

$$\text{denoted value} \neq \text{expressed value}$$

Scheme exclusively uses references as denoted values in the sense that *all* denoted values are references in Scheme.

In Java:

expressed value	=	reference to object $\cup$ primitive value
denoted value	=	reference to object $\cup$ primitive value

so that

$$\text{denoted value} = \text{expressed value}$$

Java is slightly less consistent than Scheme in the use of references: all denoted values in Java, save for primitive values, are references. While all denoted values in Scheme are references, it appears to the Scheme programmer as if all denoted values are the same as expressed values because Scheme uses automatic or implicit dereferencing. Similarly, while all denoted values, save for primitives, are references in Java, it appears to the Java programmer as if all denoted values are the same as expressed values because Java also uses implicit referencing.

The functions `dereference` and `assignreference` are defined through `primitive_dereference` and `primitive_assignreference` because later we will reuse the latter two functions in implementations of references.

12.2.4 Environment Revisited

Now that we have a `Reference` data type, we must modify the environment implementation so that it can make use of references. We assume that denoted values in an environment are of the form `Ref(x)` for some `x`. We realize this environment structure by adding the function `apply_environment_reference` to the environment interface. This function is similar to `apply_environment`, except that when it finds the matching identifier, it returns the “reference to its value” instead of its value (Friedman, Wand, and Haynes 2001). Therefore, as in Scheme, all denoted values in Camille are references:

$$\begin{aligned}\text{expressed value} &= \text{integer} \cup \text{closure} \\ \text{denoted value} &= \text{reference to an expressed value}\end{aligned}$$

Thus,

$$\text{denoted value} \neq \text{expressed value} (= \text{integer} \cup \text{closure})$$

The function `apply_environment` then can be defined through the `apply_environment_reference` and `dereference` (Friedman, Wand, and Haynes 2001) functions:

```
def apply_environment(environ, identifier):
    return apply_environment_reference(environ, identifier).dereference()

def apply_environment_reference(environ, identifier):
    if environ.flag == "empty-environment-record":
        raise IndexError
    elif environ.flag == "extended-environment-record":
        try:
            return Reference(environ.symbols.index(identifier),
                             environ.values)
        except:
            return apply_environment_reference(environ.environ, identifier)
    elif environ.flag == "recursively-extended-environment-record":
        try:
            position = environ.fun_names.index(identifier)

            # pass-by-value
            return Reference(0,
                             [make_closure(environ.parameterlists[position],
                                             environ.bodies[position], environ)])
        except:
            return apply_environment_reference(environ.environ, identifier)
```

Notice that we are using an abstract-syntax representation (ASR) of a named environment here. To complete the implementation of variable assignment, we add the following case to the `evaluate_expr` function:

```
elif expr.type == ntAssignment:
    tempref = apply_environment_reference(environ, expr.leaf)

    tempref.assignreference(evaluate_expr(expr.children[0], environ))

    # ignored return value of assignment
    return 1
```

Notice that a value is returned. Here, we explicitly return the integer 1 (as seen in the last line of code) because the return value of the function `assignreference` is unspecified and we must always return an expressed value. When using assignment statements in a variety of programming languages, the return value can be ignored (e.g., `x--`; in C). In Camille, the return value of an assignment statement is ignored, especially when a series of assignment statements are used within a series of `let` expressions to simulate sequential execution, as illustrated in this section.

12.2.5 Stack Object Revisited

Consider the following enhancement, using references, of a simple stack object in Camille as presented in Section 11.2.4:

```

1  let
2    new_stack = fun ()
3      let *
4        empty_stack = fun (msg)
5          if eqv?(msg,1)
6            200 --- cannot top an empty stack
7          else
8            if eqv?(msg,2)
9              100 --- cannot pop an empty stack
10           else if eqv?(msg,3)
11             1 --- represents true: stack is empty
12           else
13             300 --- not a valid message
14         stack_data = empty_stack
15         prior_stack_data = empty_stack
16     in
17       let
18         --- constructor
19         push = fun (item)
20           let
21             ignore = assign! prior_stack_data = stack_data
22           in
23             assign! stack_data =
24               fun (msg)
25                 if eqv?(msg,1)
26                   item
27                 else
28                   if eqv?(msg,2)
29                     assign! stack_data = prior_stack_data
30                   else if eqv?(msg,3)
31                     0 --- represents false:
32                     --- stack is not empty
33                   else
34                     300 --- not a valid message
35         --- observers
36         empty? = fun () (stack_data 3)
37         top = fun () (stack_data 1)
38         pop = fun () (stack_data 2)
39         reset = fun () assign! stack_data = empty_stack
40     in
41       let
42         --- collection_of_functions uses
43         --- a closure to simulate an array

```



```

44         collection_of_functions = fun(i)
45         if eqv?(i,3)
46             empty?
47         else
48             if eqv?(i,1)
49                 top
50             else
51                 if eqv?(i,2)
52                     pop
53                 else
54                     if eqv?(i,4)
55                         push
56                     else if eqv?(i,5)
57                         reset
58                     else
59                         400
60
61         in
62             collection_of_functions
63         get_empty?_method = fun(stk) (stk 3)
64         get_push_method = fun(stk) (stk 4)
65         get_top_method = fun(stk) (stk 1)
66         get_pop_method = fun(stk) (stk 2)
67         get_reset_method = fun(stk) (stk 5)
68     in
69         let
70             s1 = (new_stack)
71             s2 = (new_stack)
72         in
73             let
74                 empty1? = (get_empty?_method s1)
75                 push1 = (get_push_method s1)
76                 top1 = (get_top_method s1)
77                 pop1 = (get_pop_method s1)
78                 reset1 = (get_reset_method s1)
79                 empty2? = (get_empty?_method s2)
80                 push2 = (get_push_method s2)
81                 top2 = (get_top_method s2)
82                 pop2 = (get_pop_method s2)
83                 reset2 = (get_reset_method s2)
84             in
85                 --- main program
86                 let*
87                     t1 = (push1 15)
88                     t2 = (push1 16)
89                     t3 = (push2 inc1((top1)))
90                     t4 = (push2 31)
91                 in
92                     if eqv?((top2),0)
93                         (top1)
94                     else
95                         let
96                             d = (pop2)
97                         in
98                             (top2)

```

In this version of the stack object, the stack is a true object because its methods are encapsulated within it. Notice that the `let` expression on lines 41–61 builds and returns a closure that *simulates* an array (of stack functions): It accepts an index `i` as an argument and returns the stack function located at that index.

Programming Exercise	Camille	Description	Start from	Representation of Closures	Representation of Environment
12.2.3	3.0(cells)	cells	3.0	ASR CLS	ASR
12.2.4	3.0(arrays)	arrays	3.0	ASR CLS	ASR

Table 12.1 New Versions of Camille, and Their Essential Properties, Created in the Programming Exercises of This Section (Key: ASR = abstract-syntax representation; CLS = closure.)

Table 12.1 summarizes the properties of the new versions of the Camille interpreter developed in the programming exercises in this section.

Conceptual and Programming Exercises for Section 12.2

Exercise 12.2.1 In the version of Camille developed in this section, we stated that denoted values are *references* to expressed values. Does this mean that references to expressed values are stored in the environment of the Camille interpreter developed in this section? Explain.

Exercise 12.2.2 Write a Camille program that defines the mutually recursive functions `iseven?` and `isodd?` (i.e., each function invokes the other). Neither of these functions accepts any arguments. Instead, they communicate with each other by changing the state of a shared “global” variable `n` that represents the number being checked. The functions should each decrement the variable `n` throughout the lifetime of the program until it reaches 0—the base case. Thus, the functions `iseven?` and `isodd?` communicate by side effect rather than by returning values.

Exercise 12.2.3 (Friedman, Wand, and Haynes 2001, Exercise 3.41, p. 103) In Scheme and Java, everything is a reference (except for primitives in Java), although both languages use implicit (pointer) dereferencing. Thus, it may appear as if no denoted value represents a reference in these languages. In contrast, C has reference (e.g., `int* intptr;`) and non-reference (e.g., `int x;`) types and uses explicit (pointer) dereferencing (e.g., `*x`). Thus, an alternative scheme for variable assignment in Camille is to have references be expressed values, and have allocation, dereferencing, and assignment operators be explicitly used by the programmer (as in C):

expressed value = integer $\cup$ closure $\cup$ reference to an expressed value
denoted value = expressed value

Modify the Camille interpreter of this section to implement this alternative design, with the following new primitives:

- `cell`: creates a reference
- `contents`: dereferences a reference
- `assigncell`: assigns a reference

In this version of Camille, the counter program at the beginning of Section 12.2 is rendered as follows:

```
let
  g = let
    count = cell(0)
  in
    fun()
      let
        ignored = assigncell(count, incl(contents(count)))
      in
        contents(count)
  in
    +((g), (g))
```

Exercise 12.2.4 (Friedman, Wand, and Haynes 2001, Exercise 3.42, p. 105) Add support for arrays to Camille. Modify the Camille interpreter presented in this section to implement arrays. Use the following interface for arrays:

- `array`: creates an array
- `arrayreference`: dereferences an array
- `arrayassign`: updates an array

Thus,

<code>array</code>	=	a list of zero or more references to expressed values
<code>expressed value</code>	=	<code>integer</code> $\cup$ <code>closure</code> $\cup$ <code>array</code>
<code>denoted value</code>	=	reference to an expressed value

Note that the first occurrence of “reference” (on the right-hand side of the equal sign in the first equality expression) can be a different implementation of references than that described in this section. For example, a Python list is already a sequence of references.

What is the result of the following Camille program?

```
let
  a = array(2) --- allocates a two-element array
  p = fun(x)
    let
      v = arrayreference(x, 1)
    in
      arrayassign(x, 1, incl(v))
  in
    let
```

```

        ignored = arrayassign(a, 1, 0)
in
    let
        ignored = (p a)
in
    let
        ignored = (p a)
in
    arrayreference(a, 1)

```

Exercise 12.2.5 Rewrite the Camille stack object program in Section 12.2.5 so that it uses arrays. Specifically, eliminate the closure that *simulates* an array (of stack functions) built and returned through the `let` expression on lines 41–60 and use an array instead to store the collection of stack functions. Use the array-creation and -manipulation interface presented in Programming Exercise 12.2.4.

12.3 Survey of Parameter-Passing Mechanisms

We start by surveying parameter-passing mechanisms in a variety of languages prior to discussing implementation strategies for these mechanisms.

12.3.1 Pass-by-Value

Pass-by-value is a parameter-passing mechanism in which copies of the arguments are passed to the function. For this reason, pass-by-value is sometimes referred to as *pass-by-copy*. Consider the classical `swap` function in C:

```

$ cat swap_pbv.c
#include <stdio.h>

/* swap pass-by-value */
void swap (int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    printf("In swap: ");
    printf("a = %d, b = %d.\n", a, b);
}

int main() {
    int x = 3;
    int y = 4;

    printf("In main, before call to swap: ");
    printf("x = %d, y = %d.\n", x, y);

    swap (x, y);

    printf("In main, after call to swap: ");
    printf("x = %d, y = %d.\n", x, y);
}

$ gcc swap_pbv.c
$ ./a.out

```

In main, before call to swap: $x = 3, y = 4$.
 In swap: $a = 4, b = 3$.
 In main, after call to swap: $x = 3, y = 4$.

C only passes arguments by value (i.e., by copy). Figure 12.2 shows the run-time stack of this swap function with signature **void** swap(**int** a, **int** b):

1. (top left) Before swap is called.
2. (top right) After swap is called. Notice that *copies* of x and y are passed in.
3. (bottom left) While swap executes. Notice that the swap takes place within the activation record of the swap function, not main.
4. (bottom right) After swap returns.

As can be seen, the function does not swap the two integers.

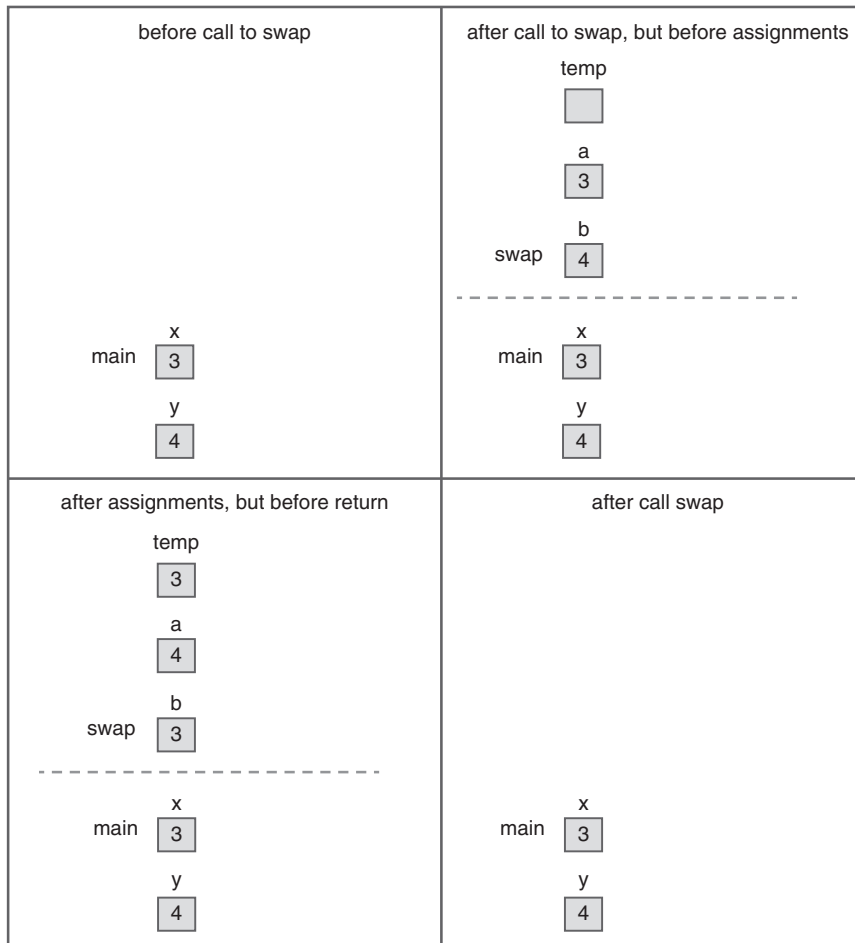


Figure 12.2 Passing arguments by value in C. The run-time stack grows upward. (Key: $\square$ = memory cell; $\cdots$ = activation-record boundary.)

Java also only passes arguments by value. Consider the following swap method in Java, which accepts integer primitives as arguments:

```
class NoSwapPrimitive {  
    private static void swap(int a, int b) {  
        int temp = a;  
  
        a = b;  
        b = temp;  
  
        System.err.print("In swap: ");  
        System.err.print("a = " + a + ", ");  
        System.err.println("b = " + b + ".");  
    }  
  
    public static void main(String args[]) {  
        int x = 3;  
        int y = 4;  
  
        System.err.print("In main, before call to swap: ");  
        System.err.print("x = " + x + ", ");  
        System.err.println("y = " + y + ".");  
  
        NoSwapPrimitive.swap(x, y);  
  
        System.err.print("In main, after call to swap: ");  
        System.err.print("x = " + x + ", ");  
        System.err.println("y = " + y + ".");  
    }  
}
```

The output of this program is

```
$ javac NoSwapPrimitive.java  
$ java NoSwapPrimitive  
In main, before call to swap: x = 3, y = 4.  
In swap: x = 4, y = 3.  
In main, after call to swap: x = 3, y = 4.
```

The status of the run-time stack in Figure 12.2 applies to this Java swap method with signature **void** swap(**int** a, **int** b) as well. Since all parameters, including primitives, are passed by value in Java, this swap method does not swap the two integers. Consider the following version of the swap program in Java, where the arguments to the swap method are references to objects instead of primitives:

```
class NoSwapObject {  
    private static void swap(Integer a, Integer b) {  
        Integer temp = a;  
  
        a = b;  
        b = temp;  
  
        System.err.print("In swap: ");
```

```

        System.err.print("a = " + Integer.valueOf(a) + ", ");
        System.err.println("b = " + Integer.valueOf(b) + ".");
    }

    public static void main(String args[]) {

        Integer x = Integer.valueOf(3);
        Integer y = Integer.valueOf(4);

        System.err.print("In main, before call to swap: ");
        System.err.print("x = " + Integer.valueOf(x) + ", ");
        System.err.println("y = " + Integer.valueOf(y) + ".");

        NoSwapObject.swap(x, y);

        System.err.print("In main, after call to swap: ");
        System.err.print("x = " + Integer.valueOf(x) + ", ");
        System.err.println("y = " + Integer.valueOf(y) + ".");
    }
}

```

The output of this program is

```

$ javac NoSwapObject.java
$ java NoSwapObject
In main, before call to swap: x = 3, y = 4.
In swap: x = 4, y = 3.
In main, after call to swap: x = 3, y = 4.

```

Figure 12.3 illustrates the run-time stack during the execution of this Java swap method with signature **void** swap(Integer a, Integer b):

1. (top left) Before swap is called. Notice the denoted values of x and y are *references to objects*.
2. (top right) After swap is called. Notice that *copies* of the *references* x and y are passed in.
3. (bottom left) While swap executes. Notice that the *references* are swapped rather than the objects to which they point. As before, the swap takes place within the activation record of the swap method, not main.
4. (bottom right) After swap returns.

As can be seen, this swap method does not swap its Integer object-reference arguments. The references to the objects in main are not swapped because “Java manipulates objects ‘by reference,’ but it passes object references to methods ‘by value’” (Flanagan 2005). Consequently, a swap method intended to swap primitives or references to objects cannot be defined in Java.

Scheme also only supports passing arguments by value. Thus, as in Java, references in Scheme are passed by value. However, unlike in Java, all denoted values are references to expressed values in Scheme. Consider the following Scheme program:

```

(define swap
  (lambda (a b)
    (let ((temp a))      ; temp = a

```

```

(begin
  (set! a b)      ; a = b
  (set! b temp)   ; b = temp
  (display "In swap: a=")
  (display a)
  (display ", b=")
  (display b)
  (display ".")
  (newline))))

(let ((x 3) (y 4))
  (begin
    (display "Before call to swap: x=")
    (display x)
    (display ", y=")
    (display y)
    (display ".")
    (newline)

    (swap x y)

    (display "After call to swap: x=")
    (display x)
    (display ", y=")
    (display y)
    (display ".")))

```

The output of this program is

```

Before call to swap: x=3, y=4.
In swap: a=4, b=3.
After call to swap: x=3, y=4.

```

Figure 12.4 depicts the run-time stack as this Scheme program executes:

1. (top left) Before `swap` is called. Notice the denoted values of `x` and `y` are *references to expressed values*.
2. (top right) After `swap` is called. Notice that *copies* of the *references* `x` and `y` are passed in.
3. (bottom left) While `swap` executes. Notice that it is the *references* that are swapped. As before, the swap takes place within the activation record of the `swap` function, not the outermost `let` expression.
4. (bottom right) After `swap` returns.

As can be seen, this `swap` function does not swap its reference arguments.

Passing a reference by copy has been referred to as *pass-by-sharing*, especially in languages where all denoted values are references (e.g., Scheme, and Java except for primitives), though use of that term is not common.

Notice also the primary difference between denoted values in C and Scheme in Figures 12.2 and 12.4, respectively. In Scheme, all denoted values are *references* to expressed values; in C, denoted values are the same as expressed values. We need to explore the pass-by-reference parameter-passing mechanism to define a `swap` function that successfully swaps its arguments in the calling function (i.e., persistently).

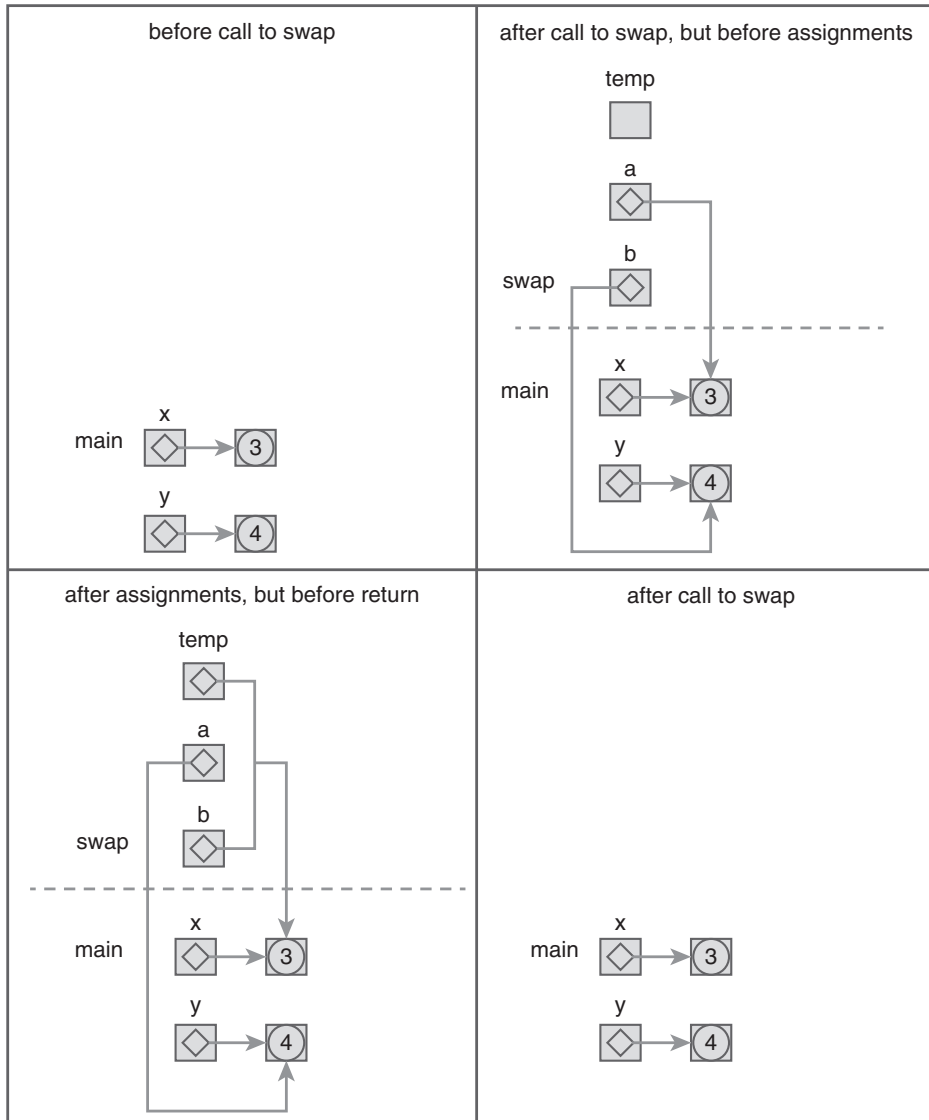


Figure 12.3 Passing of references (to objects) by value in Java. The run-time stack grows upward. (Key: $\square$ = memory cell; $\circ$ = object; $\diamond \rightarrow$ = reference; $\cdots$ = activation-record boundary.)

12.3.2 Pass-by-Reference

In the *pass-by-reference* parameter-passing mechanism, the called function is passed a direct reference to the argument. As a result, changes made to the corresponding parameter in the called function affect the value of the argument in the calling function. Consider the classical `swap` function in C++:

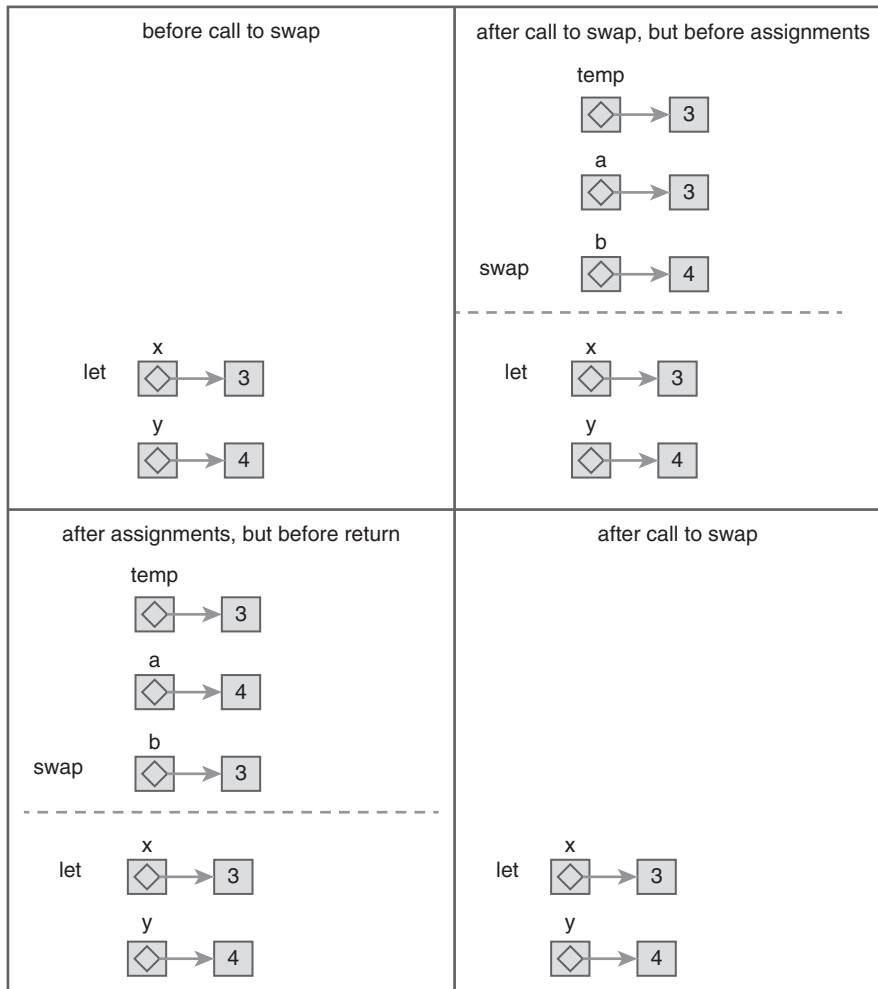


Figure 12.4 Passing arguments by value in Scheme. The run-time stack grows upward. (Key: $\square$ = memory cell; $\diamond \rightarrow$ = reference; $\cdots$ = activation-record boundary.)

```
$ swap_pbv.cpp
#include <iostream>

using namespace std;

/* swap pass-by-reference */
void swap (int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
    cout << "In swap: ";
    cout << "a = " << a << ", b = " << b << endl;
}

int main() {
```

```

    int x = 3;
    int y = 4;

    cout << "In main, before call to swap: ";
    cout << "x = " << x << ", y = " << y << endl;

    swap (x, y);

    printf("In main, after call to swap: ");
    cout << "x = " << x << ", y = " << y << endl;
}

$ gcc swap_pbv.cpp
$ ./a.out
In main, before call to swap: x = 1, y = 2
In swap: a = 2, b = 1
In main, after call to swap: x = 2, y = 1

```

C++ supports both the pass-by-value and pass-by-reference parameter-passing mechanisms. Pass-by-value is the default mechanism in C++. To use pass-by-reference, append an `&` (ampersand) to the end of the data type of any parameter in the signature of the called function that you desire to be passed by reference. Figure 12.5 illustrates the run-time stack during the execution of this C++ swap function with signature `void swap(int& a, int& b)`:

1. (top left) Before `swap` is called. Notice the denoted values of `x` and `y` are `int` and not references to integers.
2. (top right) After `swap` is called. Notice that *references* to `x` and `y` are passed in.
3. (bottom left) While `swap` executes. Notice that changes to the parameters `a` and `b` are reflected in the arguments `x` and `y` in `main`. Thus, unlike with pass-by-value, the swap here takes place within the activation record of the `main` function, and not `swap`.
4. (bottom right) After `swap` returns.

As can be seen, this `swap` function does swap its integer arguments.

As discussed previously, C supports *only* pass-by-value. However, we can *simulate* pass-by-reference in C by passing the memory address of a variable *by value*. Consider the following C program:

```

1  $ cat swap_pabv.c
2  #include<stdio.h>
3
4  /* swap pass address by value: simulated pass-by-reference */
5  void swap (int* a, int* b) {
6      int temp = *a;
7      *a = *b;
8      *b = temp;
9
10     printf("In swap: ");
11     printf("a = %x, b = %x and ", a, b);
12     printf("*a = %d, *b = %d.\n", *a, *b);
13 }
14

```

```

15  int main() {
16
17      int x = 3;
18      int y = 4;
19
20      printf("In main, before call to swap: ");
21      printf("&x = %x, &y = %x and ", &x, &y);
22      printf("x = %d, y = %d.\n", x, y);
23
24      swap (&x, &y);
25
26      printf("In main, after call to swap: ");
27      printf("&x = %x, &y = %x and ", &x, &y);
28      printf("x = %d, y = %d.\n", x, y);
29  }
30
31  $ gcc swap_pabv.c
32  $ ./a.out
33  In main() before call to swap: &x = ef0816ec, &y = ef0816e8 and
34                                x = 3,          y = 4.
35  In swap, a = ef0816ec, b = ef0816e8 and *a = 4, *b = 3.
36  In main() after call to swap: &x = ef0816ec, &y = ef0816e8 and
37                                x = 4,          y = 3.

```

This program is the *simulated-pass-by-reference* analog of the pass-by-value swap program in C. The `&` operator returns the memory address of its variable argument. (e.g., `&x` and `&y` on lines 21, 24, and 27). The `*` operator (e.g., `*a` and `*b` on lines 6–8) is the pointer dereferencing operator in C, which returns the value to which its variable argument points rather than the denoted value of its variable argument. In C, `*` is an overloaded operator. When it appears after the name of a data type, it denotes a pointer type. Thus, `int` is the integer data type and `int*` is the integer pointer data type. A variable of type `int*` is an 8-byte memory cell that stores a hexadecimal address that points to a 4-byte integer. The signature of this swap function is **void** swap(**int*** a, **int*** b). Unlike the prior signature [i.e., **void** swap(**int** a, **int** b)], this function does not accept two integers as arguments. Instead, it accepts two *addresses* to `int` as arguments (i.e., two variables, each of type `int*`). Figure 12.6 shows the status of the run-time stack of this swap function with signature **void** swap(**int*** a, **int*** b):

1. (top left) Before swap is called.
2. (top right) After swap is called. Notice that *copies* of the *addresses* of the integers `x` and `y` are passed in (i.e., the address of `x` and `y` are passed in; see line 24, where `&x = 16ec` and `&y = 16e8`).
3. (bottom left) While swap executes. By dereferencing the pointers `a` and `b` (see `*a` and `*b` on lines 6–8), the swap function is swapping the value to which the memory addresses denoted by `a` and `b` point, rather than swapping the actual memory addresses. Here, the swap takes place within the activation record of the `main` function, not swap.
4. (bottom right) After swap returns.

As can be seen, the function does swap the two integers. But notice that the addresses are still passed *by value*. (Recall that passing references by value or, in other words, simulated pass-by-reference, has been called *pass-by-sharing*.) In

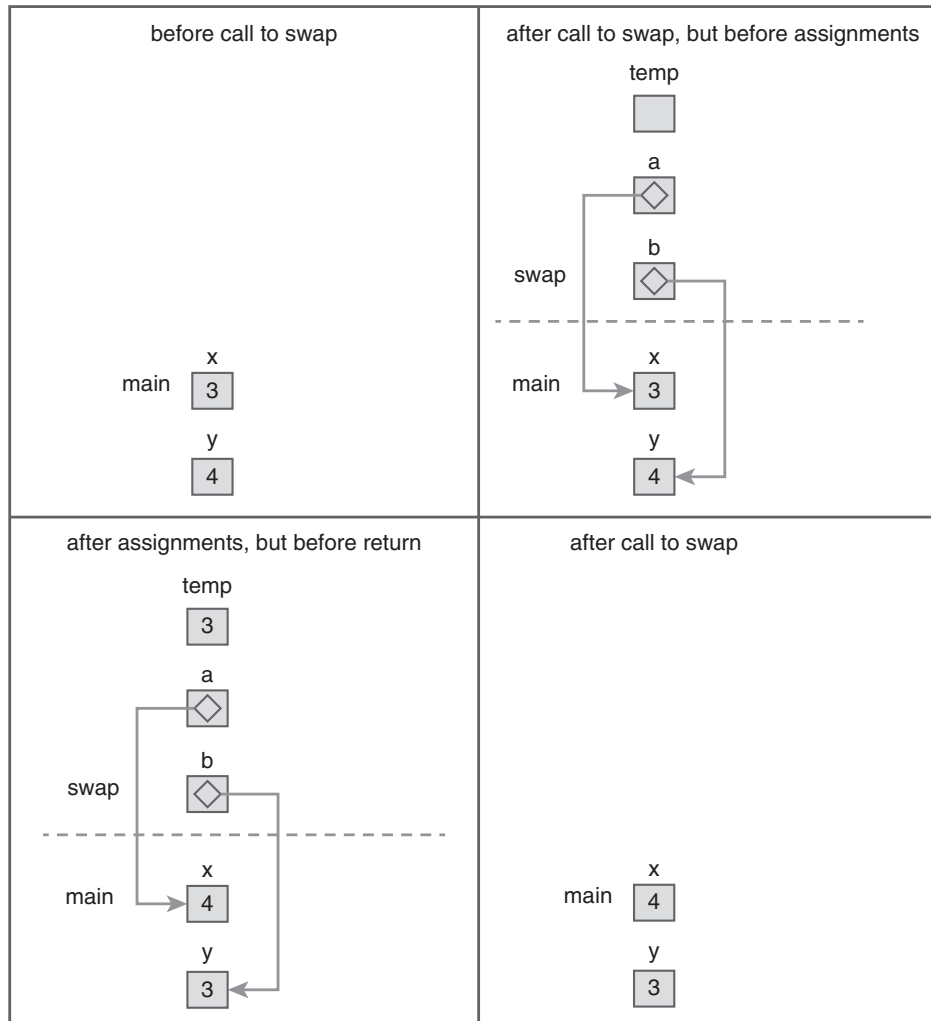


Figure 12.5 The pass-by-reference parameter-passing mechanism in C++. The run-time stack grows upward. (Key: $\square$ = memory cell; $\diamond \rightarrow$ = reference; $\cdots$ = activation-record boundary.)

general, for a C function to modify (i.e., mutate) a variable that is not local to the function, but is also not a global variable, the function must receive a copy of the memory address of the variable it intends to modify rather than its value.

Pass-by-value and pass-by-reference are the two most widely supported parameter-passing mechanisms in programming languages. However, a variety of other mechanisms are supported, especially the pass-by-name and pass-by-need approaches, which are commonly referred to as *lazy evaluation* (Section 12.5). We complete this section by briefly discussing pass-by-result and pass-by-value-result.

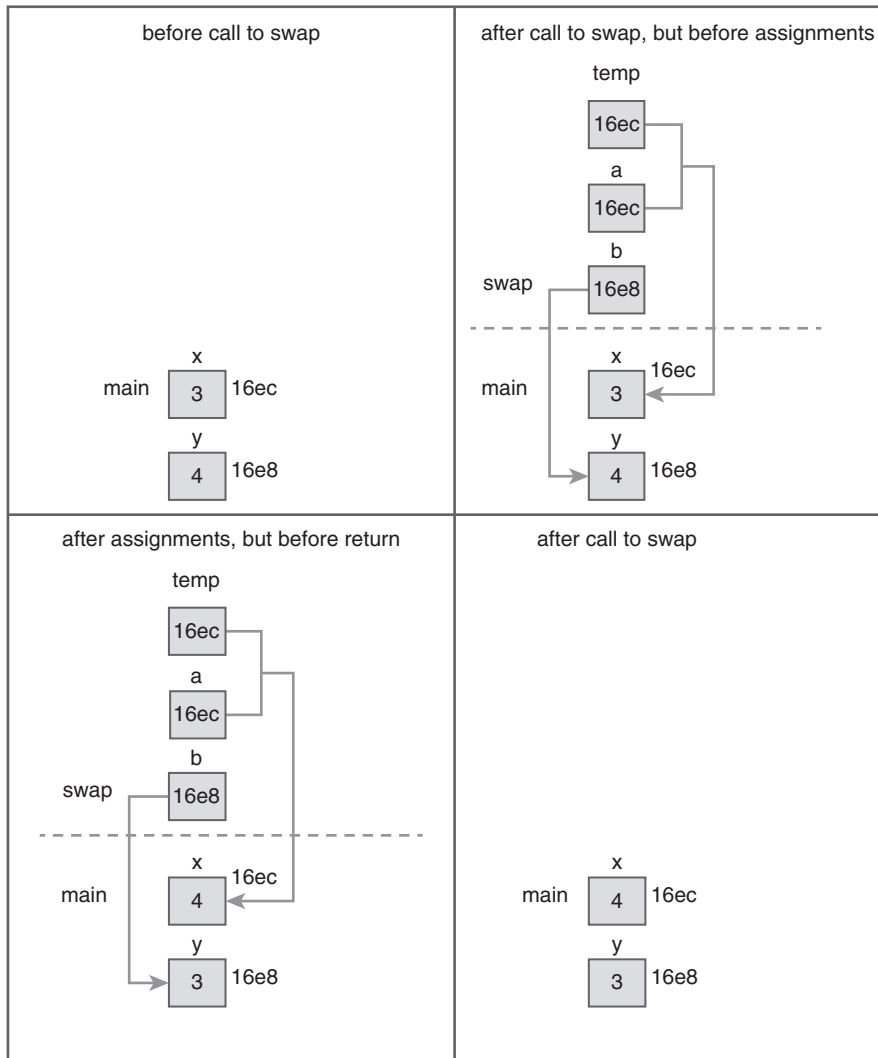


Figure 12.6 Passing memory-address arguments by value in C. The run-time stack grows upward. (Key: $\square$ = memory cell; $\cdots$ = activation-record boundary.)

12.3.3 Pass-by-Result

In the pass-by-value mechanism, copies of the values of the arguments are passed to the called function by copy, but nothing is passed back to the caller. The *pass-by-result* parameter-passing mechanism is the reverse of this approach: No data is passed in to the called function, but copies of the values of the parameters in the called function are passed back to the caller. Consider the following C program:

```
1 void f(int a, int b) {
2     printf("In f, before assignments: ");
```

```

3     printf("a = %d, b = %d.\n", a, b);
4
5     a = 1;
6     b = 2;
7
8     printf("In f, after assignments: ");
9     printf("a = %d, b = %d.\n", a, b);
10 }
11
12 int main() {
13     int x = 3;
14     int y = 4;
15
16     printf("In main, before call to f: ");
17     printf("x = %d, y = %d.\n", x, y);
18
19     f(x,y);
20
21     printf("In main, after call to f: ");
22     printf("x = %d, y = %d.\n", x, y);
23 }

```

The output of this program is

```

In main, before call to f: x = 3, y = 4.
In f, before assignments: a = undefined, b = undefined.
In f, after assignments: a = 1, y = 2.
In main, after call to f: x = 1, y = 2.

```

Note that C syntax is used here only for purposes of illustration; it is not intended to convey that C uses the pass-by-result parameter-passing mechanism. Figure 12.7 presents the run-time stack of this function with signature **void f(int a, int b)**:

1. (top left) Before *f* is called.
2. (top right) After *f* is called. Notice that copies of *x* and *y* are *not* passed in.
3. (bottom left) While *f* executes. Notice that the assignments take place within the activation record of the *f* function, not *main*.
4. (bottom right) After *f* returns.

As shown in the output, the `printf` statement on line 3 is unable to print the values for *a* and *b* because no values are passed into *f*. The values of *x* and *y* in *main* after *f* returns are the final values of *a* and *b* in *f*.

12.3.4 Pass-by-Value-Result

Pass-by-value-result (sometimes referred to as *pass-by-copy-restore*) is a combination of the pass-by-value (on the front end of the call) and pass-by-result (on the back end of the call) parameter-passing mechanisms. In the pass-by-value-result mechanism, arguments are passed into the called function in the same manner as with the pass-by-value approach (i.e., by copy). However, the values of the corresponding parameters within the called function are passed back to the caller in the same manner as with the pass-by-result mechanism (i.e., by copy). Consider the following C program:

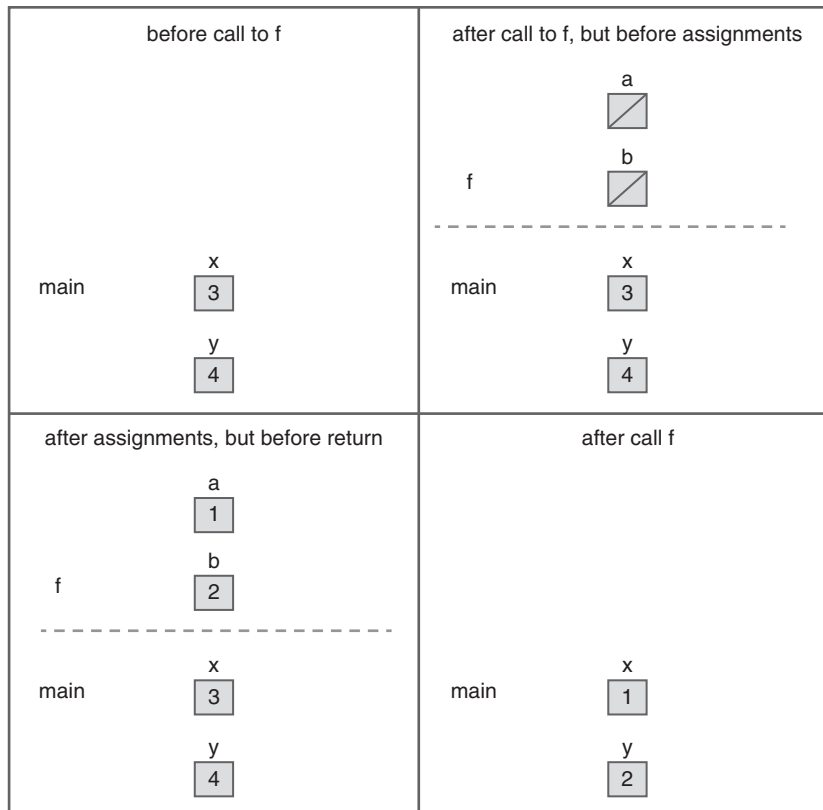


Figure 12.7 Passing arguments by result. The run-time stack grows upward. (Key: = memory cell; = activation-record boundary.)

```

void f (int a, int b) {
    printf("In f, before assignments: ");
    printf("a = %d, b = %d.\n", a, b);

    a = a + 2;
    b = b + 2;

    printf("In f, after assignments: ");
    printf("a = %d, b = %d.\n", a, b);
}

int main() {
    int x = 3;
    int y = 4;

    printf("In main, before call to f: ");
    printf("x = %d, y = %d.\n", x, y);

    f(x,y);

    printf("In main, after call to f: ");
    printf("x = %d, y = %d.\n", x, y);
}

```

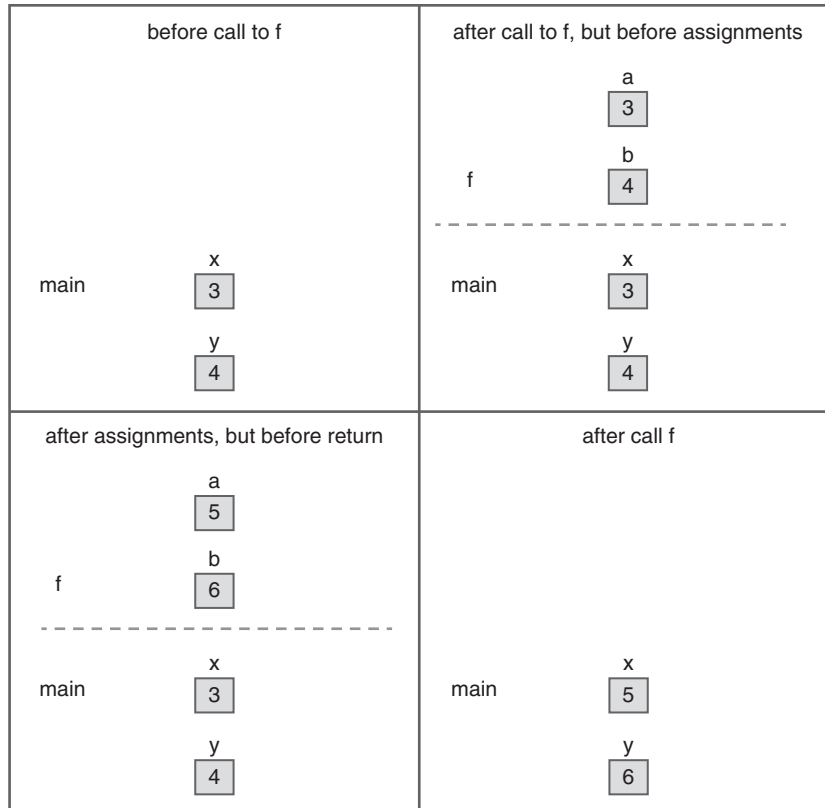



Figure 12.8 Passing arguments by value-result. The run-time stack grows upward. (Key: = memory cell; = activation-record boundary.)

The output of this program is

```
In main, before call to f: x = 3, y = 4.
In f, before assignments: a = 3, b = 4.
In f, after assignments: a = 1, y = 2.
In main, after call to f: x = 1, y = 2.
```

Again, C syntax is used here only for purposes of illustration; it is not intended to convey that C uses the pass-by-value-result mechanism. Figure 12.8 presents the run-time stack of this function with signature **void f(int a, int b)**:

1. (top left) Before *f* is called.
2. (top right) After *f* is called. Notice that *copies* of *x* and *y* are passed in.
3. (bottom left) While *f* executes. Notice that the additions and assignments take place within the activation record of the *f* function, not *main*.
4. (bottom right) After *f* returns. Notice that *copies* of the values of the parameters *a* and *b* from the called function *f* are copied back to the calling function *main*.

12.3.5 Summary

The following abbreviations identify the direction in which data flows between the calling and called functions in parameter-passing mechanisms:

- IN = data flows from the calling to the called function.
- OUT = data flows from the called function back to the calling function.
- IN-OUT = data flows both from the calling to the called function and from the called function back to the calling function.

The following is a classification using these mnemonics to help think about the parameter-passing mechanisms discussed in this section.

- pass-by-value: IN
- pass-by-result: OUT
- pass-by-reference: IN-OUT
- pass-by-value-result: IN at the front; OUT at the back

Although they may appear to be the same, note that

pass-by-reference (IN-OUT) $\neq$ pass-by-value-result (IN-OUT)

This inequality is explored in Programming Exercise 12.3.13.

The pass-by-value parameter-passing mechanism works the same way in all of the languages used to illustrate it here. The factor on which a successful swap depends is the sets of expressed and denoted values in each language. These sets do vary in C, C++, Java, and Scheme. Table 12.2 summarizes this and other factors in the languages discussed in this section in relation to parameter-passing. Figure 12.9 presents this tabular summary in a graphical fashion.

Language	Denoted Value	Dereferencing	Parameter-Passing Mechanism
Java	reference to object, or primitive value	implicit	by value (for both primitives and references)
Scheme	reference to expressed value	implicit	by value
C	expressed value	explicit (use <code>*</code> ; e.g., <code>*x</code>)	by value; pass address argument [e.g., <code>swap(&x, &y);</code>] to simulate pass-by-reference
C++	expressed value	explicit for addresses (use <code>*</code> ; e.g., <code>*x</code>); implicit for references	by value or by reference [prepend <code>&</code> to parameter; e.g., <code>void swap(int& x, int& y)</code>]

Table 12.2 Relationship Between Denoted Values, Dereferencing, and Parameter-Passing Mechanisms in Programming Languages Discussed in This Section

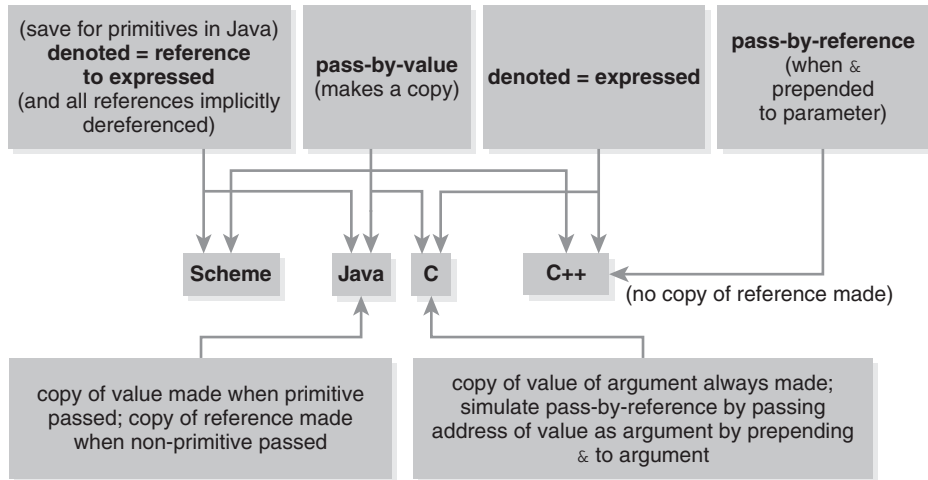


Figure 12.9 Summary of parameter-passing concepts in Java, Scheme, C, and C++. (Key: arrow from concept to language indicates source concept is supported in target language.)

Conceptual Exercises for Section 12.3

Exercise 12.3.1 What are some of the disadvantages of the *pass-by-value* parameter-passing mechanism? Explain.

Exercise 12.3.2 Indicate which parameter-passing mechanisms the following programming languages support: Swift, Smalltalk, C#, Ruby, Python, and Perl.

Exercise 12.3.3 Which parameter-passing mechanism does ML use? Explain with reasons and defend your answer with code.

Exercise 12.3.4 Consider the following C program:

```

1 void f (int a, int b) {
2     a = a + 1;
3     b = b + 1;
4 }
5
6 int main() {
7     int x = 1;
8     f (x, x);
9     printf("x = %d.\n", x);    /* what is the value of x here? */
10 }
11
12 /* if pass-by-value is used,      x=?
13    if pass-by-reference is used, x=?
14    if pass-by-result is used,    x=?
15    if pass-by-value-result is used, x=?

```

Give the output that the `printf` statement on line 9 produces if the arguments to the function `f` on line 8 are passed using the following parameter-passing mechanisms:

- (a) pass-by-value
- (b) pass-by-reference
- (c) pass-by-result
- (d) pass-by-value-result

Exercise 12.3.5 As illustrated in this section, we cannot write a `swap` method in Java because all variables in Java—both primitives and references to objects—are passed by value. Given this approach in Java, what could a programmer do to swap two integers in Java?

Exercise 12.3.6 Consider the following Java program:

```

1  class Exercise {
2
3      private static void increment(Integer i) {
4
5          i = Integer.valueOf(Integer.valueOf(i) + 1);
6
7          System.err.print("In increment: ");
8          System.err.println("i = " + Integer.valueOf(i) + ".");
9      }
10
11     public static void main(String args[]) {
12
13         Integer i = Integer.valueOf(5);
14
15         System.err.print("In main, before call to increment: ");
16         System.err.println("i = " + Integer.valueOf(i) + ".");
17
18         Exercise.increment(i);
19
20         System.err.print("In main, after call to increment: ");
21         System.err.println("i = " + Integer.valueOf(i) + ".");
22     }
23 }
```

The output of this program is

```

$ javac Exercise.java
$ java Exercise
In main, before call to increment: i = 5.
In increment: i = 6.
In main, after call to increment: i = 5.
```

Given that denoted values in Java are references for all identifiers declared as objects, such as `i` on line 13, explain why the `i` on line 21 in the `main` method does not reflect the incremented value of `i` (i.e., the value 6) after the call to the method `increment` on line 18.

Exercise 12.3.7 Consider the following C program:

```

1  #include<stdio.h>
2
```

```

3  int i = 2;
4  int A[200];
5
6  void f(int x, int y) {
7      i = x+y;
8  }
9
10 int main() {
11     A[i] = 99;
12     f(i, A[i]);
13     printf("i = %d\n", i);
14     printf("A[i] = %d\n", A[i]);
15 }

```

Passing the arguments to the function `f` on line 12 using which of the parameter-passing mechanisms discussed in this section produces the following output:

```

i = 2
A[i] = 99

```

Defend your answer. There may be more than one answer.

Exercise 12.3.8 How can a called function, which is evaluated using the *pass-by-result* parameter-passing mechanism, reference a parameter whose corresponding argument is a literal or an expression [e.g., `f(1, a+b);`]?

Exercise 12.3.9 Can *parameter collision* [e.g., `f(x, x);`] occur in a function evaluated using the *pass-by-value-result* parameter-passing mechanism? Describe the problems that may occur.

Exercise 12.3.10 In the *pass-by-result* parameter-passing mechanism, when should the address of each parameter be evaluated?

Programming Exercises for Section 12.3

Exercise 12.3.11 Replicate in Python the *pass-by-value* `swap` function and program in C.

Exercise 12.3.12 Define a function in C, Python, or Scheme, and present an invocation of it, that produces different results when its arguments are passed *by result* and *by value-result*. Explain in comments in the program how one of the mechanisms produces different results than the other.

Exercise 12.3.13 Define a Python function, and present an invocation of it, that produces different results when its arguments are passed *by value-result* than when passed *by reference*. Explain in comments in the program how one of the mechanisms produces different results than the other. Keep the function to five lines of code or less.

Exercise 12.3.14 Write a `swap` method in Java that successfully swaps its arguments in the calling function. *Hint:* The arguments being swapped cannot

be of type `int` or `Integer`, but rather must be references to objects whose data members of type `int` are the values being swapped.

Exercise 12.3.15 Is it possible to *simulate pass-by-reference* in Scheme? If so, write a Scheme function that demonstrates this simulation. *Hint*: Explore the concept of *boxing* in Scheme. Can you rewrite your function without boxing yet still simulate pass-by-reference? Explain. Provide your answer as a comment in your program.

12.4 Implementing Pass-by-Reference in the Camille Interpreter

The Camille interpreter currently supports only *pass-by-value* because every time the interpreter encounters an operand, it creates a new reference. For instance, in the following Camille program, the assignment to passed argument `x` in the called function `f` does not affect the value of `x` in the outermost `let` expression:

```
Camille> let
  x = 3
  f = fun(a) assign! a = 4
in
  let
    d = (f x)
  in
    x
3
```

The denoted value of `a` is a reference that initially contains a *copy* of the value with which the reference `x` is associated, *but these references are distinct*. Thus, the assignment to `a` in the body of the function `f` has no effect on the `x` in the outermost `let` expression; as a result, the value of the expression is 3.

Let us implement *pass-by-reference* in Camille. We want to mutate Camille so that literals (i.e., integers and functions/closures) are passed by value and variables are passed by reference. The difference between the purely pass-by-value Camille interpreter and the new hybrid pass-by-value (for literals), pass-by-reference (for variables) Camille interpreter is summarized as follows:

- *Pass-by-value* involves creating a new reference for the evaluation of *every* operand.
- *Pass-by-reference* involves creating a new reference only for the evaluation of a *literal* operand.

In other words, unlike the prior implementation of Camille, now we only create a new reference for *literal* operands. In the prior implementation, we created a new reference for *every* operand. As a consequence, in Camille now, we use:

- *pass-by-value* for *literals* (i.e., numbers and functions/closures)
- *pass-by-reference* for all *non-literals* (i.e., variables)

12.4.1 Revised Implementation of References

We retain the following:

expressed value	=	integer $\cup$ closure
denoted value	=	reference to an expressed value

However, we need a revised implementation of references. A reference is still a location in a list. However, instead of only containing expressed values, the elements of that list can now contain either expressed values or denoted values—which are references to expressed values. We use the following terminology (Friedman, Wand, and Haynes 2001):

- A list element that contains an expressed value is called a *direct target* (i.e., pass-by-value).
- A list element that contains a denoted value is called an *indirect target* (i.e., pass-by-reference).

The following is an abstract-syntax implementation of a Target ADT as well as the revised abstract-syntax implementation of the Reference ADT:

```

1  def expressedvalue(x):
2      return (isinstance(x,int) or is_closure(x))
3
4  # begin abstract-syntax representation of Target
5
6  class Target:
7      def __init__(self, value, flag):
8
9          type_flag_dict = { "directtarget" : expressedvalue,
10                           "indirecttarget" :
11                               (lambda x : isinstance(x, Reference)) }
12
13          # if flag is not a valid flag value,
14          # build a lambda function that always
15          # returns false so we throw an error
16          type_flag_dict = \
17              defaultdict(lambda: lambda x: False, type_flag_dict)
18          if (type_flag_dict[flag](value)):
19              self.flag = flag
20              self.value = value
21          else:
22              raise Exception("Invalid Target Construction.")
23
24  # end abstract-syntax representation of Target
25
26  # begin abstract-syntax representation of Reference
27
28  class Reference:
29      # ...
30      # definitions of primitive_dereference and
31      # primitive_assignreference functions are same as earlier
32      # ...
33
34      def dereference(self):
35          target = self.primitive_dereference()
36          if target.flag == "directtarget":
37              return target.value

```

```

38     elif target.flag == "indirecttarget":
39         innertarget = target.value.primitive_dereference()
40         if innertarget.flag == "directtarget":
41             return innertarget.value
42     # double indirect references not allowed
43     raise Exception("Invalid dereference.")
44
45 def assignreference(self, expressedvalue):
46     target = self.primitive_dereference()
47
48     if target.flag == "directtarget":
49         temp = self
50     elif target.flag == "indirecttarget":
51         innertarget = target.value.primitive_dereference()
52         if innertarget.flag == "directtarget":
53             temp = target.value
54         elif innertarget.flag == "indirecttarget":
55             # double indirect references not allowed
56             raise Exception("Invalid creation of reference.")
57
58     temp.primitive_assignreference(Target(expressedvalue, "directtarget"))
59
60 # end abstract-syntax representation of Reference

```

12.4.2 Reimplementation of the `evaluate_operand` Function

The `extend_environment` and `apply_environment_reference` functions need not change. However, the function `extend_environment` now accepts a list of targets and returns a list containing those targets. The function `apply_environment_reference` looks up an identifier and creates a reference to the location containing the appropriate target.

We now have the support structures in place to implement the pass-by-reference parameter-passing mechanism. Let us consider each context in which subexpressions are evaluated. For primitive applications, we simply pass the value. For instance:

```

1  def evaluate_expr(expr, environ):
2      try:
3          if expr.type == ntPrimitive_op:
4              # expr leaf is mapped during parsing to
5              # the appropriate binary operator function
6              argumentRefs = \
7              list (evaluate_prim_app_expr_operands(expr.children, environ)) [0]
8              arguments = []
9
10             for argref in argumentRefs:
11                 if isinstance(argref, Reference):
12                     arguments.append(argref.dereference())
13                 else:
14                     arguments.append(argref)
15
16             return apply_primitive (expr.leaf, arguments)
17
18     # rest of cases in evaluate_expr
19     elif ...
20     ...

```


where `evaluate_prim_app_expr_operands` is defined as

```
def evaluate_prim_app_expr_operands(operands, environ):
    return map (lambda x: evaluate_expr (x, environ), operands)
```

Therefore, the evaluation of primitive applications is unchanged and remains as pass-by-value. We will also retain pass-by-value for `let`-bound variables:

```
21 elif expr.type == ntLet:
22     temp = evaluate_expr (expr.children[0], environ) # assignment
23
24     identifiers = []
25     arguments = []
26
27     for name in temp:
28         identifiers.append (name)
29         arguments.append (evaluate_let_expr_operand (temp[name]))
30
31     temp = evaluate_expr (expr.children[1],
32                           extend_environment (identifiers, arguments,
33                                                environ)) # evaluation
34
35     return localbindingDereference(temp)
36
37 # rest of cases in evaluate_expr
38 elif ...
39     ...
```

where the `evaluate_let_expr_operand` and `localbindingDereference` functions are defined, respectively, as

```
def evaluate_let_expr_operand(operand):
    if isinstance(operand, Reference):
        operand = operand.dereference()
    return Target(operand, "directtarget")

def localbindingDereference(possiblereference):
    if isinstance(possiblereference, Reference):
        return possiblereference.dereference()
    else:
        return possiblereference
```

We define these functions because some expressions in the bindings or body of a `let` expression may not evaluate to references (e.g., `let a=5 in 5`). Therefore, we must inspect the value returned from `evaluate_expr` to determine if it is a reference that needs to be dereferenced before it is used. The evaluation of `let` expressions in Camille is also unchanged and remains as pass-by-value. For function applications, we continue to evaluate each operand using `evaluate_operand`:

```
1 def evaluate_operand(operand, environ):
2
3     if isinstance(operand, Reference):
4
5         ## if the operand is a variable, then it denotes a location
6         ## containing an expressed value and
7         ## we return an "indirect target" pointing to that location
```

```

8      target = operand.primitive_dereference()
9
10     ## if the variable is bound to a "location" that
11     ## contains a direct target,
12
13     if target.flag == "directtarget":
14
15         ## then we return an indirect target to that location
16         return Target(operand, "indirecttarget")
17
18     ## but if the variable is bound to a "location"
19     ## that contains an indirect target, then
20     ## we return the same indirect target
21
22     elif target.flag == "indirecttarget":
23         innertarget = target.value.primitive_dereference()
24         if innertarget.flag == "indirecttarget":
25
26             # double indirect references not allowed
27             return Target(innertarget, "indirecttarget")
28         else:
29             return innertarget
30
31     ## if the operand is a literal (i.e., integer or function/closure),
32     ## then we create a new location, as before, by returning
33     ## a "direct target" to it (i.e., pass-by-value)
34
35     elif isinstance(operand, int) or is_closure(operand):
36         return Target(operand, "directtarget")

```

Let us unpack the three cases of operands handled in this function:

- If the operand is a *literal* [e.g., an integer (`ntNumber`) or function/closure (`ntFuncDecl`)], then return a *direct target* to it (lines 31–36).
- If the operand is a *variable* (i.e., `ntIdentifier`) that points to a *direct target*, then return an *indirect target* to it (lines 10–16).
- If the operand is a *variable* (i.e., `ntIdentifier`) that points to an *indirect target*, then return a *copy* of the same *indirect target* (lines 18–29).
- If the operand is a *variable* (i.e., `ntIdentifier`) that points to an *indirect target* (lines 18–29) that points to an *indirect target*, then return a *copy* of the same *indirect target* (line 27).
- If the operand is a *variable* (i.e., `ntIdentifier`) that points to an *indirect target* (lines 18–29) that points to a *direct target*, then return the *direct target* (line 29).

This definition of the `evaluate_operand` function maintains the invariant that a reference contains either an expressed value or a reference to an expressed value. It also means that Camille does not support double indirect references (e.g., `int** x` in C).

Consider the following illustrative Camille program modified from Friedman, Wand, and Haynes (2001):

```

(fun (a, b, c, d)      --- we refer to this literal function as f1
  (fun (e, f)          --- we refer to this literal function as f2
    (fun (g, h, i)     --- we refer to this literal function as f3
      assign! h=31

```

e, 6, f)	---	e, 6, f	are the arguments to f3
5, c)	---	5, c	are the arguments to f2
1, 2, 3, 4)	---	1, 2, 3, 4	are the arguments to f1

Figure 12.10 presents the references associated with the arguments to the three literal functions in this program. Notice that both parameters `b` and `y` are indirect targets to parameter `v`, which is a direct target to the argument 7, rather than `y` being an indirect target to the indirect target `b`—double indirect pointers are not supported. Figure 12.11 depicts the relationship of the references `b` and `y` to each other and to the argument 7 in more detail. Since the Camille interpreter now supports pass-by-reference for variable arguments, a Camille function is now able to modify the value of an argument:

```
Camille> let
  x = 1
in
  let
    f = fun(a) assign! a = incl(a) --- a++
  in
    let
      d = (f x)
    in
      --- function f has changed
      x --- the value of x from 1 to 2

2
```

Now we can also define a `swap` function in Camille that successfully swaps its arguments in the calling expression/function:

```
Camille> let
  x = 3
  y = 4

  --- swap function: pass-by-reference
  swap = fun(a,b)
    let
      temp = a --- temp = a
    in
      let
        ignored1 = assign! a = b --- a = b
      in
        assign! b = temp --- b = temp
    in
      let
        ignored2 = (swap x,y)
      in
        -(x,y) --- since x and y have been swapped, 4 - 3 = 1

1
```

Programming Exercise for Section 12.4

Exercise 12.4.1 (Friedman, Wand, and Haynes 2001, Exercise 3.55, p. 114) Implement the *pass-by-value-result* parameter-passing mechanism in the version of Camille in Section 12.2 (i.e., 3.0). To use pass-by-value-result, the argument must

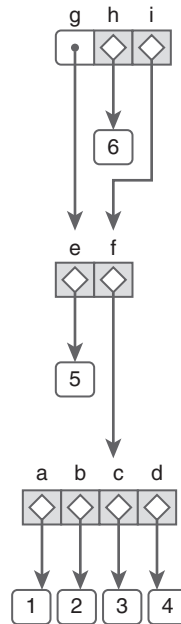


Figure 12.10 Three layers of references to indirect and direct targets representing parameters to functions (Friedman, Wand, and Haynes 2001). (Key: $\square$ = memory cell; $\diamond \rightarrow$ = reference.)

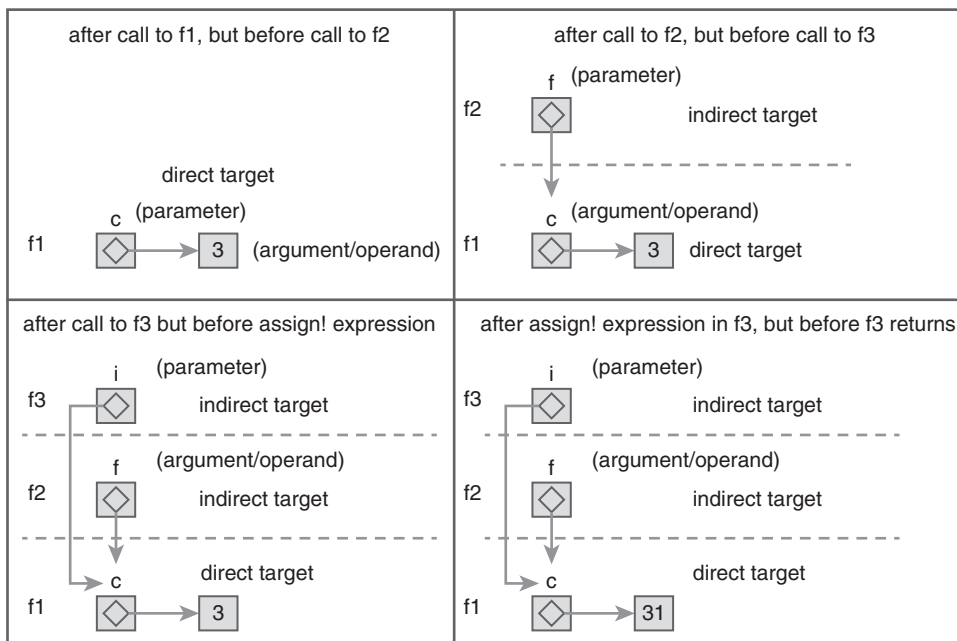


Figure 12.11 Passing variables by reference in Camille. The run-time stack grows upward. (Key: $\square$ = memory cell; $\diamond \rightarrow$ = reference; $\cdots$ = activation-record boundary.)

be a variable. When an argument is passed by value-result, the parameter is bound to a new reference initialized to the value of the argument, akin to pass-by-value. The body of the function is then evaluated as usual. However, when the function returns, the value in the new reference is copied back into the reference denoted by the argument. In addition to the modified Camille interpreter, provide a Camille program that produces different results using pass-by-reference and pass-by-value-result.

12.5 Lazy Evaluation

12.5.1 Introduction

At its core, *lazy evaluation* is a parameter-passing strategy in which an operand is evaluated only when its value is needed. This simple idea has compelling consequences. An obvious advantage of this approach is that if the value of an operand is never needed in the body of a function, then the time required to evaluate it is saved. Most languages, including Python and Java, implement *short-circuit evaluation*, which is an instance of lazy evaluation restricted to boolean operators. For instance, in the expression `false && (true || false)`, there is no need to evaluate the subexpression on the right-hand side of the logical “and” (`&&`) operator.

In what follows, we first describe the mechanics of the lazy evaluation parameter-passing mechanism and briefly consider how to implement it. We then discuss the compelling implications it has for programming.

12.5.2 β -Reduction

Lazy evaluation supports the simplest possible form of reasoning about a program:

1. Replace every function call with its body.
2. Replace every reference to a parameter in the body of a function with the corresponding argument.

Formally, this evaluation strategy in λ -calculus is called β -reduction (or the *copy rule*). More practically, we can say lazy evaluation involves *simple string substitution* (e.g., substitution of a function name for the function body, and substitution of parameters for arguments); for this reason, the lazy evaluation parameter-passing mechanism is sometimes generally referred to as *pass-by-name*.

We use Scheme to demonstrate β -reduction. Consider the following simple squaring function: `(define square (lambda (x) (* x x)))`. Let us temporarily forget that this is a Scheme function that can be evaluated. Instead, we will simply think of this expression as associating the string `(lambda (x) (* x x))` with the mnemonic `square`. Now, consider the following expression: `(square 2)`. We will temporarily suspend the association of this expression with an “invocation” of `square` and simply think of it as a string. Now, let us apply the two substitutions. Step 1 involves replacing the

mnemonic (i.e., identifier) `square` with the string associated with it (i.e., the body of the function); step 2 involves replacing each `x` in the replacement string (from step 1) with `2` (i.e., replacing each reference to a parameter in the body of the function with the corresponding argument):

$$(\text{square } 2) \xRightarrow{\text{apply step 1}} ((\text{lambda } (x) \text{ } (* x x)) 2) \xRightarrow{\text{apply step 2}} (* 2 2)$$

Expressing the steps of β -reduction in λ -calculus, if $\text{square} = (\lambda x . x * x)$, then

$$\text{square}(2) \xRightarrow{\text{apply step 1}} (\lambda x . x * x)(2) \xRightarrow{\text{apply step 2}} 2 * 2$$

Thinking of these steps as a parameter-passing mechanism (i.e., pass-by-name) may seem foreign, especially since most readers may be most familiar with pass-by-value semantics and internally conceptualize run-time stacks visually (e.g., Figures 12.2–12.8). However, when viewed through a purely mathematical lens, this “parameter-passing mechanism” is quite natural. For instance, if we told someone without a background in computing that $x = (3 \diamond 2)$, and then inquired as to the representation of $x \diamond x$, that person would likely intuitively respond with $(3 \diamond 2) \diamond (3 \diamond 2)$. Thus, if $x = (3 * 2)$, the representation of $x * x$ is, similarly, $(3 * 2) * (3 * 2)$, not $6 * 6$. Again, this interpretation is purely mathematical and independent of any implementation approaches or constraints. Now let us consider another “invocation” of `square`: `(square (* 3 2))`. Using β -reduction:

$$(\text{square } (* 3 2)) \xRightarrow{\text{apply step 1}} ((\text{lambda } (x) \text{ } (* x x)) (* 3 2)) \xRightarrow{\text{apply step 2}} (* (* 3 2) (* 3 2)) \Rightarrow (* 6 6) \Rightarrow 36$$

We can compare this evaluation of the function with the typical programming language semantics for this invocation:

$$(\text{square } (* 3 2)) \Rightarrow (\text{square } 6) \Rightarrow ((\text{lambda } (x) \text{ } (* x x)) (6)) \Rightarrow (* 6 6) \Rightarrow 36$$

The following Scheme code presents another comparison of these two approaches:

```
;; normal-order evaluation
> (square (* 3 2))
(* (* 3 2) (* 3 2))
(* 6 6)
36

;; applicative-order evaluation
> (square (* 3 2))
(square 6)
(* 6 6)
36
```

The former approach is called *lazy evaluation* or *normal-order evaluation*: It evaluates the arguments to a function if they are *needed* during the evaluation of the body of the function. Thus, lazy evaluation is sometimes generally referred to as *pass-by-need*. The latter approach is called *eager evaluation* or *applicative-order evaluation*: It evaluates the arguments to a function prior to evaluating the body of the function.

Intuitively, it would seem that the use of lazy evaluation (of arguments) is intended for purposes of efficiency. Specifically, if the argument is not needed in the body of the function, the time that would have been spent on evaluating it is saved. However, upon closer examination, in a (perceived) attempt to be efficient, the evaluation of the expression `(square (* 3 2))` requires double the work—the expression `(* 3 2)` passed as an argument is evaluated twice! (We discuss the relationship between lazy evaluation and space complexity in Section 13.7.4).

When considering the savings in time resulting from not evaluating an unused argument, one might question why a programmer would define a function that accepts an argument it does not use. In other words, it seems as if lazy evaluation is a safeguard against poorly defined functions. However, when we think about boolean operators as functions, and operands to boolean operators as arguments to a function, then suddenly it makes sense not to use eager evaluation: `false && (true || false)`. Similarly, when thinking of an `if` conditional structure as a ternary boolean function and thinking of the conditional expression, true branch, and false branch as arguments to this function, using eager evaluation is unreasonable:

```
;;; unreasonable in eager languages
(define our-if
  (lambda (condition usual-value exceptional-value)
    (cond
      (condition usual-value)
      (else exceptional-value))))
```

Thus, in many languages using pass-by-value semantics, including Camille, the `if` conditional structure is implemented as a syntactic form as opposed to a user-defined function; for example, we implement conditionals in Camille in `evaluate_expr` and not as a user-defined function. Since arguments to functions are evaluated eagerly in these languages, the `if` structure must be implemented as a syntactic form. This is also why programmers cannot extend (or modify) control structures (e.g. `if`, `while`, or `for`) in such languages using standard mechanisms (e.g., a user-defined function) *within* the language itself.

The terms on each of the main rows of Table 12.3 are generally used interchangeably to refer to evaluation strategies for function arguments. However, sometimes the term used depends on the scope to which it is applied: a specific language in its entirety (e.g., “Haskell is a lazy language”), a particular function invocation (e.g., “evaluate `f(x, 3*2)` using normal-order evaluation”), or a specific argument (e.g., “`x` is a non-strict argument”). Also, do not be misled

Language Level	Invocation Level	Argument Level
eager evaluation	= applicative-order evaluation	= strict
lazy evaluation	= normal-order evaluation	= non-strict

Table 12.3 Terms Used to Refer to Evaluation Strategies for Function Arguments in Three Progressive Contexts

by the word *normal*. Applicative-order evaluation most likely seems “normal” to readers familiar with programming in languages like Python and Java. However, as mentioned previously, the β -reduction approach is more intuitive, natural, or “normal” to someone without a background in computing.

12.5.3 C Macros to Demonstrate Pass-by-Name: β -Reduction Examples

Let us apply β -reduction to multiple programs and make some notable observations on the results. The expansion of macros defined in C/C++ using `#define` by the C preprocessor involves the string substitution in β -reduction. Thus, (the expansion of) C macros can be used to demonstrate the β -reduction involved in functions whose arguments are evaluated lazily.¹ We begin with a brief introduction to C macros. Consider the following C program:

```

1  $ cat macros.c
2  #include<stdio.h>
3
4  #define FIVE 5
5
6  #define SQUARE(X) ((X)*(X))
7
8  /* #A in a replacement string of a macro:
9     1. replace by argument (i.e., actual parameter)
10    2. enclose it in quotes */
11  #define PRINT(A, B) printf("#A ": %d, " #B ": %d\n", A, B)
12
13  /* max of two ints macro */
14  #define MAX(a,b) ((a) > (b) ? (a) : (b))
15
16  int main() {
17      int x = SQUARE(3);
18      int y = SQUARE(x+1);
19
20      printf("%d\n", FIVE);
21
22      PRINT(x, y);
23
24      printf ("The max of %d and %d is %d.\n", 1, 2, MAX(1,2));
25      printf ("The max of %d and %d is %d.\n", x, y, MAX(x,y));
26      printf ("The max of %d and %d is %d.\n", y, x, MAX(y,x));
27      printf ("The max of %d and %d is %d.\n", x+1, y+1,
28              MAX(++x, ++y));
29      x--; y--;
30
31      printf ("The max of %d and %d is %d.\n", x+1, y,
32              MAX(x++, y));
33  }
```

1. The examples of C macros in this chapter are not intended to convey that C macros correspond to lazy evaluation. “Macros do not correspond to lazy evaluation. Laziness is a property of when the implementation evaluates arguments to functions. ... Indeed, macro expansion (like type-checking) happens in a completely different phase than evaluation, while laziness is very much a part of evaluation. So please don’t confuse the two” (Krishnamurthi 2003). The examples of C macros used here are simply intended to help the reader get a feel for the pass-by-name parameter-passing mechanism and β -reduction; they are used entirely for purposes of demonstration.

The following code is the result of the expansion of the four macros on lines 4, 6, 11, and 14:

```

1  $ gcc -E macros.c > macros.E # cpp macros.c > macros.E can be used as well
2  $ cat macros.E
3  int main() {
4      int x = ((3)*(3));
5      int y = ((x+1)*(x+1));
6
7      printf("%d\n", 5);
8
9      printf("x" ": %d, " "y" ": %d\n", x, y);
10
11     printf ("The max of %d and %d is %d.\n", 1, 2, ((1) > (2) ? (1) : (2)));
12     printf ("The max of %d and %d is %d.\n", x, y, ((x) > (y) ? (x) : (y)));
13     printf ("The max of %d and %d is %d.\n", y, x, ((y) > (x) ? (y) : (x)));
14     printf ("The max of %d and %d is %d.\n", x+1, y+1,
15             ((++x) > (++y) ? (++x) : (++y)));
16     x--; y--;
17
18     printf ("The max of %d and %d is %d.\n", x+1, y,
19             ((x++) > (y) ? (x++) : (y)));
20 }
```

When the C preprocessor encounters a `#define`, it substitutes the third string on the same line of code as the definition for all occurrences in the program of the second string on that line. For instance, line 4 of the unexpanded program is the definition of the macro `FIVE`: `#define FIVE 5`. Thus, the preprocessor replaces the statement

```
printf ("%d\n", FIVE); (line 20 of the unexpanded program)
```

with

```
printf ("%d\n", 5); (line 7 of the expanded program)
```

In other words, `FIVE` is textually replaced with `5`. This substitution can be thought of as solely step 1 of β -reduction.

Expanding the macros defined on lines 6, 11, and 14 involves both steps 1 and 2 of β -reduction. For instance, consider the `SQUARE` macro defined on line 6 of the unexpanded version. Using this macro to demonstrate β -reduction:

1. All occurrences of the string `SQUARE` in the program (e.g., lines 17 and 18) are replaced with `((X)*(X))`;
2. All occurrences of `X` in the replacement string are substituted with `3`.

Thus, the statement

```
int x = SQUARE(3); (line 17 in the unexpanded program)
```

is replaced with the statement

```
int x = ((3)*(3)); (line 4 in the expanded program)
```

Similarly, the statement

```
int y = SQUARE(x+1); (line 18)
```

is replaced with

```
int y = ((x+1)*(x+1)); (line 5)
```

Prefacing a string representing a parameter in the replacement string of a macro with # causes the corresponding argument to be enclosed in double quotes after it is replaced for the parameter. For instance, the statement

```
PRINT(x, y); (line 22)
```

is replaced with the statement

```
printf("x": %d, "y": %d\n", x, y); (line 9)
```

because the PRINT(A, B) macro is defined as

```
printf("#A ": %d, "#B ": %d\n", A, B) (line 11)
```

The MAX macro defined on line 14 is similarly expanded; that is, lines 24–26, 28, and 32 are replaced with lines 11–13, 15, and 19, respectively. The output of this program is

```
1 $ gcc macros.c
2 $ ./a.out
3 5
4 x: 9, y: 100
5 The max of 1 and 2 is 2.
6 The max of 9 and 100 is 100.
7 The max of 100 and 9 is 100.
8 The max of 10 and 101 is 102.
9 The max of 10 and 101 is 101.
```

Line 8 of this output appears to be incorrect or, at least, inaccurate. Conceptual Exercise 12.5.1 explores why.

With an understanding of the β -reduction conducted by the C preprocessor as it expanded macros, we can define the classical swap function as a macro to explore pass-by-name semantics. Consider the following C program:

```
1 #include<stdio.h>
2
3 /* pass-by-name swap macro */
4 #define swap(a, b) { int temp = (a); (a) = (b); (b) = temp; }
5
6 int main() {
7
8     int x = 3;
9     int y = 4;
10    int temp = 5;
11
12    printf ("Before pass-by-name swap(x,y) macro: x = %d, y = %d\n", x, y);
13
14    swap(x,y)
15
16    printf (" After pass-by-name swap(x,y) macro: x = %d, y = %d\n\n", x, y);
17 }
```

The `swap` macro is defined on line 4. The preprocessed version of this program with the `SWAP` macro expanded is

```

1  int main() {
2
3  int x = 3;
4  int y = 4;
5  int temp = 5;
6
7  printf ("Before pass-by-name swap(x,y) macro: x = %d, y = %d\n", x, y);
8
9  { int temp = (x); (x) = (y); (y) = temp; }
10
11 printf ("After pass-by-name swap(x,y) macro: x = %d, y = %d\n\n", x, y);
12 }
```

The output of this program is

```

$ gcc swap_pbn.c
$ ./a.out
Before pass-by-name swap(x,y) macro: x = 3, y = 4
After pass-by-name swap(x,y) macro: x = 4, y = 3
```

The output indicates that the pass-by-name `swap` macro worked. However, another use of this `swap` macro tells a different story:

```

#include<stdio.h>

/* pass-by-name swap macro */
#define swap(x, y) { int temp = (x); (x) = (y); (y) = temp; }

int main() {
    int a[6];
    int i = 1;
    a[1] = 5;

    printf ("Before pass-by-name swap(i, a[i]) macro: i = %d, a[1] = %d\n",
            i, a[i]);

    swap(i, a[i]);

    printf (" After pass-by-name swap(i, a[i]) macro: i = %d, a[1] = %d\n",
            i, a[1]);
}
```

The output of this program is

```

$ gcc swap_pbn2.c
$ ./a.out
Before pass-by-name swap(i, a[i]) macro: i = 1, a[1] = 5
After pass-by-name swap(i, a[i]) macro: i = 5, a[1] = 5
```

The values of `i` and `a[1]` are not swapped after the expanded code from the `swap` macro executes: `a[1]` is 5 both before and after the replacement code of the macro executes. This outcome occurs because of side effect. The expansion of the macro replaces the statement

```
swap(i, a[i]);
```

with

```
{ int temp = (i); (i) = (a[i]); (a[i]) = temp; };
```

The side effect of the second assignment statement `(i) = (a[i])` changes the value of `i` from 1 to 5. Thus, the third assignment, `(a[i]) = temp;`, places the original value of `i` (i.e., 1) in array element `a[5]` rather than `a[1]`. Consequently, after the replacement code of the macro executes, `a[1]` is unchanged. Side effect caused a similar problem in the execution of the replacement code of the `MAX` macro on line 28 in the first C program in Section 12.5.3, which produced the following output: The max of 10 and 101 is 102. Thus, we rephrase the first sentence of Section 12.5.2 as “lazy evaluation in a language without side effects supports the simplest possible form of reasoning about a program.” We explore the implications of side effect for the pass-by-name parameter-passing mechanism further in the Conceptual Exercises.

12.5.4 Two Implementations of Lazy Evaluation

Reconsider the `square` function defined in Scheme in Section 12.5.2: `(define square (lambda (x) (* x x)))`. Recall that the β -reduction involved in the pass-by-name semantics of the `(square (* 3 2))` invocation of `square` resulted in the argument expression `(* 3 2)` being evaluated twice because the parameter `x` is referenced twice in the body of the `square` function. Implementations of lazy evaluation differ in how they handle multiple references to the same parameter, as in the body of a function:

- *Pass-by-name*: Evaluate the argument expression every time the parameter is referenced in the body of the function being evaluated.
- *Pass-by-need*: Only evaluate the argument expression the first time the parameter is referenced in the body of the function, but save the value so that it can be retrieved for any subsequent references to the parameter. This saves the time needed to reevaluate the argument expression with each subsequent reference.

In a function without side effect, evaluating arguments to the function with pass-by-name semantics yields the same result as doing so with pass-by-need semantics. Thus, in languages without side effects, it is practical to use pass-by-need semantics to save the time required to repeatedly reevaluate an expression argument (i.e., a thunk) that will always returns the same value. However, in a function with side effects, evaluating arguments to the function with pass-by-name semantics may not yield the same result as doing so with pass-by-need semantics. For instance, consider the following Python program:

```
1  x = 0
2
3  def inc_x():
4      global x
5      x = x + 1
6      return x
7
```

```

8  # two references to one parameter in body of function
9  def double(x):
10     return x + x
11
12     print(double(inc_x()))
13
14 # one reference to each parameter in body of function,
15 # but each parameter is same
16 def add(x,y):
17     return x + y
18
19 # reset x
20 x = 0
21
22 print(add(inc_x(), inc_x()))

```

If the argument `inc_x()` is *passed by name* to the `double` function on line 12, then the `double` function returns 3 ($= 1 + 2$) because the parameter `x` is referenced *twice* in the body of the `double` function (line 10). Thus, the argument expression `inc_x()` is evaluated twice: The first time it is evaluated `inc_x()` returns 1, and the second time it returns 2 because `inc_x` has a side effect (i.e., it increments the global variable `x`). In contrast, if the argument `inc_x()` is *passed by need* to the `double` function on line 12, then the `double` function returns 2 ($= 1 + 1$) because the argument expression `inc_x()` is evaluated only once: The first time `inc_x()` returns the value 1, which is stored so that it can be retrieved the second time the parameter `x` is referenced.

Contrast the definitions of the `double` (lines 9–10) and `add` (line 16–17) functions: The `double` function accepts one parameter, which it references twice in its body (line 10); the `add` function accepts two parameters, each of which it references once in its body (line 17). However, the `add` function is invoked with the same expression for each argument (line 22). If each argument `inc_x()` is *passed by name* to the `add` function on line 22, then the `add` function returns 2 ($= 1 + 2$). While the parameters `x` and `y` are each referenced only once in the body of the `add` function (line 17), the argument expression `inc_x()` is evaluated twice, but for a different reason than it is for the pass-by-name invocation of the `double` function on line 12. Here, the argument expression `inc_x()` is evaluated once for each of the `x` and `y` parameters because the same argument expression is passed for both parameters. The first time `inc_x()` is evaluated, it returns 1, and the second time it returns 2 because `inc_x` has a side effect. Evaluating the invocation of the `add` function on line 22 using *pass-by-need* semantics yields the same result. Since each parameter is referenced only once in the body of the `add` function (line 17), there is no opportunity to retrieve the return value of each argument expression recorded. In other words, there is no opportunity to obviate a reevaluation during a subsequent reference because there are no subsequent references. Thus, unlike the invocation of the `double` function on line 12, the invocation of the `add` function on line 22 yields the same result when using pass-by-name or pass-by-need semantics.

Note that the Java or C analog of the invocation to the `add` function on line 22 is `x=0; x++ + x++;`, where `x++` is an argument that is passed (by name or by need) twice to the `+` function. In summary,

pass-by-name	is	non-memoized	lazy evaluation;
pass-by-need	is	memoized	lazy evaluation.

ALGOL 60 was the first language to use pass-by-name, while Haskell was the first modern language to use pass-by-need. The statistical programming language R also uses the pass-by-name parameter-passing mechanism.

12.5.5 Implementing Lazy Evaluation: Thunks

Implementing lazy evaluation involves building support for pass-by-name and pass-by-need arguments. Lazy evaluation is easily implemented in a programming language with first-class functions and closures (e.g., Python or Scheme). We must delay the evaluation of an operand (perhaps indefinitely) by encapsulating it within the body of a function with no arguments—called a *thunk*. A thunk acts as a shell for a delayed argument expression. A thunk must contain all the information required to produce the value of the argument expression when it is needed in the body of a function, as if it had been evaluated at the time of the function application. Thus, a thunk is sometimes called a *promise*—invoking a thunk promises to return the value of the expression that the thunk encapsulates. To produce the value of the expression on demand, a thunk must have access to both the argument expression and the environment at the time of the call. Consider the following Python function `f`:

```

1 >>> def f(x, y):
2 ...     if x == 0:
3 ...         return 1
4 ...     else:
5 ...         return y()
6 ...

```

In the function call on line 5, `y` is not an invocation of some arbitrary Python function defined elsewhere, but rather the second parameter to the function `f`. When `f` is invoked with a non-zero integer as the first argument and the expression `(1/0)` as the second argument in a language that uses eager evaluation (e.g., Scheme, Java, or Python), it produces a run-time error:

```

7 >>> f(0, (1/0))
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10 ZeroDivisionError: division by zero

```

To avoid this run-time error, we can pass the second argument to `f` *by name*. Thus, instead of passing the expression `(1/0)` as the second argument, we must pass a thunk:

```

11 >>> # This function is a thunk (or a shell) for the expression 1/0.
12 >>> def divbyzero():
13 ...     return 1/0
14 ...
15 >>> # invoking f with a named function as the second argument
16 >>> f(0, divbyzero)

```

forming a <i>thunk</i> (or a <i>promise</i>)	=	freezing an expression operand	=	delaying its evaluation
evaluating a <i>thunk</i> (or a <i>promise</i>)	=	thawing a thunk	=	forcing its evaluation

Table 12.4 Terms Used to Refer to Forming and Evaluating a Thunk

```

17 1
18 >>> # invoking f with a lambda expression as the second argument
19 >>> f(0, lambda: 1/0)
20 1

```

When the argument being passed involves references to variables [e.g., (x/y) instead of $(1/0)$], the thunk created for the argument requires more information. Specifically, the thunk needs access to the referencing environment that contains the bindings to the variables being referenced.

Rather than hard-code a thunk every time we desire to delay the evaluation of an argument (as shown in the preceding example), we desire to develop a pair of functions for forming and evaluating a thunk (Table 12.4). We can then invoke the thunk-formation function each time the evaluation of an argument expression should be delayed (i.e., each time a pass-by-name argument is desired). Thus, we want to abstract away the process of thunk formation. Since a thunk is simply a nullary (i.e., argumentless) function, evaluating it is straightforward:

```

21 >>> def force(thunk):
22 ...     return thunk()

```

The definition of the thunk to be created depends on the use of pass-by-name or pass-by-need semantics. On the one hand, if the argument to be delayed is to be passed by name, thunk formation is straightforward:

```

23 >>> # pass-by-name semantics
24 >>> def delay(expr):
25 ...     # return a thunk
26 ...     return lambda: eval(expr)

```

The Python function `eval` accepts a string representing a Python expression, evaluates it, and returns the result of the expression evaluation. Implementing pass-by-need semantics, on the other hand, requires us to

1. Record the value of the argument expression the first time it is evaluated (line 36).
2. Record the fact that the expression was evaluated once (line 37).
3. Look up and return the recorded value for all subsequent evaluations (line 41).

```

27 >>> # pass-by-need semantics
28 >>> def delay(expr):
29 ...     result = [False]
30 ...     first = [True]
31 ...
32 ...     # define a thunk
33 ...     def thunk():

```

```

34 ...     if first[0]:
35 ...         print("first and only computation")
36 ...         result[0] = eval(expr)
37 ...         first[0] = False
38 ...     else:
39 ...         print("lookup, no recomputation")
40 ...
41 ...     return result[0]
42 ...
43 ...     # return a thunk
44 ...     return thunk

```

Notice that the `delay` function builds the thunk as a first-class closure so that it can “remember” the return value of the evaluated argument expression in the variable `result` after `delay` returns. First-class closures are an important construct for implementing a variety of concepts from programming languages.

Since `delay` is a user-defined function and uses applicative-order evaluation, we must pass a string representing an expression, rather than an expression itself, to prevent the expression from being evaluated. For instance, in the invocation `delay (1/0)`, the argument to be delayed [i.e., `(1/0)`] is a strict argument and will be evaluated eagerly (i.e., before it is passed to `delay`). Thus, we must only pass strings (representing expressions) to `delay`:

```

45 >>> # invoking f with a thunk as the second argument
46 >>> f(0, delay("1/0"))
47 1

```

Enclosing the argument in quotes in Python is the analog of using the `quote` function or single quote in Scheme—for example, `(quote (/ 1 0))` or `' (/ 1 0)`.

Now let us apply our newly defined functions for lazy evaluation in Python to function invocations whose arguments involve references to variables as opposed to solely literals. Thus, we reconsider the Python program from Section 12.5.4:

```

48 >>> x = 0
49
50 >>> def inc_x():
51 ...     global x
52 ...     x = x + 1
53 ...     return x
54
55 >>> # two references to one parameter in body of function
56 >>> def double(x):
57 ...     return force(x) + force(x)
58
59 >>> double(delay("inc_x()"))
60 first and only computation
61 lookup, no recomputation
62 2
63
64 >>> # one reference to each parameter in body of function,
65 >>> # but each parameter is same
66 >>> def add(x,y):
67 ...     return force(x) + force(y)
68

```



```

69 >>> x = 0
70
71 >>> add(delay("inc_x()"), delay("inc_x()"))
72 first and only computation
73 first and only computation
74 3

```

In this program, we call `delay` to suspend the evaluation of the arguments in the function invocations (lines 59 and 71), and we use the function `force` in the body of functions to evaluate the argument expressions represented by the parameters when those parameters are needed (lines 57 and 67). In other words, a thunk is formed and passed for each argument using the `delay` function, and those thunks are evaluated using the `force` function when referenced in the bodies of the functions. Again, notice the difference in the two functions invoked with non-strict arguments. The function `double` is a unary function that references its sole parameter twice; the function `add` is a binary function that references each of its parameters once. Thus, the advantage of pass-by-need is only manifested with the invocation to `double`. The output of the invocation of `double` (line 59) is

```

60 first and only computation
61 lookup, no recomputation
62 2

```

The second reference to `x` does not cause a reevaluation of the thunk. The output of the invocation of `add` on line 71 is

```

72 first and only computation
73 first and only computation
74 3

```

In the invocation of the `add` function, one thunk is created for each argument and each thunk is separate from the other. While the two thunks are duplicates of each other, each thunk is evaluated only once.

The Scheme `delay` and `force` syntactic forms (which use *pass-by-need* semantics, also known as *memoized lazy evaluation*) are the analogs of the Python function `delay` and `force` defined here. Programming Exercise 12.5.19 entails implementing the Scheme `delay` and `force` syntactic forms as user-defined Scheme functions.

The Haskell programming language was designed as an intended standard for lazy, functional programming. In Haskell, pass-by-need is the default parameter-passing mechanism and, thus, the use of syntactic forms like `delay` and `force` is unnecessary. Consider the following transcript with Haskell:²

```

1 Prelude> import Data.Function (fix)
2 Prelude Data.Function> fix (\x -> x)
3 ^CInterrupted.

```

2. We cannot use the simpler argument expression `1/0` to demonstrate a *non-strict* argument in Haskell because `1/0` does not generate a run-time error in Haskell—it returns `Infinity`.

```

4  Prelude Data.Function> -- f is guaranteed to return successfully
5  Prelude Data.Function> -- using lazy evaluation
6  Prelude Data.Function> f x = 2
7  Prelude Data.Function> f (fix (\x -> x))
8  2
9
10 Prelude Data.Function> False && (fix (\x -> x))
11 False
12
13 Prelude Data.Function> True || (fix (\x -> x))
14 True

```

The Haskell function `fix` returns the *least fixed point* of a function in the domain theory interpretation of a fixed point. A *fixed point* of a function is a value x such that $f(x) = x$. For instance, a fixed point of a square root function $f(x) = \sqrt{x}$ is 1 because $\sqrt{1} = 1$. Since there is no least fixed point of an identity function $f(x) = x$, the invocation `fix (\x -> x)` never returns—it searches indefinitely (lines 2–3). Haskell supports pass-by-value parameters as a special case. When an argument is prefaced with `$!`, the argument is passed by value or, in other words, the evaluation of the argument is forced. In this case, the argument is treated as a *strict* argument and evaluated *eagerly*:

```

15 Prelude Data.Function> -- use of $! forces evaluation of (fix (\x -> x))
16 Prelude Data.Function> f $! (fix (\x -> x))
17 ^CInterrupted.

```

The built-in Haskell function `seq` evaluates its first argument before returning its second. Using `seq`, we can define a function `strict`:

```

18 Prelude Data.Function> strict f x = seq x (f x)

```

We can then apply `strict` to treat an argument to a function `f` as strict. In other words, we evaluate the argument `x` eagerly before evaluating the body of `f`:

```

19 Prelude Data.Function> :type seq
20 seq :: a -> b -> b
21
22 Prelude Data.Function> :type strict
23 strict :: (t -> b) -> t -> b
24
25 Prelude Data.Function> strict f (fix (\x -> x))
26 ^CInterrupted.

```

There is an interesting relationship between the space complexity of a function and the strategy used to evaluate parameters (e.g., non-strict or strict). We discuss the details in Section 13.7.4. For now, it is sufficient to know that an awareness of the space complexity of a program is important, especially in languages using lazy evaluation. Moreover, “[t]he space behavior of lazy programs is complex: ... some programs use less space than we might predict, while others use more” (Thompson 2007, p. 413). Finally, strict parameters are primarily used in lazy languages to improve the space complexity of a function.

12.5.6 Lazy Evaluation Enables List Comprehensions

Lazy evaluation leads to potentially infinite lists that are referred to as *list comprehensions* or *streams*. More generally, lazy evaluation leads to infinite data structures (e.g., trees). For instance, consider the Haskell expression `ones = 1 : ones`. Since the evaluation of the arguments to cons are delayed by default, `ones` is an infinite list of 1s. Haskell supports the definition of list comprehension using syntactic sugar:

```

1  Prelude> ones = 1 : ones
2  Prelude>
3  Prelude> -- .. is syntactic sugar and, thus,
4  Prelude> -- [1,1..] is shorthand for 1:ones
5  Prelude> ones = [1,1..]
6  Prelude>
7  Prelude> nonnegatives = [0..]
8  Prelude> naturals = [1,2..] -- same as naturals = [1..]
9  Prelude> evens = [2,4..]
10 Prelude> odds = [1,3..]
```

We can define functions `take1` and `drop1` to access parts of list comprehensions:³

```

11 Prelude> :{
12 Prelude| take1 0 _ = []
13 Prelude| take1 _ [] = []
14 Prelude| take1 n (h:t) = h : take1 (n-1) t
15 Prelude|
16 Prelude| drop1 0 l = l
17 Prelude| drop1 _ [] = []
18 Prelude| drop1 n (_:t) = drop1 (n-1) t
19 Prelude| :}
```

Let us unpack the evaluation of `take1 2 ones`:

```

20 Prelude> take1 2 ones
21 [1,1]
22 Prelude> :{
23 Prelude| {--
24 Prelude| take1 2 ones = 1 : take1 (2-1) (1:ones)
25 Prelude|               1 : take1 (2-1-1) (1:ones)
26 Prelude|               take1 0 (1:ones)
27 Prelude|               []
28 Prelude|               [1]
29 Prelude|               [1,1]
30 Prelude| --}
31 Prelude| :}
32 Prelude>
33 Prelude> take1 100 positives
34 [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...,98,99,100]
35 Prelude> nums100 = take1 100 positives
```

Since only enough of the list comprehension is explicitly realized when needed, we can think of this as laying down railroad track as we travel rather than building

3. We use the function names `take1` and `drop1` because these functions are defined in Haskell as `take` and `drop`, respectively.

the entire railroad prior to embarking on a voyage. Thus, we must be mindful when applying functions to streams so to avoid enumerating the list ad infinitum. Consider the following continuation of the preceding transcript:

```

36 Prelude> squares = [n*n | n <- naturals]
37 Prelude> elem 16 squares
38 True
39 Prelude> elem 15 squares -- searches indefinitely
40 ^CInterrupted.
41 Prelude> :{
42 Prelude| -- guarded equations are an alternative to
43 Prelude| -- conditional expressions;
44 Prelude| -- guarded equations tend to be more readable than
45 Prelude| -- conditional expressions
46 Prelude| sortedElem e (x:xs)
47 Prelude|     | x < e = sortedElem e xs
48 Prelude|     | x == e = True
49 Prelude|     | otherwise = False
50 Prelude| :}
51 Prelude> sortedElem 15 squares
52 False

```

Note on line 36 that Haskell uses notation similar to *set-former* or *set-builder notation* from mathematics to define the `squares` list comprehension: $\text{squares} = \{n * n \mid n \in \mathbb{N}, \text{ where } \mathbb{N} = \{1, 2, \dots, \infty\}\}$. We can see that Haskell brings programming closer to mathematics. Here, the invocation of the built-in `elem` (or `member`) function (line 37) returns `True` because 16 is a square. However, the `elem` function does not know that the input list is sorted, so it will search for 15 (line 39) indefinitely. While doing so, it will continue to enumerate the list comprehension indefinitely. Defining a `sortedElem` function that assumes its list argument is sorted causes the search and enumeration (line 51) to be curtailed once it encounters a number greater than its first argument.

Lazy evaluation also leads to terse implementation of complex algorithms. Consider the implementation of both the *Sieve of Eratosthenes* algorithm for generating prime numbers (in two lines of code) and the *quicksort* sorting algorithm (in four lines of code):

```

53 Prelude> :{
54 Prelude| -- implementation of Sieve of Eratosthenes algorithm
55 Prelude| -- for enumerating prime numbers
56 Prelude| sieve [] = []
57 Prelude| sieve (two:lon) = two : sieve [n | n <- lon, (mod n two) /= 0]
58 Prelude| :}
59 Prelude>
60 Prelude> primes = sieve [2..]
61 Prelude>
62 Prelude> take1 100 primes
63 [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,...,523,541]
64 Prelude>
65 Prelude> :{
66 Prelude| quicksort [] = []
67 Prelude| quicksort (h:t) = quicksort [x | x <- t, x <= h]
68 Prelude|                     ++ [h] ++
69 Prelude|                     quicksort [x | x <- t, x > h]
70 Prelude| :}
71 Prelude>

```

```

72 Prelude> quicksort [9,6,8,7,10,3,4,2,1,5]
73 [1,2,3,4,5,6,7,8,9,10]
74 Prelude>
75 Prelude> reverse first100primes
76 [541,523,521,509,503,499,491,487,479,467,463,461,457,449,443,...,3,2]
77 Prelude>
78 Prelude> first100primes = take1 100 primes
79 Prelude> unsorted = reverse first100primes
80 Prelude>
81 Prelude> quicksort unsorted
82 [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,...,523,541]

```

Let us trace the sieve [2,3,4,5,6,7,8,9,10] invocation:

```

Prelude| sieve (two:lon)= two : sieve [n | n <- lon, (mod n two) /= 0]

sieve [2,3,4,5,6,7,8,9,10] = -- filter out all multiples of 2
    2 : sieve [3,5,7,9] = -- filter out all multiples of 3
        2 : 3 : sieve [5,7] = -- filter out all multiples of 5
            2 : 3 : 5 : sieve [7] = -- filter out all multiples of 7
                2 : 3 : 5 : 7 : sieve [] =
                    2 : 3 : 5 : 7 : [] =
                        [2,3,5,7]

```

The beauty of using lazy evaluation in the implementation of this algorithm is that the filter will filter the list only as far as the function that called `sieve` (e.g., `take1`) requires. We can see in the `sieve` function that lazy evaluation enables a *generate-filter* style of programming (Hughes 1989) resembling the filter style of programming common in Linux, where concurrent processes are not only communicating, but also maintaining synchronous execution with each other, through a possible infinite stream of data flowing through pipes—for example, `cat lazy.txt | aspell list | sort | uniq | wc -l`. Similarly, let us trace the `quicksort` [10,9,8,7,6,5,4,3,2,1] invocation:

```

quicksort [5,1,9,2,8,3,7,4,6,10] =

quicksort 5 : [1,9,2,8,3,7,4,6,10] =

(quicksort [1,2,3,4] ++ [5] ++ quicksort [9,8,7,6,10]) =

((quicksort [] ++ [1] ++ quicksort [2,3,4])
++ [5] ++
 (quicksort [8,7,6] ++ [9] ++ quicksort [10])) =

(([] ++ [1] ++ (quicksort [] ++ [2] ++ quicksort [3,4]))
++ [5] ++
 ((quicksort [7,6] ++ [8] ++ quicksort []) ++ [9] ++
 (quicksort [] ++ [10] ++ quicksort []))) =

(([] ++ [1] ++ ([] ++ [2] ++ (quicksort [] ++ [3] ++ quicksort [4])))
++ [5] ++
 ((quicksort [6] ++ [7] ++ quicksort []) ++ [8] ++ [])
++ [9] ++
 ([] ++ [10] ++ [])) =

(([] ++ [1] ++ ([] ++ [2] ++ ([] ++ [3] ++ (quicksort []
++ [4] ++

```

```

                                quicksort [])))))
++ [5] ++
((quicksort [6] ++ [7] ++ []) ++ [8] ++ [])
++ [9] ++
([] ++ [10] ++ [])) =

(([] ++ [1] ++ ([] ++ [2] ++ ([] ++ [3] ++ ([] ++ [4] ++ []))))
++ [5] ++
((quicksort [] ++ [6] ++ quicksort []) ++ [7] ++ []) ++ [8] ++ [])
++ [9] ++
([] ++ [10] ++ [])) =

(([] ++ [1] ++ ([] ++ [2] ++ ([] ++ [3] ++ ([] ++ [4] ++ []))))
++ [5] ++
((([] ++ [6] ++ []) ++ [7] ++ []) ++ [8] ++ [])
++ [9] ++
([] ++ [10] ++ [])) =

(([] ++ [1] ++ ([] ++ [2] ++ ([] ++ [3] ++ ([4]))))
++ [5] ++
((([] ++ [6] ++ []) ++ [7] ++ []) ++ [8] ++ []) ++ [9] ++ ([10])) =

(([] ++ [1] ++ ([] ++ [2] ++ ([] ++ [3] ++ [4])))
++ [5] ++
([6] ++ [7] ++ []) ++ [8] ++ []) ++ [9] ++ [10]) =

(([] ++ [1] ++ ([] ++ [2] ++ ([3,4])))
++ [5] ++
([6,7] ++ [8] ++ []) ++ [9] ++ [10]) =

(([] ++ [1] ++ ([] ++ [2] ++ [3,4]))
++ [5] ++
([6,7] ++ [8] ++ []) ++ [9] ++ [10]) =

(([] ++ [1] ++ ([2,3,4]))
++ [5] ++
([6,7,8] ++ [9] ++ [10]) =

(([] ++ [1] ++ [2,3,4])
++ [5] ++
[6,7,8] ++ [9] ++ [10]) =

([1,2,3,4] ++ [5] ++ [6,7,8] ++ [9] ++ [10]) =

([1,2,3,4,5,6,7,8,9,10]) = [1,2,3,4,5,6,7,8,9,10]

```

While Python evaluates arguments eagerly, it does have facilities that enable the program to define infinite streams, thereby obviating the enumeration of a large list in memory. Python makes a distinction between a *list comprehension* and a *generator comprehension* or *generator expression*. In Python, a generator expression is what we call a list comprehension in Haskell—that is, a function that generates list elements on demand. List comprehensions in Python, however, are syntactic sugar for defining an enumerated list without a loop using set-former notation. Consider the following transcript with Python:

```

1  >>> import sys
2

```

```

3 >>> squaresListcomp = [n*2 for n in range(1000)] # list comprehension
4
5 >>> type(squaresListcomp)
6 <class 'list'>
7
8 >>> sys.getsizeof(squaresListcomp)
9 9016
10
11 >>> squaresListcomp[4]
12 8
13
14 >>> squaresGenexpr = (n*2 for n in range(1000)) # generator expression
15
16 >>> type(squaresGenexpr)
17 <class 'generator'>
18
19 >>> sys.getsizeof(squaresGenexpr)
20 112
21
22 >>> squaresGenexpr[4]
23 Traceback (most recent call last):
24   File "<stdin>", line 1, in <module>
25 TypeError: 'generator' object is not subscriptable
26
27 >>> sum(squaresListcomp)
28 999000
29
30 >>> sum(squaresGenexpr)
31 999000

```

Syntactically, the only difference between lines 3 and 14 is the use of square brackets in the definition of the list comprehension (line 3) and the use of parentheses in the definition of the generator expression (line 14). However, lines 9 and 20 reveal a significant savings in space required for the generator expression. In terms of space complexity, a list comprehension is preferred if the programmer intends to iterate over the list multiple times; a generator expression is preferred if the list is to be iterated over once and then discarded. Thus, if only the sum of the list is desired, a generator expression (line 30) is preferable to a list comprehension (line 27). Generator expressions can be built using functions calling `yield`:

```

1 >>> # the use of yield turns the function into a generator expression;
2 >>> # naturals() is a generator expression
3 >>> def naturals():
4 ...     i = 1
5 ...     while True:
6 ...         yield i
7 ...         i += 1
8
9 >>> from itertools import islice
10
11 >>> # analog of Haskell's take function
12 >>> def take(n, iterable):
13 ...     #returns first n elements of iterable as a list
14 ...     return list(islice(iterable, n))
15
16 >>> take(10, naturals())
17 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```

Lines 1–7 define a generator for the natural numbers (e.g., `[1..]` in Haskell). Without the `yield` statement on line 6, this function would spin in an infinite loop and never return. The `yield` statement is like a `return`, except that the next time the function is called, the state in which it was left at the end of the previous execution is “remembered” (see the concept of *coroutine* in Section 13.6.1). The `take` function defined on lines 11–14 realizes in memory a portion of a generator and returns it as a list (lines 16–17).

12.5.7 Applications of Lazy Evaluation

Streams and infinite data structures are useful in a variety of artificial intelligence problems and applications involving search (e.g., a game tree for tic-tac-toe or chess) for avoiding the need to enumerate the entire search space, especially since large portions of the space need not ever be explored. The power of lazy evaluation in obviating the need to enumerate the entire search space prior to searching it is sufficiently demonstrated in the solution to the simple, yet emblematic for purposes of illustration, same-fringe problem. The *same-fringe problem* is a classical problem from functional programming that requires a generator-filter style of programming. The problem entails determining if the non-null n atoms in two S-expressions are equal and in the same order. A straightforward approach proceeds in this way:

1. Flatten both lists.
2. Recurse down each flat list until a mismatch is found.
3. If a mismatch is found, the lists do not have the same fringe.
4. Otherwise, if both lists are exhausted, the fringes are equal.

Problem: If the first non-null atoms in each list are different, we flattened the lists for naught. Lazy evaluation, however, will realize only enough of each flattened list until a mismatch is found. If the lists have the same fringe, each flattened list must be fully generated. The same-fringe problem calls for the power of lazy evaluation and the streams it enables. Programming Exercises 12.5.21 and 12.5.22 explore solutions to this problem.

12.5.8 Analysis of Lazy Evaluation

Three properties of lazy evaluation are:

- “[I]f there exists any evaluation sequence which terminates for a given expression, then [pass]-by-name evaluation will also terminate for this expression, and produce the same final result (Hutton 2007, p. 129).
- [A]rguments are evaluated precisely once using [pass]-by-value evaluation, but may be evaluated many times using [pass]-by-name (Hutton 2007, p. 130).

- [U]sing lazy evaluation, expressions are only evaluated as much as required by the context in which they are used” (Hutton 2007, p. 132).

The power of lazy evaluation is manifested in the form of solutions to problems it enables. By acting as the glue binding entire programs together, lazy evaluation enables a generate-filter style of programming that is reminiscent of the filter style of programming in which pipes are used to connect processes communicating through I/O in UNIX (e.g., `cat lazy.txt | aspell list | sort | uniq | wc -l`). Lazy evaluation and higher-order functions are tools that can be used to both modularize a program and generalize the modules, which makes them reusable (Hughes 1989).

Curried HOFs + Lazy Evaluation = Modular Programming

12.5.9 Purity and Consistency

Lazy evaluation encourages uniformity in languages because it obviates the need for syntactic forms for constructs for which *applicative-order evaluation* is unreasonable (e.g., `if`). As a consequence, a language can be extended by a programmer in standard ways, such as through a user-defined function. Consider Scheme, which uses *applicative-order evaluation* by default.

- Syntactic forms such as `if` and `cond` use *normal-order evaluation*:

```
> if
if: bad syntax in: if
> cond
cond: bad syntax in: cond
```

- The boolean operators `and` and `or` are also special syntactic forms and use *normal-order evaluation*:

```
> and
and: bad syntax in: and
> or
or: bad syntax in: or
>
```

- Arithmetic operators such as `+` and `>` are procedures (i.e., functions). Thus, like user-defined functions, they use *applicative-order evaluation*:

```
> +
#<procedure:+>
> >
#<procedure:>>
```

The Scheme syntactic forms `delay` and `force` permit the programmer to define and invoke functions that use normal-order evaluation. A consequence of this impurity is that programmers cannot extend (or modify) control structures (e.g.,

if, while, or for) in such languages using standard mechanisms (e.g., a user-defined function).

Why is lazy evaluation not more prevalent in programming languages? Certainly there is overhead involved in freezing and thawing thunks, but that overhead can be reduced with memoization (i.e., pass-by-need semantics) in the absence of side effects. In the presence of side effects, pass-by-need cannot be used. More importantly, in the presence of side effects, lazy evaluation renders a program difficult to understand. In particular, lazy evaluation generally makes it difficult to determine the flow of program control, which is essential to understanding a program with side effects. An attempt to conceptualize the control flow of a program with side effects using lazy evaluation requires digging deep into layers of evaluation, which is contrary to a main advantage of lazy evaluation—namely, modularity (Hughes 1989). Conversely, in a language with no side effects, flow of control has no effect on the result of a program. As a result, lazy evaluation is most common in languages without provisions for side effects (e.g., Haskell) and rarely found elsewhere.

Conceptual Exercises for Section 12.5

Exercise 12.5.1 Explain line 8 of the output in Section 12.5.3 (replicated here) of the first C program with a MAX macro:

The max of 10 and 101 is 102.

Exercise 12.5.2 Describe what problems might occur in a variety of situations if the MAX macro on line 14 of the first C program in Section 12.5.3 is defined as follows:

```
#define MAX(a,b) (a > b ? a : b)
```

(i.e., without each parameter in the replacement string enclosed in parentheses). Which uses of this macro would cause the identified problems to manifest? Explain.

Exercise 12.5.3 Consider the following swap macro using *pass-by-name* semantics defined on line 4 (replicated here) of the second C program in Section 12.5.3:

```
#define swap(a, b){ int temp = (a); (a)= (b); (b)= temp; }
```

For each of the following main programs in C, give the expansion of the swap macro in main and indicate whether the swap works.

(a)

```
int main() {
    int a[6];
    int i = 1;
    int j = 2;
    a[i] = 3;
    a[j] = 4;
    swap(a[i], a[j]);
}
```

(b)

```

int main() {
    int a[6];
    int i = 1;
    int j = 1;
    a[1] = 5;
    swap(i, a[j]);
}

```

Exercise 12.5.4 Consider the following C program:

```

1  #include<stdio.h>
2
3  /* swap macro: pass-by-name */
4  #define swap(x, y) { int temp = (x); (x) = (y); (y) = temp; }
5
6  int main() {
7
8      int x = 3;
9      int y = 4;
10     int temp = 5;
11
12     printf ("Before pass-by-name swap(x,temp) macro: x = %d, temp = %d\n",
13            x, temp);
14
15     swap(x, temp)
16
17     printf (" After pass-by-name swap(x,temp) macro: x = %d, temp = %d\n",
18            x, temp);
19 }

```

The preprocessed version of this program with the `swap` macro expanded is

```

1  int main() {
2
3      int x = 3;
4      int y = 4;
5      int temp = 5;
6
7      printf ("Before pass-by-name swap(x,temp) macro: x = %d, temp = %d\n",
8             x, temp);
9
10     { int temp = (x); (x) = (temp); (temp) = temp; }
11
12     printf (" After pass-by-name swap(x,temp) macro: x = %d, temp = %d\n",
13            x, temp);
14 }

```

The output of this program is

```

$ gcc collision.c
$ ./a.out
Before pass-by-name swap(x,temp) macro: x = 3, temp = 5
After pass-by-name swap(x,temp) macro: x = 3, temp = 5

```

While this (pass-by-name) `swap` macro works when invoked as `swap(x, y)` on line 14 in the second C program in Section 12.5.3, here it does not swap the

arguments—the values of `x` and `temp` are the same both before and after the code from the expanded `swap` macro executes. This outcome occurs because there is an identifier in the replacement string of the macro (line 4 of the unexpanded version) that is the same as the identifier for one of the variables being swapped, namely `temp`. When the macro is expanded in `main` (line 10), the identifier `temp` in `main` is used to refer to two different entities: the variable `temp` declared in `main` on line 5 and the local variable `temp` declared in the nested scope on line 10 (from the replacement string of the macro). The identifier `temp` in `main` *collides* with the identifier `temp` in the replacement string of the macro. What can be done to avoid this type of collision in general?

Exercise 12.5.5 Consider the following `f` macro using *pass-by-name* semantics:

```
#define f(x, y) { (x) = 1; (y) = 2; (x) = 2; (y) = 3; }
```

Consider the following `main` program in C that uses this macro:

```
int main() {
    int a[6];
    int i=0;
    f(i, a[i]);
}
```

Expand the `f` macro in `main` and give the values of `i` and `a[i]` before and after the statement `f(i, a[i])`.

Exercise 12.5.6 Consider the following `f` macro using *pass-by-name* semantics:

```
#define f(x, y, z) {int k = 1; (y) = (x); k = 5; (z) = (x); }
```

Consider the following `main` program in C that uses this macro:

```
int main() {
    int i=0;
    int j=0;
    int k=0;
    f(k+1, j, i);
}
```

Expand the `f` macro in `main` and give the values of `i`, `j`, and `k` before and after the statement `f(k+1, j, i)`.

Exercise 12.5.7 Consider the following `f` macro using *pass-by-name* semantics:

```
#define f(x) (x) + (x);
```

Consider the following `main` program in C that uses this macro:

```
int main() {
    f(read());
}
```

Assume the invocation of `read()` reads an integer from an input stream. Give the expansion of the `f` macro in `main`.

Exercise 12.5.8 In Section 12.5.3, we demonstrated that the expansion of macros defined in C/C++ using `#define` by the C preprocessor involves the string substitution in β -reduction. However, not all functions can be defined as macros in C. What types of functions do not lend themselves to definition as macros?

Exercise 12.5.9 Verify which semantics of lazy evaluation Racket uses through the `delay` and `force` syntactic forms: *pass-by-name* or *pass-by-need*. Specifically, modify the following Racket expression so that the parameters are evaluated lazily. Use the return value of the expression to determine which semantics of lazy evaluation Racket implements.

```
(let ((n 0))
  (let ((counter (lambda ()
                   ;; the function counter has a side effect
                   (set! n (+ n 1))
                   n)))
    ((lambda (x) (+ x x)) (counter))))
```

Given that Scheme makes provisions for side effects (through the `set!` operator), are the semantics of lazy evaluation that Scheme implements what you expected? Explain.

Exercise 12.5.10 Common Lisp uses *applicative-order evaluation* for function arguments. Is it prudent to treat the `if` expression in Common Lisp as a function or a syntactic form (i.e., not a function) and why?

The following is an example of an `if` expression in Common Lisp:
`(if (atom 'x) 'yes 'no).`

Exercise 12.5.11 The second argument to each of the Haskell built-in boolean operators `&&` and `||` is non-strict. Define the `(&&) :: Bool -> Bool -> Bool` and `(||) :: Bool -> Bool -> Bool` operators in Haskell.

Exercise 12.5.12 Consider the following definition of a function `f` defined using Python syntax:

```
def f(a, b):
    if a == 0:
        return 1
    else:
        return b
```

Is it advisable to evaluate `f` using *normal-order evaluation* or *applicative-order evaluation*? Explain and give your reasoning.

Exercise 12.5.13 Give an expression that returns different results when evaluated with *applicative-order evaluation* and *normal-order evaluation*.

Exercise 12.5.14 For each of the following programming languages, indicate whether the language uses *short-circuit evaluation* and give a program to unambiguously defend your answer.

(a) Common Lisp

(b) ML

Exercise 12.5.15 Lazy evaluation can be said to encapsulate other parameter-passing mechanisms. Depending on the particular type and form of an argument, lazy evaluation can simulate a variety of other parameter-passing mechanisms. For each of the following types of arguments, indicate which parameter-passing mechanism lazy evaluation is simulating. In other words, if each of the following types of arguments is passed by name, then the result of the function invocation is the same as if the argument was passed using which other parameter-passing mechanism?

(a) A scalar variable (e.g., x)

(b) A literal or an expression involving only literals [e.g., 3 or $(3 * 2)$]

Exercise 12.5.16 Recall that Haskell is a (nearly) pure functional language (i.e., provision for side effect only for I/O) that uses lazy evaluation. Since Haskell has no provision for side effect and pass-by-name and pass-by-need semantics yield the same results in a function without side effects, it is reasonable to expect that any Haskell interpreter would use pass-by-need semantics to avoid reevaluation of thunks. Since a provision for side effect is necessary to implement the *pass-by-need* semantics of lazy evaluation, can a self-interpreter for Haskell (i.e., an interpreter for Haskell written in Haskell) be defined? Explain. What is the implementation language of the *Glasgow Haskell Compiler*?

Programming Exercises for Section 12.5

Exercise 12.5.17 Rewrite the entire first Python program in Section 12.5.4 as a single Camille expression.

Exercise 12.5.18 Consider the following Scheme expression, which is an analog of the entire first Python program in Section 12.5.4:

```

1  (let ((counter (let ((n 0))
2                      (lambda ()
3                          ;; the function counter has a side effect
4                          (set! n (+ n 1)) ; n++
5                          n))))
6
7      (cons ((lambda (x) (+ x x)) (counter))
8            (cons ((lambda (x y) (+ x y)) (counter) (counter)) '())))

```

Rewrite this Scheme expression using the Scheme `delay` and `force` syntactic forms so that the arguments passed to the two anonymous functions on lines 7 and 8 are *passed by need*. The return value of this expression is `'(2 5)` using pass-by-need.

Exercise 12.5.19 The Scheme programming language uses *pass-by-value*. In this exercise, you implement lazy evaluation in Scheme. In particular, define a pair of functions, `freeze` and `thaw`, for forming and evaluating a thunk, respectively. The functions `freeze` and `thaw` have the following syntax:

```
;;; returns a thunk (or a promise)
(define freeze
  (lambda (expr)
    ...))

;;; returns result of evaluating thunk (or a promise)
(define thaw
  (lambda (thunk)
    ...))
```

The `thaw` and `freeze` functions are the Scheme analogs of the Python functions `force` and `delay` presented in Section 12.5.5. The `thaw` and `freeze` functions are also the user-defined function analogs of the Scheme built-ins `force` and `delay`, respectively.

In this implementation, an expression subject to lazy evaluation is not evaluated until its value is required; once evaluated, it is never reevaluated, (i.e., pass-by-need semantics). Specifically, the first time the thunk returned by `freeze` is thawed, it evaluates `expr` and remembers the return value of `expr` as demonstrated in Section 12.5.5. For each subsequent thawing of the thunk, the saved value of the expression is returned without any additional evaluation. Add print statements to the thunk formed by the `freeze` function, as done in Section 12.5.5, to distinguish between the first and subsequent evaluations of the thunk.

Examples:

```
1 > (define thunkarg (freeze '(+ 2 3)))
2 >
3 > ;; computes (+ 2 3) for the first time
4 > (thaw thunkarg)
5 first and only computation
6 5
7 > ;; does not recompute (+ 2 3); simply retrieves value 5
8 > (thaw thunkarg)
9 lookup, no recomputation
10 5
```

Be sure to quote the argument `expr` passed to `freeze` (line 1) to prevent it from being evaluating when `freeze` is invoked (i.e., eagerly). Also, the body of the thunk formed by the `freeze` function must invoke the Scheme function `eval` (as discussed in Section 8.2). So that the evaluation of the frozen expression has access to the base Scheme bindings (e.g., bindings for primitives such as `car` and `cdr`) and any other user-defined functions, place the following lines at the top of your program:

```
(define-namespace-anchor a)
(define ns (namespace-anchor->namespace a))
```

Then pass `ns` as the second argument to `eval` [e.g., `(eval expr ns)`]. See <https://docs.racket-lang.org/guide/eval.html> for more information on using Racket Scheme namespaces.

Exercise 12.5.20 (Scott 2006, Exercise 6.30, pp. 302–303) Use lazy evaluation through the syntactic forms `delay` and `force` to implement a lazy iterator object in Scheme. Specifically, an iterator is either the null list or a pair consisting of an element and a promise that, when forced, returns an iterator. Define an `uptoby` function that returns an iterator, and a `for-iter` function that accepts a one-argument function and an iterator as arguments and returns an empty list. The functions `for-iter` and `uptoby` enable the evaluation of the following expressions:

```
;; print the numbers from 1 to 10 in steps of 1, i.e., 1, 2, ..., 9, 10
(for-iter (lambda (e) (display e) (newline)) (uptoby 1 10 1))

;; print the numbers from 0 to 9 in steps of 1, i.e., 0, 1, ..., 8, 9
(for-iter (lambda (e) (display e) (newline)) (uptoby 0 9 1))

;; print the numbers from 1 to 9 in steps of 2, i.e., 1, 3, 5, 7, 9
(for-iter (lambda (e) (display e) (newline)) (uptoby 1 9 2))

;; print the numbers from 2 to 10 in steps of 2, i.e., 2, 4, 6, 8, 10
(for-iter (lambda (e) (display e) (newline)) (uptoby 0 10 2))

;; print the numbers from 10 to 50 in steps of 3, i.e., 10,13,...,47,50
(for-iter (lambda (e) (display e) (newline)) (uptoby 10 50 3))
```

The function `for-iter`, unlike the built-in Scheme form `for-each`, does not require the existence of a list containing the elements over which to iterate. Thus, the space required for `(for-iter f (uptoby 1 n 1))` is $O(1)$, rather than $O(n)$.

Exercise 12.5.21 Use lazy evaluation (`delay` and `force`) to solve Programming Exercise 5.10.12 (repeated here) in Scheme. Define a function `samefringe` in Scheme that accepts an integer `n` and two *S-expressions*, and returns `#t` if the first non-null `n` atoms in each *S-expression* are equal and in the same order and `#f` otherwise.

Examples:

```
> (samefringe 2 '(1 2 3) '(1 2 3))
#t
> (samefringe 2 '(1 1 2) '(1 2 3))
#f
> (samefringe 5 '(1 2 3 (4 5)) '(1 2 (3 4) 5))
#t
> (samefringe 5 '(1 ((2) 3) (4 5)) '(1 2 (3 4) 5))
#t
> (samefringe 5 '(1 6 3 (7 5)) '(1 2 (3 4) 5))
#f
> (samefringe 3 '(((1)) 2 (((3)))) '(((1) (((2)))) 3))
#t
> (samefringe 3 '(((1)) 2 (((3)))) '(((1) (((2)))) 4))
#f
> (samefringe 2 '((((a)) c)) '(((a) b)))
#f
```


Exercise 12.5.22 Solve Programming Exercise 5.10.12 (repeated here) in Haskell. Define a function `samefringe` in Haskell that accepts an integer `n` and two *S-expressions*, and returns `True` if the first non-null `n` atoms in each *S-expression* are equal and in the same order and `False` otherwise. Because of the homogeneous nature of lists in Haskell, we cannot use a list to represent an *S-expression* in Haskell. Thus, use the following definition of an *S-expression* in Haskell:

```
data Sexpr t = Nil
             | Atom t           -- an atom
             | List [Sexpr t]   -- or a list of S-expressions
             deriving (Show)
```

Examples:

```
Prelude> --           '(1 2 3) '(1 2 3)
Prelude> :{
Prelude> samefringe 2 (List [(Atom 1), (Atom 2), (Atom 3)])
Prelude|             (List [(Atom 1), (Atom 2), (Atom 3)])
Prelude| :}
True
Prelude> --           '(1 1 2) '(1 2 3)
Prelude> :{
Prelude> samefringe 2 (List [(Atom 1), (Atom 1), (Atom 2)])
Prelude|             (List [(Atom 1), (Atom 2), (Atom 3)])
Prelude| :}
False
Prelude> --           '(1 2 3 (4 5)) '(1 2 (3 4) 5)
Prelude> :{
Prelude> samefringe 5 (List [(Atom 1), (Atom 2), (Atom 3),
Prelude|                  (List [(Atom 4), (Atom 5)])])
Prelude|                  (List [(Atom 1), (Atom 2),
Prelude|                  (List [(Atom 3), (Atom 4)]), (Atom 5)])
Prelude| :}
True
Prelude> --           '(1 ((2) 3) (4 5)) '(1 2 (3 4) 5)
Prelude> :{
Prelude> samefringe 5 (List [(Atom 1),
Prelude|                  (List [(List [(Atom 2)]), (Atom 3)]),
Prelude|                  (List [(Atom 4), (Atom 5)])])
Prelude|                  (List [(Atom 1), (Atom 2),
Prelude|                  (List [(Atom 3), (Atom 4)]), (Atom 5)])
Prelude| :}
True
Prelude> --           '(1 6 3 (7 5)) '(1 2 (3 4) 5)
Prelude> :{
Prelude> samefringe 5 (List [(Atom 1), (Atom 6), (Atom 3),
Prelude|                  (List [(Atom 7), (Atom 5)])])
Prelude|                  (List [(Atom 1), (Atom 2),
Prelude|                  (List [(Atom 3), (Atom 4)]), (Atom 5)])
Prelude| :}
False
Prelude> --           '(((1)) 2 (((3)))) '((1) (((2)))) 3)
Prelude> :{
Prelude> samefringe 3 (List [(List [(List [(Atom 1)]), (Atom 2),
Prelude|                  (List [(List [(List [
Prelude|                  (List [(Atom 3)]))])])])])
Prelude|                  (List [(List [(Atom 1)]),
Prelude|                  (List [(List [(List [
```

```

Prelude|                                     (List [(List [(Atom 2)]))])))),
Prelude|                                     (Atom 3)])
Prelude| :}
True
Prelude> --                               '((1)) 2 (((3)))) '((1) (((2)))) 4)
Prelude> :{
Prelude> samefringe 3 (List [(List [(List [(Atom 1)])), (Atom 2),
Prelude|               (List [(List [(List [
Prelude|               (List [(Atom 3)]))]))]))))
Prelude|               (List [(List [(Atom 1)]), (List [(List [(List [
Prelude|               (List [(List [(Atom 2)]))]))]))]), (Atom 4)])
Prelude| :}
False
Prelude> --                               '((((a)) c))) '((a) b))
Prelude> :{
Prelude| samefringe 2 (List [(List [(List [(List [
Prelude|               (List [(Atom 'a')]))]), (Atom 'c')]))]))
Prelude|               (List [(List [(List [(Atom 'a')]), (Atom 'b')]))])
Prelude| :}
False

```

Exercise 12.5.23 Define the built-in Haskell function `iterate :: (a -> a) -> a -> [a]` as `iterate1` in Haskell. The `iterate` function accepts a unary function `f` with type `a -> a` and a value `x` of type `a`; it generates an (infinite) list by applying `f` an increasing number of times to `x` (i.e., `iterate f x = [x, (f x), f (f x), f (f (f x)), ...]`).

Examples:

```

Prelude> take 15 (iterate1 (2*) 1)
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384]
Prelude> take 5 (iterate1 sqrt 25)
[25.0,5.0,2.23606797749979,1.4953487812212205,1.2228445449938519]
Prelude> take 8 (iterate1 (1:) [])
[[], [1], [1,1], [1,1,1], [1,1,1,1], [1,1,1,1,1], [1,1,1,1,1,1], [1,1,1,1,1,1,1], [1,1,1,1,1,1,1,1]]

```

Exercise 12.5.24 Define the built-in Haskell function `filter :: (a -> Bool) -> [a] -> [a]` as `filter1` using list comprehensions (i.e., set-former notation) in Haskell. The `filter` function accepts a predicate [i.e., `(a -> Bool)`] and a list (i.e., `[a]`), in that order, and returns a list (i.e., `[a]`) filtered based on the predicate.

Examples:

```

Prelude> filter1 (>10) [100,3,101,500,5,9,10]
[100,101,500]
Prelude> filter1 even [100,3,101,500,5,9,10]
[100,500,10]
Prelude> filter1 (\x->length x>3) ["Est-ce", "que", "vous", "le", "voyez?"]
["Est-ce", "vous", "voyez."]

```

Exercise 12.5.25 Read John Hughes's essay "Why Functional Programming Matters" published in *The Computer Journal*, 32(2), 98–107, 1989, and available at <https://www.cse.chalmers.se/~rjmh/Papers/whyfp.html>. Read this article with

the *Glasgow Haskell Compiler* (GHC) open so you can enter the expressions as you read them, which will help you to better understand them. You will need to make some minor adjustments, such as replacing `cons` with `:`. The GHC is available at <https://www.haskell.org/ghc/>. Study Sections 1–3 of the article. Then implement one of the numerical algorithms from Section 4 in Haskell (e.g., *Newton-Raphson* square roots, numerical differentiation, or numerical integration). If you are interested in artificial intelligence, implement the search described in Section 5. Your code must run using GHCi—the interactive interpreter that is part of GHC.

12.6 Implementing Pass-by-Name/Need in Camille: Lazy Camille

We demonstrate how to modify the Camille interpreter supporting pass-by-reference from Section 12.4 so that it supports pass-by-name/need. To implement lazy evaluation in Camille, we extend the `Reference` data type with a third target variant: a thunk target. A thunk is the same as a direct target, except that it contains a thunk that *evaluates* to an expressed value, rather than containing an expressed value:

```

1  class Target:
2      def __init__(self, value, flag):
3
4          type_flag_dict = { "directtarget" : expressedvalue,
5                             "indirecttarget" : (lambda x : isinstance(x, Reference)),
6                             "frozen_expr" : (lambda x : isinstance(x, list)) }
7
8          # if flag is not a valid flag value, construct a lambda expression
9          # that always returns false so we throw an error
10         type_flag_dict = \
11             defaultdict (lambda: lambda x: False, type_flag_dict)
12
13         if (type_flag_dict[flag] (value)):
14             self.flag = flag
15             self.value = value
16         else:
17             raise Exception("Invalid Target Construction.")

```

Note that we added `frozen_expr` flag to the dictionary of possible target types.

If the dereference function is passed a reference containing a thunk, it evaluates the thunk using the `thaw_thunk` function. This function evaluates the expression in the thunk and returns the corresponding value:

```

1  def dereference(self):
2      target = self.primitive_dereference()
3
4      if target.flag == "directtarget":
5          return target.value
6      elif target.flag == "indirecttarget":
7          innertarget = target.value.primitive_dereference()
8
9          if innertarget.flag == "directtarget":

```

```

10         return innertarget.value
11
12     elif innertarget.flag == "frozen_expr":
13         return target.value.thaw_thunk()
14
15     elif target.flag == "frozen_expr":
16         return self.thaw_thunk()
17
18     raise Exception("Invalid dereference.")
19
20 def thaw_thunk(self):
21
22     # self.vector[self.position].frozen_expr[1] is the root of the tree
23     # self.vector[self.position].frozen_expr[1] is environment
24     #                                     at time of call
25     # print ("Thaw")
26
27     if (camilleconfig.__lazy_switch__ == camilleconfig.pass_by_name):
28         return evaluate_expr(self.vector[self.position].value[0],
29                             self.vector[self.position].value[1])
30
31     elif (camilleconfig.__lazy_switch__ == camilleconfig.pass_by_need):
32         # the first time we evaluate the thunk we save the result
33         if isinstance(self.vector[self.position].value, list):
34             self.vector[self.position].value = evaluate_expr(
35                 self.vector[self.position].value[0],
36                 self.vector[self.position].value[1])
37             self.vector[self.position].flag = "directtarget"
38         return self.vector[self.position].value
39
40     else:
41         raise Exception("Configuration Error.")

```

When dereferencing a reference (lines 1–18), we now must handle the case where the target is a `frozen_expr` (lines 12–13 and 15–16). We thaw the thunk when it is frozen by evaluating the saved tree in the saved environment with the `thaw_thunk` function (lines 20–41). The switch `camilleconfig.__lazy_switch__` accessed on lines 27 and 31 is set prior to run-time to specify the implementation of lazy evaluation as either pass-by-name or pass-by-need (lines 31–38). If we use pass-by-name semantics, the `thaw_thunk` function evaluates the saved tree in the saved environment and returns the result (lines 27–29). If we use pass-by-need semantics, the `thaw_thunk` function must update the location containing the thunk to store a direct target with the expressed value the first time the thunk is thawed (lines 33–37). The function simply retrieves and returns the saved expressed value on each subsequent reference to the same parameter (line 38).

We must also replace line 48 in the definition of the `assignreference` function starting in Section 12.4.1 with the following line:

```

48 if target.flag == "directtarget" or target.flag == "frozen_expr":

```

A target may be a `frozen_expr` during assignment. Thus, we must treat a `frozen_expr` the same way as a `directtarget`. We must also replace the `ntArguments` case in the `evaluate_expression` function

```

elif expr.type == ntArguments:
    ArgList = []
    ArgList.append(evaluate_expr(expr.children[0], environ))

    if len(expr.children) > 1:
        ArgList.extend(evaluate_expr(expr.children[1], environ))
    return ArgList

```

with

```

elif expr.type == ntArguments:
    return freeze_function_arguments(expr.children, environ)

```

The `freeze_function_arguments` freezes the function arguments rather than evaluating them.

```

1 def freeze_function_arguments(arg_tree, environ):
2     argument_list = []
3     if (arg_tree[0].type == ntNumber or arg_tree[0].type == ntIdentifier):
4         argument_list.append(evaluate_expr(arg_tree[0], environ))
5     else:
6         argument_list.append([arg_tree[0], environ])
7
8     if (len(arg_tree) > 1):
9         argument_list.extend(freeze_function_arguments(arg_tree[1].children,
10                                                         environ))
11     return argument_list

```

This function recurses argument lists. However, now only literals and identifiers are evaluated. The root `TreeNode` of every other expression is saved into a list with the corresponding environment to be evaluated later. Lastly, we must update the `evaluate_operand` function:

```

1 def evaluate_operand(operand, environ):
2
3     if isinstance(operand, Reference):
4
5         ## if the operand is a variable, then it denotes a location
6         ## containing an expressed value; thus,
7         ## we return an "indirect target" pointing to that location
8         target = operand.primitive_dereference()
9
10        ## if the variable is bound to a "location" that
11        ## contains a direct target,
12
13        if target.flag == "directtarget":
14
15            ## then we return an indirect target to that location
16            return Target(operand, "indirecttarget")
17
18        ## but if the variable is bound to a "location"
19        ## that contains an indirect target, then
20        ## we return the same indirect target
21
22        elif target.flag == "indirecttarget":
23            innertarget = target.value.primitive_dereference()
24            if innertarget.flag == "indirecttarget":
25
26                # double indirect references not allowed

```

```

27         return Target(innertarget, "indirecttarget")
28     else:
29         return innertarget
30
31     elif target.flag == "frozen_expr":
32         return Target(operand, "indirecttarget")
33
34     ## if the operand is a literal (i.e., integer or function/closure),
35     ## then we create a new location, as before, by returning
36     ## a "direct target" to it (i.e., pass-by-value)
37
38     elif isinstance(operand, int) or is_closure(operand):
39         return Target(operand, "directtarget")
40
41     elif isinstance(operand, list):
42         return Target(operand, "frozen_expr")

```

Because a target can now contain a `frozen_expr` (i.e., an expression that has yet to be evaluated), we need to handle the case where an operand is a `frozen_expression` (lines 31–32). Also, the single argument to a function is passed to the `evaluate_operand` function as a `[expression_tree, environment]` list. In this case, we want to freeze that function argument and not evaluate the `expression_tree` (lines 41–42).

Lines 1–29 and 34–39 of this definition of the `evaluate_operand` function constitute the entire `evaluate_operand` function used in the pass-by-reference Camille interpreter shown in Section 12.4.2. The new lines of code in this definition are lines 31–32 and 41–42. Let us unpack the two cases of operands handled in this function:

- If the operand is a *variable* (i.e., `ntIdentifier`) that points to a *thunk target*, then return an *indirect target* to it (lines 31–32).
- If the operand is an *expression* (i.e., `ntExpression`), then return a *thunk target* containing the expression operand (lines 41–42).

Examination of this definition of `evaluate_operand` reveals that this version of Camille uses three different types of parameter-passing mechanisms:

- *pass-by-value* for *literal* arguments (i.e., numbers and functions/closures) (lines 34–39)
- *pass-by-value* for all operands to primitives operations (e.g., `+`)
- *pass-by-reference* for *variable* arguments (lines 10–29)
- *pass-by-need/normal-order evaluation* for everything else (i.e., expressions involving literals and/or variables) (lines 31–32 and 41–42)

We also add a division primitive, which is used in Camille programs demonstrating lazy evaluation. To do so, we add `"/" : operator.floordiv` to `primitive_op_dict`. We use floor division because all numbers in Camille are integers and should be represented as such in the implementing language (i.e., Python). In addition, we must add the `DIV` token to the definition of the `p_primitive` function in the parser specification.

Programming Exercise	Camille	Description	Start from	Representation of Closures	Representation of Environment
12.4.1	3.0(pass-by-value-result)	pass-by-value-result	3.0	ASR CLS	ASR
12.6.1	3.2(lazy let)	lazy let	3.2	ASR CLS	ASR
12.6.2	3.2(full lazy)	full lazy	3.2(lazy let)	ASR CLS	ASR
12.7.1	4.0(do while)	do while	4.0	ASR CLS	ASR

Table 12.5 New Versions of Camille, and Their Essential Properties, Created in Sections 12.6 and 12.7 Programming Exercises (Key: ASR = abstract-syntax representation; CLS = closure.)

Example:

```

1    --- 15 20 are passed by value
2    (fun (a,b)
3      --- +(a,b) passed by need
4        (fun (x)
5          --- +(a,b)
6            (fun (y) passed by need
7              --- +(a,b) passed by need
8                (fun (z) +(x,y), z) y)
9              x)
10         +(a,b))
11 15,20)
```

The evaluation of the operand expression $+(a, b)$ passed on line 10 is delayed until referenced. That operand is referenced as x , y , and z in the expression $+(+(x, y), z)$ on line 8. Since we are using pass-by-need semantics, the operand expression $+(a, b)$ will be evaluated only once—when x is referenced in the expression $+(+(x, y), z)$ on line 8. When the operand expression $+(a, b)$ is referenced as y and z in the expression $+(+(x, y), z)$ on line 8, it will refer to the already-evaluated thunk.

Table 12.5 summarizes the properties of the new versions of the Camille interpreter developed in the Programming Exercises in Sections 12.6 and 12.7.

Programming Exercises for Section 12.6

Exercise 12.6.1 (Friedman, Wand, and Haynes 2001, Exercise 3.58, p. 118) Implement Lazy Camille—the pass-by-need Camille interpreter (version 3.2) described in this section. Then extend it so that the bindings created in `let` expressions take place lazily.

Example:

```

Camille> let
          a = /(1,0)
        in
          2
```

Exercise 12.6.2 (Friedman, Wand, and Haynes 2001, Exercise 3.56, p. 117) Extend the solution to Programming Exercise 12.6.1 so that arguments to primitive operations are evaluated lazily. Then, implement `if` as a primitive instead of a syntactic form. Also, add a division (i.e., `/`) primitive to Camille so the lazy Camille interpreter can evaluate the following programs demonstrating lazy evaluation:

```
Camille> if (zero? (1), 10, 11)

11
Camille> let
  p = fun (x, y)
      if (zero? (x), x, y)
  in
  (p 0,4)

0
Camille> let
  d = fun (x, y) / (x,y)
  p = fun (x, y)
      if (zero? (x), 10, y)
  in
  (p 0, / (1,0))

10
```

12.7 Sequential Execution in Camille

Although Camille has a provision for variable assignment, an entire Camille program must be expressed as a single Camille expression—there is no concept of sequential evaluation in Camille. We now extend the interpreter to morph Camille into a language that supports a synthesis of expressions and statements. To syntactically support statements that are sequentially executed in Camille, we add both the following rules to the grammar and the corresponding pattern-action rules to the PLY parser generator:

```
<program> ::= <statement>

ntAssignmentStmt
<statement> ::= <identifier> = <expression>

ntOutputStmt
<statement> ::= writeln (<expression>)

ntCompoundStmt
<statement> ::= { {<statement>}*(:) }

ntIfElseStmt
<statement> ::= if <expression> <statement> else <statement>
```


ntWhileStmt

<statement> ::= while <expression> do <statement>

ntBlockStmt

<statement> ::= variable {<identifier>}*(^(,)) ; <statement>

```

1  def p_line_stmt(t):
2      '''program : statement'''
3      t[0] = t[1]
4      execute_stmt(t[0], empty_environment())
5
6  def p_statement_assignment(t):
7      '''statement : IDENTIFIER EQ functional_expression'''
8      t[0] = Tree_Node(ntAssignmentStmt, [t[3]], t[1], t.lineno(1))
9
10 def p_statement_writeln(t):
11     '''statement : WRITELN LPAREN functional_expression RPAREN'''
12     t[0] = Tree_Node(ntOutputStmt, [t[3]], None, t.lineno(1))
13
14 def p_statement_compound(t):
15     '''statement : LCURL statement_list RCURL
16                | LCURL RCURL'''
17     if len(t) == 4:
18         t[0] = Tree_Node(ntCompoundStmt, [t[2]], None, t.lineno(1))
19     else:
20         t[0] = Tree_Node(ntCompoundStmt, [None], None, t.lineno(1))
21
22 def p_statement_list(t):
23     '''statement_list : statement SEMICOLON statement_list
24                      | statement'''
25     if len(t) > 2:
26         t[0] = Tree_Node(ntStmtList, [t[1], t[3]], None, t.lineno(1))
27     else:
28         t[0] = Tree_Node(ntStmtList, [t[1]], None, t.lineno(1))
29
30 def p_statement_if(t):
31     '''statement : IF functional_expression statement ELSE statement'''
32     t[0] = Tree_Node(ntIfElseStmt, [t[3],t[5]], t[2], t.lineno(1))
33
34 def p_statement_while(t):
35     '''statement : WHILE functional_expression DO statement'''
36     t[0] = Tree_Node(ntWhileStmt, [t[4]], t[2], t.lineno(1))
37
38 def p_statement_block(t):
39     '''statement : VARIABLE id_list SEMICOLON statement
40                | VARIABLE SEMICOLON statement'''
41     if len(t) == 5:
42         t[0] = Tree_Node(ntBlockStmt, [t[2],t[4]], None, t.lineno(1))
43     else:
44         t[0] = Tree_Node(ntBlockStmt, [None,t[3]], None, t.lineno(1))
45
46 def p_identifier_list(t):
47     '''id_list : IDENTIFIER COMMA id_list
48               | IDENTIFIER'''
49     if len(t) > 2:
50         t[0] = Tree_Node(ntIdList, [t[1], t[3]], None, t.lineno(1))
51     else:
52         t[0] = Tree_Node(ntIdList, [t[1]], None, t.lineno(1))
53

```

```

54 def p_functional_expression(t):
55     '''functional_expression : expression'''
56     t[0] = Tree_Node(ntFunctionalExpression, None, t[1], t.lineno(1))

```

The informal semantics of this version of Camille are summarized here:

- A Camille program is now a *statement*, not an *expression*.
- A Camille program now functions by *executing a statement*, not by *evaluating an expression*.
- A Camille program now functions by *printing*, not by *returning* a value.
- All else is the same as in Camille 3.0.

Statements are executed for their (side) effect, not their value. The following are some example Camille programs involving statements:

```

Camille> variable x, y, z; {
    x = 1;
    y = 2;
    z = +(x,y);
    writeln (z)
}

3
Camille> variable i, j, k; {
    i = 3;
    j = 2;
    k = 1;
    writeln +(i, -(j,k))
}

4
Camille> if 1
    if 0
        writeln(5)
    else
        writeln(6)
else
    writeln(7)

6
Camille> --- while loop: 1 .. 5
Camille> variable i, j; {
    i = 1;
    j = 5;
    while j do {
        writeln(i);
        j = decl(j);
        i = incl(i)
    }
}

1
2
3
4
5
Camille> --- an alternate while loop: 1 .. 5
Camille> variable i; {
    i = 1;

```

```

        while if eqv?(i,6) 0 else 1 do {
            writeln(i);
            i = incl(i)
        }
    }

1
2
3
4
5
Camille> --- nested blocks and scoping
Camille> variable i; {
    i = 2;
    writeln(i);

    variable j; {
        j = 1;
        writeln(j)
    };

    writeln(i)
}

2
1
2
Camille> --- nested blocks and a scope hole
Camille> variable i; {
    i = 1;
    writeln(i);

    variable i; {
        i = 3;
        writeln(i)
    };

    writeln(i)
}

1
3
1
Camille> --- use of statements and expressions
Camille> variable increment, i; {
    increment = fun(n) incl(n);
    i = 0;
    writeln ((increment i))
}

12

```

Notice that **;** is the statement separator, not the statement terminator:

```

Camille> if 1 {
    if 0 {
        writeln(5)
    } else {
        writeln(6);
        writeln(6)
    }
}

```

```

    } else {
        writeln(7)
    }

```

Although some statements, including `while`, `if`, `=`, and `writeln` [e.g., `writeln (let i = 1 in i)`], syntactically permit the use of an expression, statements and expressions cannot be used interchangeably. For instance, the following program is valid:

```

Camille> --- syntactically correct use of statements and expressions
Camille> variable increment_and_print, i; {
    increment_and_print = fun(n) incl(n);
    i = 0;
    writeln((increment_and_print i))
}

```

1

However, the following conceptually equivalent program is not syntactically valid:

```

Camille> --- syntactically incorrect use of statements and expressions
Camille> variable increment_and_print, i; {
    increment_and_print = fun(n) writeln(incl(n));
    i = 0;
    (increment_and_print i);
}

```

Syntax **error**: Line 2

We must define an `execute_stmt` function to run programs like those just shown here:

```

1  def execute_stmt(stmt, environ):
2
3      try:
4          if stmt.type == ntAssignmentStmt:
5
6              tempref = apply_environment_reference(environ, stmt.leaf)
7              temp = execute_stmt(stmt.children[0], environ)
8              tempref.assignreference(temp)
9
10         elif stmt.type == ntOutputStmt:
11             print (execute_stmt (stmt.children[0], environ))
12
13         elif stmt.type == ntCompoundStmt:
14             execute_stmt(stmt.children[0], environ)
15
16         elif stmt.type == ntIfElseStmt:
17             if execute_stmt (stmt.leaf, environ):
18                 execute_stmt (stmt.children[0], environ)
19             else:
20                 execute_stmt (stmt.children[1], environ)
21
22         elif stmt.type == ntWhileStmt:
23             while execute_stmt (stmt.leaf, environ):
24                 execute_stmt (stmt.children[0], environ)

```

```

25
26     elif stmt.type == ntBlockStmt:
27
28         # building id list
29         IdList = execute_stmt(stmt.children[0], environ)
30
31         ListofZeros = list (map (lambda identifier: 0, IdList))
32
33         TargetListofZeros = list (map (evaluate_let_expr_operand,
34                                     ListofZeros))
35
36         localenv = extend_environment(IdList, TargetListofZeros, environ)
37
38         execute_stmt(stmt.children[1], localenv)
39
40     elif stmt.type == ntStmtList:
41         execute_stmt(stmt.children[0], environ)
42
43         if len(stmt.children) > 1:
44             execute_stmt(stmt.children[1], environ)
45
46     elif stmt.type == ntIdList:
47         IdList = []
48         IdList.append(stmt.children[0])
49
50         if len(stmt.children) > 1:
51             IdList.extend(execute_stmt(stmt.children[1], environ))
52         return IdList
53
54     elif stmt.type == ntFunctionalExpression:
55         t = evaluate_expr(stmt.leaf, environ)
56         return localbindingDereference(t)
57
58     else:
59         raise InterpreterException(expr.linenumber,
60                                   "Invalid tree node type %s" % expr.type)
61
62 except Exception as e:
63     if(isinstance(e, InterpreterException)):
64         # raise exception to the next level until we reach the top level
65         # of the interpreter; exceptions are fatal for a single tree,
66         # but other programs within a single file may
67         # otherwise be OK
68         raise e
69     else:
70         # we want to catch the Python interpreter exception and
71         # format it such that it can be used to debug the Camille program
72         if(debug_mode__ == detailed_debug):
73             print(traceback.format_exc())
74         raise InterpreterException(expr.linenumber,
75                                   "Unhandled error in %s" % expr.type, str(e), e)

```

The `execute_stmt` function is called from the action section of the `p_line_stmt` pattern-action rule (line 4 in the first listing). Notice in the `execute_stmt` function that, unlike prior versions of the interpreter, we rely on the (imperative) features of Python to build these new (imperative) constructs into Camille (discussed in Section 12.8). For instance, we implement a `while` loop into Camille not by building it from first principles, but rather by directly using the `while` loop in Python (lines 22–24).

Programming Exercise for Section 12.7

Exercise 12.7.1 (Friedman, Wand, and Haynes 2001, Exercise 3.63, p. 121) Add a `do-while` statement to the Camille interpreter developed in this section. A `do-while` statement operates like a `while` statement, except that the test is evaluated *after* the execution of the body, not before.

```
Camille> variable x,y; {
    x = 0;
    y = 1;

    do {
        writeln(x)
    } while x;

    writeln(y)
}

0
1
Camille> variable x, y; {
    x = 11;
    y = -(11,4);

    do {
        y = +(y,x);
        x = decl(x)
    } while x;

    writeln (y)
}
```

73

12.8 Camille Interpreters: A Retrospective

Figure 12.12 illustrates the dependencies between the versions of Camille developed in this chapter. Table 12.6 and Figure 12.13 present the dependencies between the versions of Camille developed in this text. Table 12.7 summarizes the versions of the Camille interpreter developed in this text. The presence of downward arrows in some of the cells in Table 12.7 indicates that the concept indicated by the cell is supported through its implementation in the defining language. Notice that reusing the implementation of concepts in the defining or implementation language limits what is possible in the language being interpreted. “Thus, for example, if the control frame structure in the implementation language is constrained to be stack-like, then modeling more general control structures in the interpreted language will be very difficult unless we divorce ourselves from the constrained structures at the outset” (Sussman and Steele 1975, p. 28).

Table 12.8 outlines the configuration options available in Camille for aspects of the design of the interpreter (e.g., choice of representation of referencing environment), as well as for the semantics of implemented concepts (e.g., choice of parameter-passing mechanism). As we vary the latter, we get a different version

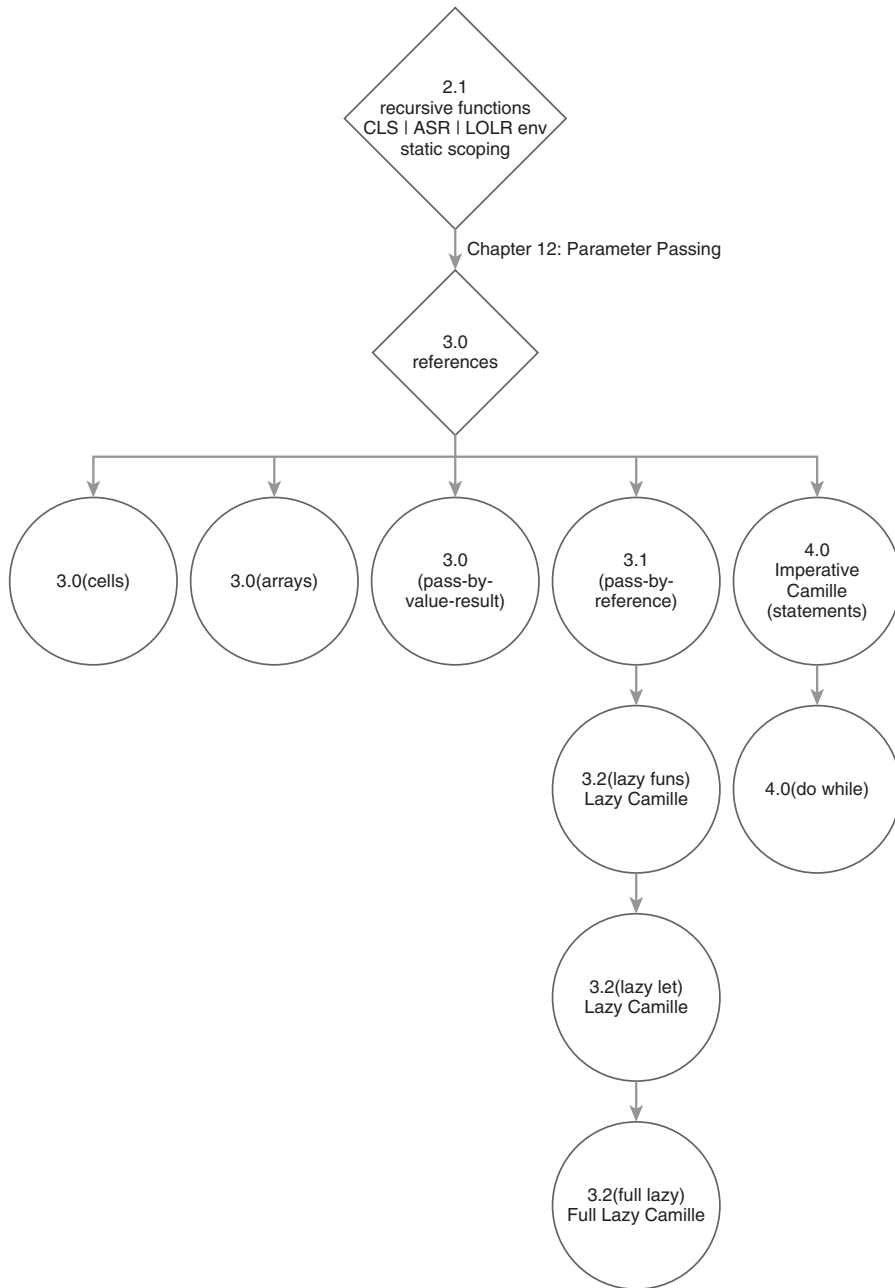


Figure 12.12 Dependencies between the Camille interpreters developed in this chapter, including those in the programming exercises. The semantics of a directed edge $a \rightarrow b$ are that version b of the Camille interpreter is an extension of version a (i.e., version b subsumes version a). (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

Version	Extends	Description
Chapter 10: Local Binding and Conditional Evaluation		
1.0	N/A	simple, no environment
1.1	1.0	let, named CLS ASR LOLR environment
1.1(named CLS)	1.1	let, named CLS environment
1.2	1.1	let, if
1.2(named CLS)	1.2	let, if/else, named CLS environment
1.2(named ASR)	1.2	let, if/else, named ASR environment
1.2(named LOLR)	1.2	let, if/else, named LOLR environment
1.2(named CLS)	1.2	let, if/else, nameless CLS environment
1.2(named ASR)	1.2	let, if/else, nameless ASR environment
1.2(named LOLR)	1.2	let, if/else, nameless LOLR environment
1.3	1.2	let*, if/else, (named nameless) (CLS ASR LOLR) environment
Chapter 11: Functions and Closures		
Non-recursive Functions		
2.0	1.2	fun, CLS ASR LOLR environment
2.0(verify ASR)	2.0	fun, verify ASR environment
2.0(nameless ASR)	2.0(verify ASR)	fun, nameless ASR environment
2.0(verify LOLR)	2.0	fun, verify LOLR environment
2.0(nameless LOLR)	2.0(verify LOLR)	fun, nameless LOLR environment
2.0(verify CLS)	2.0	fun, verify CLS environment
2.0(nameless CLS)	2.0(verify CLS)	fun, nameless CLS environment
2.0(dynamic scoping)	2.0	fun, dynamic scoping, (named nameless) (CLS ASR LOLR) environment
Recursive Functions		
2.1	2.0	letrec, CLS ASR LOLR environment
2.1(named CLS)	2.0	letrec, named CLS environment
2.1(nameless CLS)	2.0(nameless CLS) or 2.1(named CLS)	letrec, nameless CLS environment
2.1(named ASR)	2.0	letrec, named ASR environment
2.1(nameless ASR)	2.0(nameless ASR) or 2.1(named ASR)	letrec, nameless ASR environment
2.1(named LOLR)	2.0	letrec, named LOLR environment
2.1(nameless LOLR)	2.0(nameless LOLR) or 2.1(named LOLR)	letrec, nameless LOLR environment
2.1(dynamic scoping)	2.0(dynamic scoping) or 2.1	letrec, dynamic scoping, (named nameless) (CLS ASR LOLR) environment
Chapter 12: Parameter Passing		
3.0	2.1	references, named ASR environment, ASR CLS closure
3.0(cells)	3.0	cells, named ASR environment, ASR CLS closure
3.0(arrays)	3.0	arrays, named ASR environment, ASR CLS closure
3.0(pass-by-value-result)	3.0	pass-by-value-result, named ASR environment, ASR CLS closure
3.1	3.0	pass-by-reference, named ASR environment, ASR CLS closure
Lazy Camille		
3.2(lazy funs)	3.1	lazy evaluation for fun args only, named ASR environment, ASR CLS closure
3.2(lazy let)	3.2	lazy evaluation for fun args and let expr, named ASR environment, ASR CLS closure
3.2(full lazy)	3.2	lazy evaluation for fun args, let expr, and primitives, named ASR environment, ASR CLS closure
Imperative Camille		
4.0	3.0	statements, named ASR environment, ASR CLS closure
4.0(do while)	4.0	do while, named ASR environment, ASR CLS closure

Table 12.6 Complete Suite of Camille Languages and Interpreters (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

Data from Perugini, Saverio, and Jack L. Watkin. 2018. "ChAMElEoN: A customizable language for teaching programming languages." *Journal of Computing Sciences in Colleges (USA)* 34(1): 44–51.

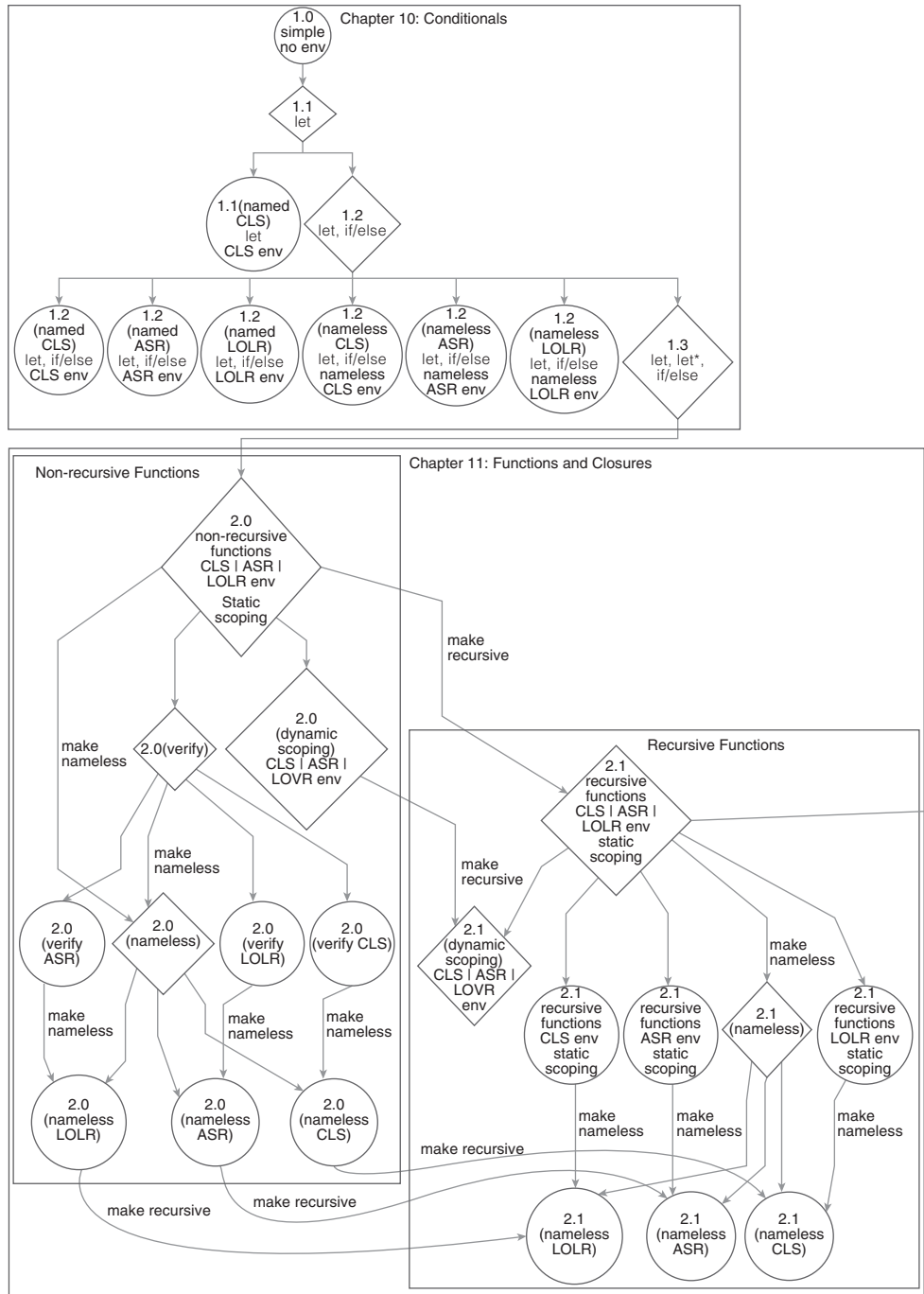


Figure 12.13 Dependencies between the Camille interpreters developed in this text, including those in the programming exercises. The semantics of a directed edge $a \rightarrow b$ are that version b of the Camille interpreter is an extension of version a (i.e., version b subsumes version a). (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

of the language (Table 12.7). (Note that the nameless environments are available for use with neither the interpreter supporting dynamic scoping nor any of the interpreters in this chapter. Furthermore, not all environment representations are available with all implementation options. For instance, all of the interpreters in this chapter use exclusively the named ASR environment.)

Conceptual and Programming Exercises for Section 12.8

Exercise 12.8.1 Compiled programs run faster than interpreted ones. Reflect on the Camille interpreters you have built in this text. What is the bottleneck in an interpreter that causes an interpreted program to run orders of magnitude slower than a compiled program?

Exercise 12.8.2 Write a Camille program using any valid combination of the features and concepts covered in Chapters 10–12 and use it to stress test—in other words, spin the wheels of—the Camille interpreter. Your program must be at least 30 lines of code and original (i.e., not an example from the text). You are welcome to rewrite a program you wrote in the past and use it to flex the muscles of your interpreter. For instance, you can use Camille to build a closure representation

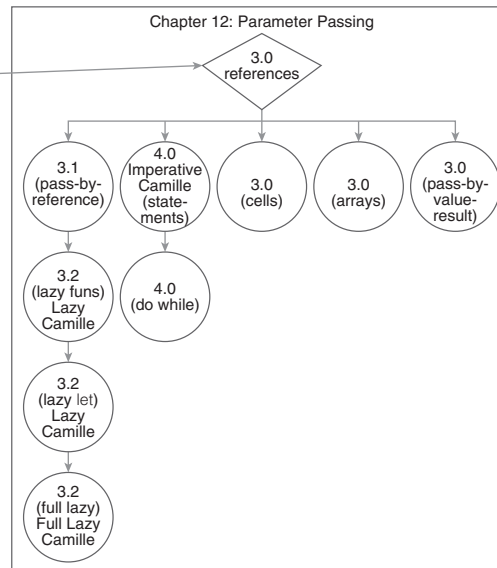


Figure 12.13 (*Continued.*) (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation.)

Version of Camille	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.1	3.2	4.0
Expressed Values	integers	integers	integers	integers	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$
Denoted Values	integers	integers	integers	integers	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$	$\text{integers} \cup \text{cls}$
Representation of Environment	N/A	ASR CLS LOLR	ASR CLS LOLR	ASR CLS LOLR	ASR CLS LOLR	ASR CLS LOLR	ASR	ASR	ASR	ASR
Representation of Closures	N/A	N/A	N/A	N/A	ASR CLS	ASR CLS	ASR CLS	ASR CLS	ASR CLS	ASR CLS
Representation of References	N/A	N/A	N/A	N/A	N/A	N/A	ASR	ASR	ASR	ASR
Local Binding	×	↑ let ↑	↑ let ↑	↑ let, let* ↑	↑ let, let* ↑	↑ let, let* ↑	↑ let, let* ↑	↑ let, let* ↑	↑ let, let* ↑	↑ let, let* ↑
Conditionals	×	×	↓ if/else ↓	↓ if/else ↓	↓ if/else ↓	↓ if/else ↓	↓ if/else ↓	↓ if/else ↓	↓ if/else ↓	↓ if/else ↓
Non-recursive Functions	×	×	×	×	↑ fun ↑	↑ fun ↑	↑ fun ↑	↑ fun ↑	↑ fun ↑	↑ fun ↑
Recursive Functions	×	×	×	×	×	↑ letrec ↑	↑ letrec ↑	↑ letrec ↑	↑ letrec ↑	↑ letrec ↑
Scoping	N/A	lexical	lexical	lexical	lexical	lexical	lexical	lexical	lexical	lexical
Environment Binding to Closure	N/A	N/A	N/A	N/A	deep	deep	deep	deep	deep	deep
References	×	×	×	×	×	×	√	√	√	√
Parameter Passing	N/A	N/A	N/A	N/A	↑ by value ↑	↑ by value ↑	↑ by value ↑	↑ by reference ↑	↑ lazy evaluation ↑	↑ by value ↑
Side Effects	×	×	×	×	×	×	↑ assign! ↑	↑ assign! ↑	↑ assign! ↑	↓ multiple ↓
Statement Blocks	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	↓ {} ↓
Repetition	N/A	N/A	N/A	N/A	N/A	recursion	recursion	recursion	recursion	↓ while ↓

Concepts/Data Structures

Table 12.7 Concepts and Features Implemented in Progressive Versions of Camille. The symbol ↓ indicates that the concept is supported through its implementation in the defining language (here, Python). The Python keyword included in each cell, where applicable, indicates which Python construct is used to implement the feature in Camille. The symbol ↑ indicates that the concept is implemented manually. The Camille keyword included in each cell, where applicable, indicates the syntactic construct through which the concept is operationalized. (Key: ASR = abstract-syntax representation; CLS = closure; LOLR = list-of-lists representation. Cells in boldface font highlight the enhancements across the versions.)

Interpreter Design Options			Language Semantic Options		
Type of Environment	Representation of Environment	Representation of Functions	Scoping Method	Environment Binding	Parameter-Passing Mechanism
named	abstract syntax	abstract syntax	static	deep	by value
nameless	list of lists	closure	dynamic		by reference
	closure				by value-result
					by name (lazy evaluation)
					by need (lazy evaluation)

Table 12.8 Complete Set of Configuration Options in Camille

of a stack or queue or a converter from decimal to binary numbers. If you like, you can add new primitives to the language and interpreter. Your program will be evaluated based on the use of novel language concepts implemented in the Camille interpreter (e.g., dynamic scoping, recursion, lazy evaluation) and the creativity of the program to solve a problem.

12.9 Metacircular Interpreters

After having explored language semantics by implementing multiple versions of Camille, we would be remiss not to make some brief remarks about self- and metacircular interpreters. Multiple approaches may be taken to define language semantics through interpreter implementation (Table 12.9). The approach here has been to implement Camille in Python. While we were able to define semantics in Camille by simply relying upon the semantics of the same concepts in Python (note all the downward arrows in Table 12.7), the reuse in the interpreter involves two different programming languages.

A *self-interpreter* is an interpreter implemented in the same language as the language being interpreted—that is, where the defined and defining languages are the same. Smalltalk is implemented as a self-interpreter.⁴ An advantage of a self-interpreter is that the language features being built into the defined language

Language Implementation	Defining Language	Defined Language	Example
Interpreter	$\mathcal{X}$	$\mathcal{Y}$	Camille interpreter in Python
Self-Interpreter	$\mathcal{L}$	$\mathcal{L}$	Smalltalk
<i>Advantage:</i> can restate language features in terms of themselves.			
Metacircular Interpreter	$\mathcal{L}_{homoiconic}$	$\mathcal{L}_{homoiconic}$	Lisp
<i>Advantage:</i> no need to convert between concrete and abstract representations.			

Table 12.9 Approaches to Learning Language Semantics Through Interpreter Implementation

4. The *System Browser* in the *Squeak* implementation of Smalltalk catalogs the source code for the entire Smalltalk class hierarchy.

that are borrowed from the defining language can be more directly and, therefore, easily expressed in the interpreter—language concepts can be restated in terms of themselves! (Sometimes this is called *bootstrapping* a language.) A more compelling benefit of this direct correspondence between host and source language results when, conversely, we do not implement features in the defined language using the same semantics as in the defining language. In that case, a self-interpreter is an avenue toward *modifying* language semantics in a programming language. By implementing pass-by-name semantics in Camille, we did not alter the parameter-passing mechanism of Python. However, if we built an interpreter for Python in Python, we could.

A self-interpreter for a *homoiconic* language—one where programs and data objects in the language are represented uniformly—is called a *metacircular interpreter*. While a metacircular interpreter is a self-interpreter—and, therefore, has all the benefits of a self-interpreter—since the program being interpreted in the defined language is expressed as a data structure in the defining language, there is no need to convert between concrete and abstraction representations. For instance, the `concrete2abstract` (in Section 9.6) and `abstract2concrete` (in Programming Exercise 9.6.1) functions from Chapter 9 are unnecessary.

Thus, the homoiconic property simplifies the ability to change the semantics of a language from within the language itself! This idea supports a *bottom-up style of programming* where a programming language is used not as a tool to write a target program, but to define a new targeted (or domain-specific) language and then develop a target program in that language (Graham 1993, p. vi). In other words, bottom-up programming involves “changing the language to suit the problem” (Graham 1993, p. 3)—and that language can look quite a bit different than Lisp. (See Chapter 15 for more information.) It has been said that “[i]f you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases” (Friedman and Felleisen 1996b, Afterword, p. 207, Guy L. Steele Jr.) and “Lisp is a language for writing Lisp.” Programming Exercise 5.10.20 builds a metacircular interpreter for a subset of Lisp.

Programming Exercise for Section 12.9

Exercise 12.9.1 In this exercise, you will build a metacircular interpreter for Scheme in Scheme. You will start from the metacircular interpreter in Section 9.7 of *The Scheme Programming Language* (Dybvig 2003), available at <https://www.scheme.com/tspl3/examples.html>. Complete Exercises 9.7.1 and 9.7.2 in that text. This metacircular interpreter is written in Scheme, but it is a simple task to convert it to Racket. Begin by adding the following lines to the top of your program:

```
#lang racket
(define-namespace-anchor anc)
(define ns (namespace-anchor->namespace anc))
(require rnrs/mutable-pairs-6)
```

Once you have the interpreter running, you will self-apply it, repeatedly, until it churns to a near halt, using the following code:

```

1  (define test1 '(((lambda (x . y) (list x y)) 'a 'b 'c 'd)))
2  (define test2 '(((call/cc (lambda (k) k))
3    (lambda (x) x)) "HEY!"))
4  ;; function to compute the length1 of a list;
5  ;; the length of list is returned as a list of empty lists
6  ;; for instance, the length of '(1 2 3) is '(() () ())
7  (define test3 '(((lambda (length1) ((length1 length1) '(1 2 3)))
8    (lambda (length1)
9      (lambda (l)
10        (if (null? l) '()
11          (cons '() ((length1 length1) (cdr l))))))))))
12
13  ;; demonstrates first-class functions
14  (define test4 '(((lambda (kons) (kons 'a '(b c))) cons)))
15
16  ;; metacircular interpreter interpreting simple test cases
17  (apply (eval int ns) test1)
18  (apply (eval int ns) test2)
19  (apply (eval int ns) test3)
20  (apply (eval int ns) test4)
21
22  ;; what follows is: ((I I) expr),
23  ;; where I is the interpreter and
24  ;; expr is the expression being interpreted
25
26  ;; metacircular interpreter interpreting itself
27  (define copy-of-interpreter (apply (eval int ns) (list int)))
28
29  (apply copy-of-interpreter test1)
30  (apply copy-of-interpreter test2)
31  ;(apply copy-of-interpreter test3)
32  (apply copy-of-interpreter test4)
33
34  ;; what follows is: (((I I) I) expr)
35  (define copy-of-copy-of-interpreter
36    (apply copy-of-interpreter (list int)))
37
38  (apply copy-of-copy-of-interpreter test1)
39  (apply copy-of-copy-of-interpreter test2)
40  (apply copy-of-copy-of-interpreter test3)
41  (apply copy-of-copy-of-interpreter test4)
42
43  ;; what follows is: (((((I I) I) I) I) expr)
44  (define copy-of-copy-of-copy-of-interpreter
45    (apply copy-of-copy-of-interpreter (list int)))
46
47  (apply copy-of-copy-of-copy-of-interpreter test1)
48  (apply copy-of-copy-of-copy-of-interpreter test2)
49  (apply copy-of-copy-of-copy-of-interpreter test3)
50  (apply copy-of-copy-of-copy-of-interpreter test4)
51
52  ;; what follows is: ((((((I I) I) I) I) I) I) expr)
53  (define copy-of-copy-of-copy-of-copy-of-interpreter
54    (apply copy-of-copy-of-copy-of-interpreter (list int)))
55
56  (apply copy-of-copy-of-copy-of-copy-of-interpreter test1)
57  (apply copy-of-copy-of-copy-of-copy-of-interpreter test2)
58  (apply copy-of-copy-of-copy-of-copy-of-interpreter test3)
59  (apply copy-of-copy-of-copy-of-copy-of-interpreter test4)

```

```

60
61 ;; what follows is: ((((((I I) I) I) I) I) expr)
62 (define copy-of-copy-of-copy-of-copy-of-copy-of-copy-of-interpreter
63   (apply copy-of-copy-of-copy-of-copy-of-copy-of-interpreter (list int)))
64
65 (apply copy-of-copy-of-copy-of-copy-of-copy-of-copy-of-interpreter test1)
66 (apply copy-of-copy-of-copy-of-copy-of-copy-of-copy-of-interpreter test2)
67 (apply copy-of-copy-of-copy-of-copy-of-copy-of-copy-of-interpreter test3)
68 (apply copy-of-copy-of-copy-of-copy-of-copy-of-copy-of-interpreter test4)

```

12.10 Thematic Takeaways

- Binding and assignment are different concepts.
- The *pass-by-value* and *pass-by-reference* parameter-passing mechanisms are widely supported in programming languages.
- Parameter-passing mechanisms differ in either the direction (e.g., in, out, or in-out) or the content (e.g., value or address) of the information that flows to and from the calling and called functions on the run-time stack.
- *Lazy evaluation* is a fundamentally different parameter-passing mechanism that involves string replacement of parameters with arguments in the body of a function (called β -reduction). Evaluation of those arguments is delayed until the value is required.
- Implementing lazy arguments involves encapsulating an argument expression within the body of a nullary function called a *thunk*.
- There are two implementations of lazy evaluation: *Pass-by-name* is a non-memoized implementation of lazy evaluation; *pass-by-need* is a memoized implementation of lazy evaluation.
- The use of lazy evaluation in a programming language has compelling consequences for programs.
- Lazy evaluation enables infinite data structures that have application in AI applications involving combinatorial search.
- Lazy evaluation enables a *generate-filter* style of programming akin to the filter style of programming common in Linux, where concurrent processes are communicating through a possible infinite stream of data flowing through pipes.
- Lazy evaluation factors control from data in computations, thereby enabling *modular programming*.
- While possible, it is neither practical nor reasonable to support lazy evaluation in a language with provision for side effect.
- The Camille interpreter operationalizes some language concepts and constructs in the Camille programming language from first principles, and others using the direct support for those same constructs in the defining language.
- “The interpreter for a computer language is just another [computer] program” (Friedman, Wand, and Haynes 2001, Foreword, p. vii, Hal Abelson) is one of the most profound, yet simple truths in computing.

12.11 Chapter Summary

Programming languages support a variety of parameter-passing mechanisms. The *pass-by-value* and *pass-by-reference* parameter-passing mechanisms are widely supported in languages. Binding and assignment are different concepts. A *binding* is an association between an identifier and an immutable expressed value; an *assignment* is a mutation of the expressed value stored in a memory cell. References refer to memory cells or variables to which expressed values can be assigned; they refer to variables whose values are mutable. Most parameter-passing mechanisms, except for lazy evaluation, differ in either the direction (e.g., in, out, or in-out) or the content (e.g., value or address), or both, of the information that flows to and from the calling and called functions on the run-time stack.

Lazy evaluation is a fundamentally different parameter-passing mechanism that involves string replacement of parameters with arguments in the body of a function (called β -reduction). Evaluation of those replacement arguments is delayed until the value is required. Thus, unlike other parameter-passing mechanisms, consideration of data flowing to and from the calling and called functions via that run-time stack is relevant to lazy evaluation. The evaluation of an operand is delayed (perhaps indefinitely) by encapsulating it within the body of a function with no arguments, called a *thunk*. A *thunk* acts as a shell for a delayed argument expression. There are two implementations of lazy evaluation: *Pass-by-name* is a non-memoized implementation of lazy evaluation, where the thunk for an argument is evaluated every time the corresponding parameter is referenced in the body of the function; and *pass-by-need* is a memoized implementation of lazy evaluation, where the thunk for an argument is evaluated the first time the corresponding parameter is referenced in the body of the function and the return value is stored so that it can be retrieved for any subsequent references to that parameter. Macros in C, which do not involve the use of a run-time stack, uses pass-by-name semantics of lazy evaluation for parameters. In a language without side effects, evaluating arguments to a function with pass-by-name semantics yields the same result as pass-by-need semantics.

The use of lazy evaluation in a programming language has compelling consequences for programs. Lazy evaluation enables infinite data structures that have application in a variety of artificial intelligence applications involving combinatorial search. It also enables a *generate-filter* style of programming akin to the filter style of programming common in Linux, where concurrent processes communicate through a possibly infinite stream of data flowing through pipes. In addition, lazy evaluation leads to straightforward implementation of complex algorithms (e.g., prime number generators and quicksort). It factors control from data in computations, thereby enabling *modular programming*. While possible, it is neither practical nor reasonable to support lazy evaluation in a language with provision for side effect because lazy evaluation requires the programmer to forfeit control over execution order, which is an integral part of imperative programming.

In this chapter, we introduced variable assignment (i.e., side effect) into Camille. We also implemented the pass-by-reference and lazy evaluation

parameter-passing mechanisms. Finally, we introduced multiple imperative features into Camille, including statement blocks and loops for repetition. The Camille interpreter operationalizes some language concepts and constructs in the Camille programming language from first principles (e.g., local binding, functions, references), and others using the direct support for those same constructs in the defining language, here Python (e.g., `while` loop and compound statements).

12.12 Notes and Further Reading

Fortran was the first programming language to use *pass-by-reference*. *Pass-by-sharing* was first described by Barbara Liskov and others in 1974 in the reference manual for the CLU programming language. The *pass-by-need* parameter-passing mechanism is an example of a more general technique called *memoization*, which is also used in *dynamic programming*. Thunks are used at compile time in the assembly code generated by compilers. Assemblers also manipulate thunks. *Jensen's device* is an application of thunks (i.e., pass-by-name parameters), named for the Danish computer scientist Jørn Jensen, who devised it.

PART IV

OTHER STYLES OF PROGRAMMING

Chapter 13

Control and Exception Handling

Alice: “Would you tell me, please, which way I ought to go from here?”
The Cheshire Cat: “That depends a good deal on where you want to get to.”

— Lewis Carroll, *Alice in Wonderland* (1865)

Continuations are a very powerful tool, and can be used to implement both multiple processes and nondeterministic choice.

— Paul Graham, *On Lisp* (1993)

THIS chapter is about how control is fundamentally imparted to a program and how to affect control in programming. A programmer generally directs flows of control in a program through traditional control structures in programming languages, including sequential statements, conditionals, repetition, and function calls. In this chapter, we explore control and how to affect control in programming through the concepts of *first-class continuations* and *continuation-passing style*. An understanding of how control is fundamentally imparted to a program not only provides a basis from which to build new control structures (e.g., control abstraction), but also provides an improved understanding of traditional control structures.

We begin by introducing first-class continuations and demonstrating their use for nonlocal exits, exception handling, and backtracking. Then we demonstrate how to use first-class continuations to build other control abstractions (e.g., coroutines). Our discussion of first-class continuations for control leads us to issues of improving the space complexity of a program through *tail calls* and *tail-call optimization*. Tail calls leads to an introduction to continuation-passing style and CPS transformation. The CPS transformation supports *iterative control behavior* (i.e., constant memory space) without compromising the one-to-one relationship between recursive specifications/algorithms with their implementation in code.

13.1 Chapter Objectives

- Establish an understanding of how control is fundamentally imparted to a program.
- Establish an understanding of *first-class continuations*.
- Establish an understanding of *tail calls*, including *tail recursion*.
- Describe *continuation-passing style*.
- Explore *control abstraction* through *first-class continuations* and *continuation-passing style*.
- Introduce *coroutines* and *callbacks*.
- Explore language support for functions without a run-time stack.

13.2 First-Class Continuations

13.2.1 The Concept of a Continuation

The concept of a *continuation* is an important, yet under-emphasized and -utilized concept in programming languages. Intuitively, a continuation is a promise to do something. While evaluating an expression in any language, the interpreter of that language must keep track of what to do with the return value of the expression it is currently evaluating. The actions entailed in the “what to do with the return value” step are the pending computations or the continuation of the computation (Dybvig 2009, p. 73). Concretely, a *continuation* is a one-argument function that represents the remainder of a computation from a given point in a program. The argument passed to a continuation is the return value of the prior computation—the one value for which the continuation is waiting to complete the next computation.

Consider the following Scheme expression: `(* 2 (+ 1 4))`. When the interpreter evaluates the subexpression `(+ 1 4)` (i.e., the second argument to the `*` operator), the interpreter must do something with the value 5 that is returned. The something that the interpreter does with the return value is the continuation of the subexpression `(+ 1 4)`. Thus, we can think of a continuation as a pending computation that is awaiting a return value. While the continuation of the expression `(+ 1 4)` is internal to the interpreter while the interpreter is evaluating the expression `(* 2 (+ 1 4))`, we can reify the implicit continuation to make it concrete. The definition of the verb *reify* is “to make (something abstract) more concrete or real” and *reification* refers to the process of reifying. The reified continuation of the subexpression `(+ 1 4)` in the example expression `(* 2 (+ 1 4))` is

```
(lambda (returnvalue)
  (* 2 returnvalue))
```

Thus, a continuation is simply a function of one argument that returns a value.

When working with continuations, it is often helpful to reify the internal, implicit continuation as an external, explicit λ -expression:

The Twentieth Commandment: When thinking about a value created with `(call/cc1 . . .)`, write down the function that is equivalent but does not forget [its surrounding context]. Then, when you use it, remember to forget [its surrounding context]. (Friedman and Felleisen 1996b, p. 160)

Therefore, in the following examples, we reify the internal continuations where possible and appropriate for clarity. For instance, the continuation of the subexpression `(+ 1 4)` in the expression `(* 3 (+ 5 (* 2 (+ 1 4))))` is

```
(lambda (returnvalue)
  (* 3 (+ 5 (* 2 returnvalue))))
```

During evaluation of the expression `(* 3 (+ 5 (* 2 (+ 1 4))))`, eight continuations exist. We present these continuations in an inside-to-out (or right-to-left) order with respect to the expression. The continuations present are the continuations waiting for the value of the following expressions (Dybvig 2009, pp. 73–74):

- (rightmost) +
- (+ 1 4)
- (rightmost) *
- (* 2 (+ 1 4))
- (leftmost) +
- (+ 5 (* 2 (+ 1 4)))
- (leftmost) *
- (* 3 (+ 5 (* 2 (+ 1 4))))

The reified continuation waiting for the value of the rightmost `*` is

```
(lambda (returnvalue)
  (* 3 (+ 5 (returnvalue 2 (+ 1 4)))))
```

The continuation of the subexpression `(+ 1 4)` in the expression

```
(cond
  ((eqv? (* 3 (+ 5 (* 2 (+ 1 4)))) 45) "Continuez")
  (else "Au revoir"))
```

is

```
(lambda (returnvalue)
  (cond
    ((eqv? (* 3 (+ 5 (* 2 returnvalue))) 45) "Continuez")
    (else "Au revoir")))
```

A continuation represents the pending computations at any point in a program—in this case, as a unary function. We can think of a continuation as the pending control context of a program point.

1. The term `call/cc` in this quote is `letcc` in Friedman and Felleisen (1996b).

13.2.2 Capturing First-Class Continuations: `call/cc`

Some language implementations (e.g., interpreters) manipulate continuations internally, but only some (e.g., Scheme, Ruby, ML, Smalltalk) give the programmer first-class access to them. The Scheme function `call-with-current-continuation` (canonically abbreviated `call/cc`) allows a programmer to capture (i.e., reify) the current continuation of any expression in a program. In other words, `call/cc` gives a programmer access to the underlying continuation used by the interpreter. Since a continuation exists at run-time (in the interpreter) and can be expressed in the source language (through the use of `call/cc`), continuations are first-class entities in Scheme. In turn, they can be passed to and returned from functions and assigned to variables.

The `call/cc` function only accepts as an argument a function of one argument f , and captures (i.e., obtains) the current continuation k of the invocation of `call/cc` (i.e., the computations waiting for the return value of `call/cc`) and calls f , passing k to it (or, in other words, applies f to k). The captured continuation is represented as the parameter k of function f . The current continuation k is also a function of one argument. If at any time (during the execution of f) the captured continuation k is invoked with an argument v , control returns from the call to `call/cc` using v as a return value and the pending computations in f are abandoned. The pending computations waiting for `call/cc` to return proceed with v as the return value to the invocation of `call/cc`. The `call/cc` function reifies (i.e., concretizes) the continuation into a function that, when called, transfers control to that captured computation and causes it to resume. If k is not invoked during the execution of f , then the value returned by f becomes the return value of the invocation to `call/cc`.

We begin with simple examples to help the reader understand which continuation is being captured and how it is being used. Later, once we are comfortable with continuations and have an understanding of the interface for capturing continuations in Scheme, we demonstrate more powerful and practical uses of continuations.

Let us discuss some simple examples of capturing continuations with `call/cc`.² Consider the expression `(+ 2 1)`. The continuation of the subexpression `2` is `(lambda (x) (+ x 1))`—expressed in English as “take the result of evaluating `2` and add `1` to it.” Now consider the expression

```
(+ (call/cc (lambda (k) (k 3))) 1)
```

where the `2` in the previous expression has been replaced with `(call/cc (lambda (k) (k 3)))`. This new subexpression captures the continuation of the first argument to the addition operator in the full expression. We already know that the continuation of the first argument is

² While the Scheme function to capture the current continuation is named `call-with-current-continuation`, for purposes of terse exposition we use the commonly used abbreviation `call/cc` for it without including the expression `(define call/cc call-with-current-continuation)` in all of our examples. The function `call/cc` is defined in Racket Scheme.

`(lambda (x) (+ x 1))`. Thus, the invocation of `call/cc` here captures the continuation `(lambda (x) (+ x 1))`. The semantics of `call/cc` are to call its function argument with the current continuation captured. Thus, the expression

```
(call/cc (lambda (k) (k 3)))
```

translates to

```
((lambda (k) (k 3)) (lambda (x) (+ x 1)))
```

The latter expression passes the current continuation, `(lambda (x) (+ x 1))`, to the function `(lambda (k) (k 3))`, which is passed to `call/cc` in the former expression. That expression evaluates to `((lambda (x) (+ x 1)) 3)` or `(+ 3 1)` or 4.

Now let us consider additional examples:

```
> (call/cc
   (lambda (k)
     (* 2 (+ 1 4))))
10
```

Here, the continuation of the invocation of `call/cc` is captured and bound to `k`. Since there are no computations waiting on the return value of `call/cc` in this case, the continuation being captured is the identity function: `(lambda (x) x)`. However, `k` is never used in the body of the function passed to `call/cc`. Thus, the return value of the entire expression is the return value of the body of the function passed to `call/cc`. Typically, we capture the current continuation because we want to use it. Thus, consider the following slightly revised example:

```
> (call/cc
   (lambda (k)
     (* 2 (k 20))))
20
```

Now the captured continuation `k` is being invoked. When `k` is invoked, the continuation of the invocation of `k` [i.e., `(* 2 returnvalue)`] is aborted and the continuation of the invocation to `call/cc` (which is captured in `k` and is still the identity function because no computations are waiting for the return value of the invocation to `call/cc`) is followed with a return value of 20. However, when `k` is invoked, we do not ever return from the expression `(k 20)`. Instead, invoking `k` replaces the continuation of the expression `(k 20)` with the continuation captured in `k`, which is the identity function. Thus, the value passed to `k` becomes the return value of the call to `call/cc`. Since the continuation waiting for the return value of the expression `(k 20)` is ignored and aborted, we can pass any value of any type to `k` because, in this case, the continuation stored in `k` is the identity function, which is polymorphic:

```
> (call/cc
   (lambda (k)
     (* 2 (k "break out"))))
"break out"
```


Now we modify the original expression so that the continuation being captured by `call/cc` is no longer the identity function:

```
> (/ 100 (call/cc
          (lambda (k)
            (* 2 (k 20)))))
5
```

The continuation being captured by `call/cc` and bound to `k` is

```
(lambda (returnvalue)
  (/ 100 returnvalue))
```

Again, when `k` is invoked, we never return from the expression `(k 20)`. Instead, invoking `k` replaces the continuation of the expression `(k 20)` with the continuation captured in `k`, which is `(lambda (returnvalue) (/ 100 returnvalue))`. Thus, the value passed to `k` becomes the return value of the call to `call/cc`. In this case, `call/cc` returns 20. Since a computation that divides 100 by the return value of the invocation of `call/cc` is pending, the return value of the entire expression is 5. Now we must pass an integer to `k`, even though the continuation waiting for the return value of the expression `(k 20)` is ignored, because it becomes the operand to the pending division operator:

```
> (/ 100 (call/cc
          (lambda (k)
            (* 2 (k "break out")))))
/: contract violation
expected: number?
given: "break out"
argument position: 2nd
other arguments...:
```

Instead of continuing with the value used as the divisor, we can continue with the value used as the dividend:

```
> (/ (call/cc
      (lambda (k)
        (* 2 (k 20)))) 5)
4
```

Thus, a first-class continuation, like a `goto` statement, supports an arbitrary transfer of control, but in a more systematic and controlled fashion than a `goto` statement does. Moreover, unlike a `goto` statement, when control is transferred with a first-class continuation, the environment—including the run-time stack at the time `call/cc` was originally invoked—is restored. A continuation represents a captured, not suspended, series of computations awaiting a value.

In summary, we have discussed two ways of working with first-class continuations. One form involves not using (i.e., invoking) the captured continuation in the body of the function passed to `call/cc`:

```

(call/cc (lambda (k)
  ;; body of lambda without a call to k
))

```

When k is *not invoked* in the body of the function f passed to `call/cc`, the return value of the call to `call/cc` is the return value of f . In general, a call to `(call/cc (lambda (k)  $E$ ))`, where k is not called in E , is the same as a call to `(call/cc (lambda (k) (k  $E$ )))` (Haynes, Friedman, and Wand 1986, p. 145). In the other form demonstrated, the captured continuation is *invoked* in the body of the function passed to `call/cc`:

```

(call/cc (lambda (k)
  ...
  ;; body of lambda with a call to k
  (k v)
  ... rest of body is ignored (i.e., is not evaluated)
))

```

If the continuation is invoked inside f , then control returns from the call to `call/cc` using the value passed to the continuation as a return value. Control does not return to the function f and all pending computations are left unfinished—this is called a *nonlocal exit* and is explored in Section 13.3.1. The examples of continuations in this section demonstrate that, once captured, a programmer can use (i.e., call) the captured continuation to replace the current continuation elsewhere in a program, when desired, to circumvent the normal flow of control and thereby affect, manipulate, and direct control flow. Figure 13.1 illustrates the general process of capturing the current continuation k through `call/cc` in Scheme and later replacing the current continuation k' with k .

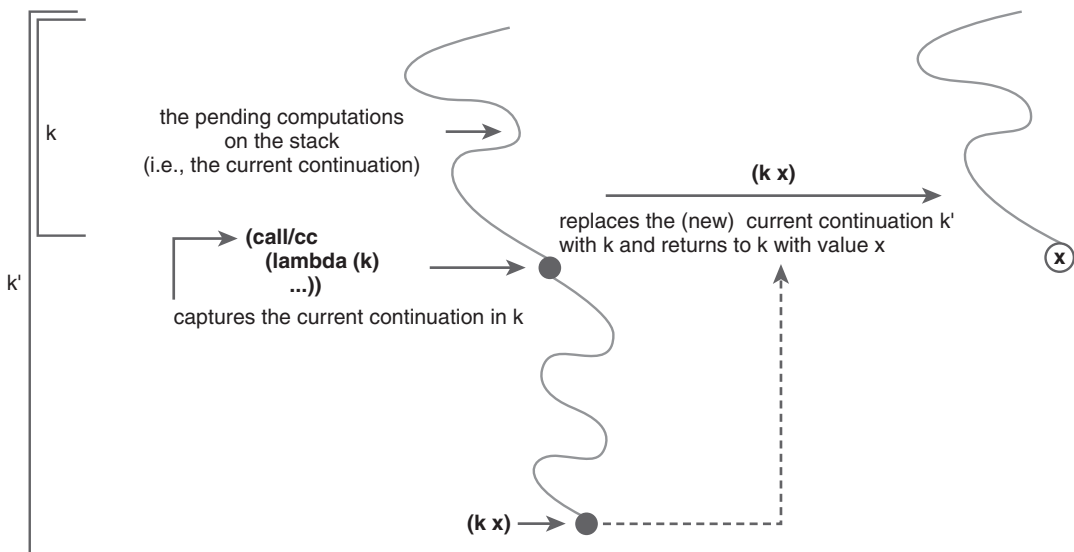


Figure 13.1 The general `call/cc` continuation capture and invocation process.

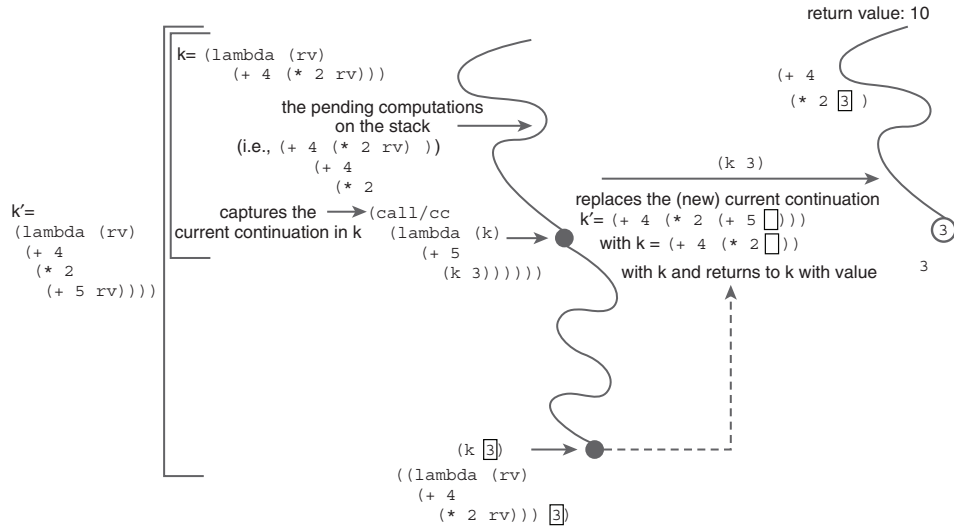


Figure 13.2 Example of a `call/cc` continuation capture and invocation process.

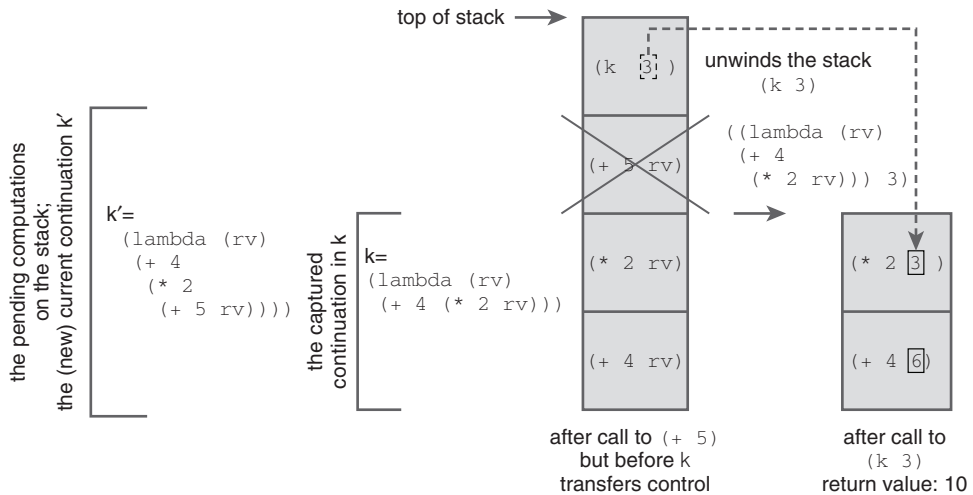


Figure 13.3 The run-time stack during the continuation replacement process depicted in Figure 13.2.

Figure 13.2 provides an example of the process, and Figure 13.3 depicts the run-time stack during the continuation replacement process from that example.

Conceptual Exercises for Section 13.2

Exercise 13.2.1 Consider the expression `(* 2 3)`. Reify the continuation of each of the following subexpressions:

(a) *

(b) 2

(c) 3

Exercise 13.2.2 Reify the continuation of the expression `(+ x 2)` in the expression `(* 3 (+ x 2))`.

Exercise 13.2.3 Predict the output of the following expression:

```
(sqrt (* (call/cc
          (lambda (k)
            (cons 2 (k 20)))) 5))
```

Exercise 13.2.4 Consider the following Scheme expression:

```
> (+ 1 (call/cc (lambda(k) (k (k 1)))))
2
```

Explain, by appealing to transfer of control and the run-time stack, why the return value of this expression is 2 and not 3. Also, reify the continuation captured by the call to `call/cc` in this expression. Does a continuation ever return (like a function)?

Programming Exercises for Section 13.2

Exercise 13.2.5 In the following example, when `k` is invoked, we do not return from the expression `(k 20)`. Instead, invoking `k` replaces the continuation of the expression `(k 20)` with the continuation captured in `k`, which is the identity function:

```
> (call/cc
   (lambda (k)
     (* 2 (k 20))))
20
```

Modify this expression to also capture the continuation of the expression `(k 20)` with `call/cc`. Name this continuation `k2` and use it to complete the entire computation with the default continuation (now captured in `k2`).

Exercise 13.2.6 The interface for capturing continuations used in *The Seasoned Schemer* (Friedman and Felleisen 1996b) is called `letcc`. Although `letcc` has a slightly different syntax than `call/cc`, both have approximately the same semantics (i.e., they capture the current continuation). The `letcc` function only accepts an identifier and an expression, in that order, and it captures the continuation of the expression and binds it to the identifier. For instance, the following two expressions are analogs of each other:

```

> (/ 100 (call/cc
        (lambda (k)
          (* 2 (k 20)))))
5

> (/ 100 (letcc k (* 2 (k 20))))
5

```

- (a) Give a general rewrite rule that can be used to convert an expression using `letcc` to an equivalent expression using `call/cc`. In other words, give an expression using only `call/cc` that can be used as a replacement for every occurrence of the expression `(letcc k e)`.
- (b) Assume `letcc` is a primitive in Scheme. Define `call/cc` using `letcc`.

Exercise 13.2.7 Investigate and experiment with the interface for first-class continuations in ML (see the structure `SMLofNJ.Cont`):

```

- open SMLofNJ.Cont;
[autoloading]
[library $SMLNJ-BASIS/basis.cm is stable]
[library $SMLNJ-BASIS/(basis.cm):basis-common.cm is stable]
[autoloading done]
opening SMLofNJ.Cont
  type 'a cont = 'a ?.cont
  val callcc : ('a cont -> 'a) -> 'a
  val throw : 'a cont -> 'a -> 'b
  val isolate : ('a -> unit) -> 'a cont
  type 'a control_cont = 'a ?.InlineT.control_cont
  val capture : ('a control_cont -> 'a) -> 'a
  val escape : 'a control_cont -> 'a -> 'b

```

Replicate any three of the examples in Scheme involving `call/cc` given in this section in ML.

13.3 Global Transfer of Control with Continuations

Armed with an elementary understanding of the concept of a continuation and how to capture a continuation in Scheme, we present some practical examples of first-class continuations. While continuations are used for a variety of purposes in these examples, all of these examples use `call/cc` for global transfer of control.

13.3.1 Nonlocal Exits

A common application of a first-class continuation is to program abnormal flows of control, such as a *nonlocal exit* from recursion without having to return through multiple layers of recursion. Consider the following recursive definition of a Scheme function `product` that accepts a list of numbers and returns the product of the numbers:

```

> (define product
  (lambda (lon)
    (cond
      ((null? lon) 1)
      (else (* (car lon) (product (cdr lon)))))))

> (product '(1 2 3 4 5))
120

```

This function exhibits *recursive control behavior*, meaning that when the function is called its execution causes the stack to grow until the base case of the recursion is reached. At that point, the computation is performed as recursive calls return and pop off the stack. The following series of expressions depicts this process:

```

> (product '(1 2 3 4 5))
> (* 1 (product '(2 3 4 5)))
> (* 1 (* 2 (product '(3 4 5))))
> (* 1 (* 2 (* 3 (product '(4 5)))))
> (* 1 (* 2 (* 3 (* 4 (product '(5)))))
> (* 1 (* 2 (* 3 (* 4 (* 5 (product '())))))) ; base case
> (* 1 (* 2 (* 3 (* 4 (* 5 1)))))
> (* 1 (* 2 (* 3 (* 4 5))))
> (* 1 (* 2 (* 3 20)))
> (* 1 (* 2 60))
> (* 1 120)
120

```

Rotating this series of expansions 90 degrees to the left yields a parabola-shaped curve. The x -axis of that parabola can be interpreted as time, while the y -axis represents memory. As time proceeds, the function requires an ever-increasing amount of memory. Once it hits the maximum point at the base case, it starts to occupy less and less memory until it finally terminates. This is the manner in which most recursive functions operate. This process remains unchanged irrespective of the input list passed to `product`. For instance, consider another invocation of the function with a list of numbers that includes a zero:

```

> (product '(1 2 3 0 4 5))
> (* 1 (product '(2 3 0 4 5)))
> (* 1 (* 2 (product '(3 0 4 5))))
> (* 1 (* 2 (* 3 (product '(0 4 5)))))
> (* 1 (* 2 (* 3 (* 0 (product '(4 5)))))
> (* 1 (* 2 (* 3 (* 0 (* 4 (product '(5)))))
> (* 1 (* 2 (* 3 (* 0 (* 4 (* 5 (product '()))))))
> (* 1 (* 2 (* 3 (* 0 (* 4 (* 5 1)))))
> (* 1 (* 2 (* 3 (* 0 (* 4 5)))))
> (* 1 (* 2 (* 3 (* 0 20))))
> (* 1 (* 2 (* 3 0)))
> (* 1 (* 2 0))
> (* 1 0)
0

```

As soon as a zero is encountered in the list, the final return value of the function is known to be zero. However, the recursive control behavior continues to build up the stack of pending computations until the base case is reached, which signals the commencement of the computations to be performed. This function is inefficient

in space whether the input contains a zero or not. It is inefficient in time only when the input list contains a zero—unnecessary multiplications are performed.

The presence of a zero in the input list can be considered an *exception* or exceptional case. Exceptions are unusual situations that happen at run-time, such as erroneous input. One application of first-class continuations is for exception handling.

We want to break out of the recursion as soon as we encounter a zero in the input list of numbers. Consider the following new definition of `product` (Dybvig 2003):

```

1  (define product
2    (lambda (lon)
3      (call/cc
4        ;; break stores the current continuation
5        (lambda (break)
6          (letrec ((P (lambda (l)
7                        (cond
8                          ;; base case
9                          ((null? l) 1)
10                         ;; exceptional case; abnormal flow of control
11                         ((zero? (car l)) (break 0))
12                         ;; inductive case; normal flow of control
13                         (else (* (car l) (P (cdr l)))))))
14            (P lon))))))

```

If `product` is invoked as `(product '(1 2 3 0 4 5))`, the continuation bound to break on line 5 is `(lambda (returnvalue) returnvalue)`, which is the identity function, because there are no pending computations waiting for `product` to complete. If `product` is invoked as `(+ 1 (product '(1 2 3 0 4 5)))`, the continuation bound to break on line 5 is `(lambda (returnvalue) (+ 1 returnvalue))`. When passed a list of numbers including a zero, `product` aborts the current continuation (i.e., the pending computations built up on the stack) and uses the continuation of the first call to `product` to break out to the main read-eval-print loop (line 11). This action is called a *nonlocal exit* because the local exit to this function is through the termination of the recursion as the stack naturally unwinds. The function builds up the *capability* of calling a series of multiplication operators, but does so only after the function has determined that the input list does not contain a zero. The function goes through the list in a left-to-right order, building up these multiplication computations. Once the function has determined that the input list does not contain a zero, the multiplication operations are conducted in a right-to-left fashion as the function backs out of the recursion:

```

> (product '(1 2 3 0 4 5))                ; works efficiently now
> (* 1 (product '(2 3 0 4 5)))
> (* 1 (* 2 (product '(3 0 4 5))))
> (* 1 (* 2 (* 3 (product '(0 4 5)))))
0

> (product '(1 2 3 4 5))                  ; still works
120

```

The case where the list does not contain a zero proceeds as usual, using the current continuation of pending multiplications on the stack rather than the captured continuation of the initial call to `product`. Like the examples in Section 13.2, this `product` function demonstrates that once a continuation is captured through `call/cc`, a programmer can use (i.e., call) the captured continuation to replace the current continuation elsewhere in a program, when desired, to circumvent the normal flow of control and, therefore, alter control flow.

Notice that in this example, the definition of the nested function `P` within the `letrec` expression (lines 6–13) is necessary because we want to capture the continuation of the *first* call to `product`, rather than recapturing a continuation every time `product` is called recursively. For instance, the following definition of `product` does not achieve the desired effect because the continuation `break` is rebound on each recursive call and, therefore, is not the exceptional/abnormal continuation, but rather the normal continuation of the computation:

```

1  (define product
2    (lambda (lon)
3      (call/cc
4        ;; break is rebound to the current continuation
5        ;; on every recursive call to product
6        (lambda (break)
7          (cond
8            ;; base case
9            ((null? lon) 1)
10           ;; exceptional case; abnormal flow of control
11           ((zero? (car lon)) (break 5))
12           ;; inductive case;      normal flow of control
13           (else (* (car lon) (product (cdr lon))))))))

```

We continue with 5 (line 11) to demonstrate that the continuation stored in `break` is actually the normal continuation:

```

> (product '(1 2 3 4 5))
120
> (product '(1 2 3 0 4 5))
30

```

To break out of this type `letrec`-free style of function definition, the function could be defined to accept an abnormal continuation, but the caller would be responsible for capturing and passing it to the called function. For instance:

```

> (define product
  (lambda (break lon)
    (cond
      ;; base case
      ((null? lon) 1)
      ;; exceptional case; abnormal flow of control
      ((zero? (car lon)) (break 0))
      ;; inductive case;      normal flow of control
      (else (* (car lon) (product break (cdr lon))))))

> (call/cc (lambda (break) (product break '(1 2 3 4 5))))
120
> (call/cc (lambda (break) (product break '(1 2 3 0 4 5))))
30

```



```

0
> (+ 100 (call/cc (lambda (break) (product break '(1 2 3 0 4 5)))))
100

```

Factoring out the constant parameter `break` (using *Functional Programming Design Guideline 6* from Table 5.7 in Chapter 5) again renders a definition of `product` using `letrec` expression:

```

(define product
  (lambda (break lon)
    (letrec ((P (lambda (l)
                  (cond
                    ;; base case
                    ((null? l) 1)
                    ;; exceptional case; abnormal flow of control
                    ((zero? (car l)) (break 0))
                    ;; inductive case; normal flow of control
                    (else (* (car l) (P (cdr l)))))))
      (P lon))))

```

While first-class continuations are used in these examples for programming efficient nonlocal exits, continuations have a broader context of applications, as we demonstrate in this chapter.

13.3.2 Breakpoints

Consider the following recursive definition of a Scheme factorial function that accepts an integer `n` and returns the factorial of `n`:

```

(define factorial
  (lambda (n)
    (cond
      ((zero? n) 1)
      (else (* n (factorial (- n 1)))))))

```

Now consider the same definition of `factorial` using `call/cc` to capture the continuation of the base case (i.e., where `n` is 0) (Dybvig 2009, pp. 75–76):

```

> (define redo "ignore")

> (define factorial
  (lambda (n)
    (cond
      ((zero? n) (call/cc (lambda (k) (set! redo k) 1)))
      (else (* n (factorial (- n 1)))))))

> ;; a side effect of the evaluation of the following expression
> ;; is that redo is bound to the continuation captured in factorial
> (factorial 5)
120

```

Unlike the continuation captured in the `product` example in Section 13.3.1, where the continuation captured is of the initial call to the recursive function `product` (i.e., the identity function), here the continuation captured includes all of the pending multiplications built up on the stack when the base of the recursion (i.e.,

`n = 0`) is reached. For instance, when `n = 5`, the continuation captured and bound to `k` is

```
(lambda (returnvalue)
  (* 5 (* 4 (* 3 (* 2 (* 1 returnvalue))))))
```

Moreover, unlike in the `product` example, here the captured continuation is not invoked from the `lambda` expression passed to `call/cc`. Instead, the continuation is stored in the variable `redo` using the assignment operator `set!`. The consequence of this side effect is that the captured continuation can be invoked from the main read-eval-print loop after `factorial` terminates, when and as many times as desired. In other words, the continuation captured by `call/cc` is invoked after the function passed to `call/cc` returns:

```
> (redo 1)
120

> (redo 0)
0

> (redo 2)
240

> (redo 3)
360

> (redo 4)
480

> (redo 5)
600

> (redo -1)
-120

> (redo -2)
-240
```

The natural base case of recursion for `factorial` is 1. However, by invoking the continuation captured through the use of `call/cc`, we can dynamically change the base case of the recursion at run-time. Moreover, this `factorial` example vividly demonstrates the—perhaps mystifying—*unlimited extent* of a first-class continuation.

The thought of transferring control to pending computations that no longer exist on the run-time stack hearkens back the examples of first-class closures returned from functions (in Chapter 6) that “remembered” their lexical environment even though that environment no longer existed because the activation record for the function that created and returned the closure had been popped off the stack (Section 6.10).

The continuation captured by `call/cc` is, more generally, a closure—a pair of (code, environment) pointers—where the code is the actual continuation and the environment is the environment in which the code is to be later evaluated. However, when invoked, the continuation (in the closure) captured with `call/cc`,

unlike a regular closure (i.e., one whose code component is not a continuation), does not return a value, but rather transfers control elsewhere. Similarly, when we invoke `redo`, we are jumping back to activation records (i.e., stack frames) that no longer exist on the stack because the `factorial` function has long since terminated, and been popped off the stack, by the time `redo` is called. The key connection back to our discussion of first-class closures in Chapter 6 is that the first-class continuations captured through `call/cc` are only possible because closures in Scheme are allocated from the heap and, therefore, have unlimited extent. If closures in Scheme were allocated from the run-time stack, an example such as `factorial`, which uses a first-class continuation to jump back to seemingly “phantom” stack frames, would not be possible.

The `factorial` example illustrates the use of first-class continuations for *breakpoints* and can be used as a basis for a breakpoint facility in a debugger. In particular, the continuation of the breakpoint can be saved so that the computation may be restarted from the breakpoint—more than once, if desired, and, with different values.

Unlike in the prior examples, here we store the captured continuation in a variable through assignment, using the `set!` operator. This demonstrates the first-class status of continuations in Scheme. Once a continuation is captured through `call/cc`, a programmer can store the continuation in a variable (or data structure) for later use. The programmer can then use the captured continuation to replace the current continuation elsewhere in a program, when and as many times as desired (now that it is recorded persistently in a variable), to circumvent the normal flow of control and, therefore, manipulate control flow. There is no limit on the number of times a continuation can be called, which implies that heap-allocated activation records must exist.

13.3.3 First-Class Continuations in Ruby

In an expression-oriented language, *the continuation of an expression* is the calling expression, which is generally found to the *left* or *above* the expression whose continuation is being captured (as in our invocations of `call/cc`). In a language whose control flows along a sequential execution of statements, *the continuation of a statement* is the set of statements *following* the statement whose continuation is being captured. Consider the following `product` function in Ruby—a language whose statements are executed sequentially:

```

1  require "continuation"
2
3  def product(lon)
4
5      # base case
6      if lon == [] then 1
7
8      # exceptional case
9      elsif lon[0] == 0 then $break.call "Encountered a zero. Break out."
10
11     # inductive case
12     else

```

```

13     #return lon[0] * product lon[1..-1]
14     print "before recursive call\n"
15     res = product lon[1..-1]
16     print "after recursive call\n"
17     return lon[0]*res
18 end
19 end
20
21 # normal case; continuation break not used in product
22 result = callcc {|k| $break = k
23               product [1,2,3,4] }
24 print result
25 print "\n"
26
27 # exceptional case; continuation break used in product for nonlocal exit
28 result = callcc {|k| $break = k
29               product [1,2,0,4] }
30 print result
31 print "\n"

```

Ruby does not support nested methods. Thus, instead of capturing the continuation of a local, nested function *P* (as done in the second definition of *product* in Section 13.3.1), here the caller saves the captured continuation *k* with *callcc*³ of each called function (lines 22–23 and 28–29) in a global variable *\$break* (lines 22 and 28) so that the called function has access to it. The continuation captured in the local variable *k* on line 22 represents the set of program statements on lines 24–31. Similarly, the continuation captured in the local variable *k* on line 28 represents the set of program statements on lines 30–31. In each case, the captured continuation in the local variable *k* is saved persistently in the global variable *\$break* so that it can be accessed in the definition of *product* by using *\$break* and called by using *\$break.call* with the string argument "Encountered a zero. Break out." (line 9). The output of this program is

```

1  $ ruby product.rb
2  before recursive call
3  before recursive call
4  before recursive call
5  before recursive call
6  after recursive call
7  after recursive call
8  after recursive call
9  after recursive call
10 24
11 before recursive call
12 before recursive call
13 Encountered a zero. Break out.

```

Lines 2–10 of the output demonstrate that the product of a list of non-zero numbers is computed while popping out of the (four) layers of recursive calls. Lines 11–13 of the output demonstrate that no multiplications are performed when a zero is encountered in the input list of numbers (i.e., the nonlocal exit abandons the recursive calls on the stack).

3. While the examples in Ruby in this chapter run in the current version of Ruby, *callcc* is currently deprecated in Ruby.

Conceptual Exercises for Section 13.3

Exercise 13.3.1 Does the following definition of `product` perform any unnecessary multiplications? If so, explain how and why (with reasons). If not, explain why not (with reasons).

```
(define product
  (lambda (lon)
    (call/cc
      (lambda (break)
        (cond
          ((null? lon) 1)
          ((zero? (car lon)) (break 0))
          (else (* (car lon) (product (cdr lon))))))))))
```

Exercise 13.3.2 Can the `factorial` function using `call/cc` given in this section be redefined to remove the side effect (i.e., without use `set!`), yet retain the ability to dynamically alter the base of the recursion? If so, define it. If not, explain why not. In other words, why is side effect necessary in that example (if it is)?

Exercise 13.3.3 Explain why the `letrec` expression is necessary in the definition of `product` using `call/cc` in this section. In other words, why can't `product` be defined just as effectively as follows? Explain.

```
(define product
  (lambda (lon)
    (call/cc
      (lambda (break)
        (cond
          ((null? lon) 1)
          ((zero? (car lon)) (break 0))
          (else (* (car lon) (product (cdr lon))))))))))
```

Exercise 13.3.4 Consider the following attempt to remove the side effect (i.e., the use of `set!`) from the `factorial` function using `call/cc` given in this section:

```
> (define factorial
    (lambda (n)
      (cond
        ((zero? n) (call/cc (lambda (k) (cons 1 k))))
        (else
         (let ((answer (factorial (- n 1))))
           (cons (* n (car answer)) (cdr answer)))))))

> (factorial 5)
' (120 . #<continuation>)

> ((cdr (factorial 5)) (cons 2 "ignore"))
application: not a procedure;
expected a procedure that can be applied to arguments
given: "ignore"
arguments...:
```

The approach taken is to have `factorial` return a pair whose `car` is an integer representing the factorial of its argument and whose `cdr` is the redo continuation, rather than just an integer representing the factorial. As can be seen from the preceding transcript, this approach does not work.

- (a) Notice that `(cdr (factorial 5))` returns the continuation of the base case (i.e., the redo continuation). Explain why that rather than passing a single number to it, as done in the example in this section, now a pair must be passed instead—for example, the list `(cons 2 "ignore")` in this case.
- (b) Evaluating `((cdr (factorial 5)) (cons 2 "ignore"))` results in an error. Explain why. You may want to try using the tracing (step-through) ability provided through the Racket debugging facility to help construct a clearer picture of the internal process.
- (c) Explain why the invocation to `factorial` and subsequent use of the continuation as `((cdr (factorial 5)) (cons 5 (cdr (factorial 5))))` never terminates.

Exercise 13.3.5 Consider the following definition of `product`:

```
(define product
  (lambda (lon)
    (call/cc
      (lambda (break)
        (cond
          ((null? lon) 1)
          ((zero? (car lon)) (break 0))
          (else (* (car lon) (product (cdr lon))))))))
```

- (a) Indicate how many (i.e., the number of) continuations are captured when this function is called as `(product '(9 12 7 3))`.
- (b) Indicate how many (i.e., the number of) continuations are captured when this function is called as `(product '(42 11 0 2 -1))`.

Programming Exercises for Section 13.3

Table 13.1 presents a mapping from the greatest common divisor exercises here to some of the essential aspects of first-class continuations and `call/cc`.

Exercise 13.3.6 Define a recursive Scheme function `member1` that accepts only an atom `a` and a list of atoms `lat` and returns the integer position of `a` in `lat` (using zero-based indexing) if `a` is a member of `lat` and `#f` otherwise. Your definition of `member1` must use `call/cc` to avoid returning back through all the recursive calls when the element `a` is not found in the list, but it must not use the captured continuation when the element `a` is found in the list.

Programming Exercise	Start from	Input		Nonlocal Exit for		No Unnecessary Operations Computed
		LoN	S-Expression	1 in List	Intermediate gcd = 1	
13.3.13	N/A	✓	×	✓	×	✓
13.3.14	13.3.13	✓	×	✓	✓	✓
13.3.15	N/A	×	✓	✓	×	✓
13.3.16	13.3.15	×	✓	✓	✓	✓

Table 13.1 Mapping from the Greatest Common Divisor Exercises in This Section to the Essential Aspects of First-Class Continuations and `call/cc`

Examples:

```
> (member1 'a '(a b c))
0
> (member1 'a '(b c a))
2
> (member1 'a '(d b c))
#f
> (member1 'c '(d a b c))
3
```

Exercise 13.3.7 Complete Programming Exercise 13.3.6 in Ruby using `callcc`.

Exercise 13.3.8 Define a Scheme function `map-reciprocal`, which uses `map`, that accepts only a list of numbers `lon` and returns a list containing the reciprocal of each number in `lon`. Use `call/cc` to foster an immediate nonlocal exit of the function as soon as a 0 is encountered in `lon` without returning through each of the recursive calls on the stack.

```
> (map-reciprocal '(1 2 3 4 5))
(1 1/2 1/3 1/4 1/5)

> (map-reciprocal '(1 2 0 4 5))
"Divide by zero!"
```

Exercise 13.3.9 Complete Programming Exercise 13.3.8 in Ruby using `callcc`.

Exercise 13.3.10 Rewrite the Ruby program in Section 13.3.3 so that the caller passes the captured continuation `k` of the called function `product` on lines 23 and 29 to the called function itself (as done in the third definition of `product` in Section 13.3.1).

Exercise 13.3.11 Define a Scheme function `product` that accepts a *variable* number of arguments and returns the product of them. Define `product` using `call/cc` such that no multiplications are performed if any of the arguments are zero.

Exercise 13.3.12 (Friedman, Wand, and Haynes 2001, Exercise 1.17.1, p. 27) Consider the following BNF specification of a binary search tree.

```

<binsearchtree> ::= ()
<binsearchtree> ::= (<integer> <binsearchtree> <binsearchtree>)

```

Define a Scheme function `path` that accepts only an integer `n` and a list `bst` representing a binary search tree, in that order, and returns a list of `lefts` and `rights` indicating how to locate the vertex containing `n`. If the integer is not found in the binary search tree, use `call/cc` to avoid returning back through all the recursive calls and return the atom `'notfound`.

Examples:

```

> (path 42 '(52 (24 (14 (8 (2 () ()) ()) (17 () ()))
                (32 (26 () ()) (42 () (51 () ())))
                (78 (61 () ()) (101 () ())))))
'(left right right)
> (path 17 '(14 (7 () (12 () ()))
              (26 (20 (17 () ()) ())
              (31 () ())))))
'(right left left)
> (path 32 '(14 (7 () (12 () ()))
              (26 (20 (17 () ())
                    (31 () ())))))
'notfound
> (path 17 '(17 () ()))
'()
> (path 17 '(18 () ()))
'notfound
> (path 2 '(31 (15 () ()) (42 () ())))
'notfound
> (path 31 '(31 (15 () ()) (42 () ())))
'()
> (path 17 '(52 (24 (14 (8 (2 () ()) ()) (17 () ()))
                (32 (26 () ()) (42 () (51 () ())))
                (78 (61 () ()) (101 () ())))))
'(left left right)

```

Exercise 13.3.13 Define a function `gcd-lon` in Scheme using `call/cc` that accepts only a non-empty list of *positive, non-zero* integers and returns the greatest common divisor of those integers. If a 1 is encountered in the list, through the use of `call/cc`, return the string "1: encountered a 1 in the list" immediately without ever executing `gcd` (which is defined in Racket Scheme) and without returning through each of the recursive calls on the stack.

Examples:

```

> (gcd-lon '(20 48 32 1))
"1: encountered a 1 in the list"
> (gcd-lon '(4 32 12 8 16))
4
> (gcd-lon '(4 32 1 12 8 16))
"1: encountered a 1 in the list"
> (gcd-lon '(12 18 24))
6
> (gcd-lon '(4 8 12))
4

```



```

> (gcd-lon '(12 4 8))
4
> (gcd-lon '(18 12 22 20 30))
2
> (gcd-lon '(4 8 11 11))
1
> (gcd-lon '(4 8 11))
1
> (gcd-lon '(128 256 512 56))
8
> (gcd-lon '(12 24 32))
4

```

Exercise 13.3.14 Modify the solution to Programming Exercise 13.3.13 so that if a 1 is ever computed as the result of an intermediate call to `gcd`, through the use of `call/cc`, the string "1: computed an intermediary gcd = 1" is returned immediately without returning through each of the recursive calls on the stack and before performing any additional arithmetic computations.

Examples:

```

> (gcd-lon '(20 48 32 1))
"1: encountered a 1 in the list"
> (gcd-lon '(4 32 12 8 16))
4
> (gcd-lon '(4 32 1 12 8 16))
"1: encountered a 1 in the list"
> (gcd-lon '(12 18 24))
6
> (gcd-lon '(4 8 12))
4
> (gcd-lon '(12 4 8))
4
> (gcd-lon '(18 12 22 20 30))
2
> (gcd-lon '(4 8 11 11))
"1: computed an intermediary gcd = 1"
> (gcd-lon '(4 8 11))
"1: computed an intermediary gcd = 1"
> (gcd-lon '(128 256 512 56))
8
> (gcd-lon '(12 24 32))
4

```

Exercise 13.3.15 Define a function `gcd*` in Scheme using `call/cc` that accepts only a non-empty *S-expression* of *positive, non-zero* integers, which contains no empty lists, and returns the greatest common divisor of those integers. If a 1 is encountered in the list, through the use of `call/cc`, return the string "1: encountered a 1 in the S-expression" immediately without ever executing `gcd` and without returning through each of the recursive calls on the stack.

Examples:

```

> (gcd* '((36 12 48) (((24 36) 6 54 240))))
6

```

```

> (gcd* '((((20)))) 48 (32) 1))
"1: encountered a 1 in the S-expression"
> (gcd* '((4 (32 12) 8) ((16))))
4
> (gcd* '((4 32 1) (12 (8) 16)))
"1: encountered a 1 in the S-expression"
> (gcd* '((((12 18 24))))
6
> (gcd* '((4) ((8) 12)))
4
> (gcd* '(12 4 8))
4
> (gcd* '((18) (12) (22) (20) (30)))
2
> (gcd* '(4 8 (((11 11)))))
1
> (gcd* '(((4 8)) (11)))
1
> (gcd* '((((128 (256 512 56)))))
8
> (gcd* '((((12) ((24))) (32))))
4

```

Exercise 13.3.16 Modify the solution to Programming Exercise 13.3.15 so that if a 1 is ever computed as the result of an intermediate call to `gcd`, through the use of `call/cc`, the string "1: computed an intermediary gcd = 1" is returned immediately without returning through each of the recursive calls on the stack and before performing any additional arithmetic computations.

Examples:

```

> (gcd* '((36 12 48) (((24 36) 6 54 240)))))
6
> (gcd* '((((20)))) 48 (32) 1))
"1: encountered a 1 in the S-expression"
> (gcd* '((4 (32 12) 8) ((16))))
4
> (gcd* '((4 32 1) (12 (8) 16)))
"1: encountered a 1 in the S-expression"
> (gcd* '((((12 18 24))))
6
> (gcd* '((4) ((8) 12)))
4
> (gcd* '(12 4 8))
4
> (gcd* '((18) (12) (22) (20) (30)))
2
> (gcd* '(4 8 (((11 11)))))
"1: computed an intermediary gcd = 1"
> (gcd* '(((4 8)) (11)))
1: computed an intermediary gcd = 1
> (gcd* '((((128 (256 512 56)))))
8
> (gcd* '((((12) ((24))) (32))))
4

```

Exercise 13.3.17 Define a function `intersect*` in Scheme using `call/cc` that accepts only a list of lists as an argument and returns the set intersection of these

lists. Your function must not perform any unnecessary computations. Specifically, if the input list contains an empty list, immediately return `()` without returning through each of the recursive calls on the stack. Further, if the input list does not contain an empty list, but contains two lists whose set intersection is empty, immediately return `()`. You may assume that each list in the input list represents a set (i.e., contains no duplicate elements). Your solution must follow *Design Guidelines 4 and 6* from Table 5.7 in Chapter 5.

13.4 Other Mechanisms for Global Transfer of Control

In this section we discuss the conceptual differences between first-class continuations and imperative mechanisms for nonlocal transfer of control. This comparison provides more insight into the power of first-class continuations.

13.4.1 The `goto` Statement

The `goto` statement in most languages supporting primarily imperative programming is reserved for nonlocal transfer of control:

```

1  $ cat goto.c
2  #include <stdio.h>
3
4  int main() {
5      printf ("%d\n", repetez);
6  again:
7      printf ("%d\n", encore);
8      goto again;
9  }
10 $
11 $ gcc goto.c
12 $ ./a.out
13 repetez
14 encore
15 encore
16 encore
17 ...

```

This simple example illustrates the use of a label `again:` (line 6) and a `goto` (line 8) to create a repeated transfer of control resulting in an infinite loop.

Programmers are generally advised to avoid `gotos` because they violate the spirit of *structured programming*. This style of (typically imperative) programming is aimed at improving the readability and maintainability, and reducing the potential for errors, of a computer program through the use of functions and block control structures (e.g., `if`, `while`, and `for`) with only one entry and exit point as opposed to tests and jumps (e.g., `goto`) found in assembly programs. Use of `goto` statements can result in “spaghetti code” that is difficult to follow and, thus, challenging to debug and maintain. Programming languages that originally

lacked structured programming constructs but now support them include Fortran, COBOL, and BASIC.

Edsger W. Dijkstra wrote a letter titled “Go To Statement Considered Harmful” in 1968 arguing against the use of the `goto` statement. His letter (Dijkstra 1968) and the emergence of imperative languages with suitably expressive control structures, including ALGOL, supported a shift toward structured programming. Later, Donald E. Knuth (1974b), in his paper “Structured Programming with `goto` Statements,” identified cases where a jump leads to clearer and more efficient code. Notwithstanding, `goto` statements cannot be used to jump across functions on the stack:

```
$ cat goto_fun.c
#include<stdio.h>

int f() {
    printf ("avant\n");
again:
    printf ("apres\n");
}

int main() {
    int i=0;

    f();
    while (i++ < 10) {
        printf ("%d\n", i);
        goto again;
    }
}
$
$ gcc goto_fun.c
goto_fun.c:15:12: error: use of undeclared label 'again'
    goto again;
        ^
```

The `goto` statement can only be used to transfer control within one lexical closure. Therefore, we cannot replicate the previous examples using `call/cc` with `gotos`. In other words, a `goto` statement is not as powerful as a first-class continuation.

13.4.2 Capturing and Restoring Control Context in C: `setjmp` and `longjmp`

The closest facility to `call/cc` in the C programming language is the `setjmp` and `longjmp`⁴ suite of library functions, which can be used in concert for nonlocal transfer of control:

```
1 $ cat simple_setjmp.c
2 #include<stdio.h>
3 #include<setjmp.h>
4
5 int main() {
6     jmp_buf env;
7     int x = setjmp(env);
```

4. The `setjmp` and `longjmp` functions tend to be highly system dependent.

```

8     printf ("x = %d\n", x);
9     longjmp(env, 5);
10  }
11  $
12  $ gcc simple_setjmp.c
13  $ ./a.out
14  x = 0
15  x = 5
16  x = 5
17  x = 5
18  ...

```

The `setjmp` function saves its calling environment in its only argument (named `env` here) and returns 0 the first time it is called. Notice the first line of output on line 14 is `x = 0`. The `setjmp` function serves the same purpose as the label `again::`; that is, it marks a destination for a subsequent transfer of control. However, unlike a label and more like capturing a continuation using `call/cc`, this function saves the current environment at the time it is called (for later restoration by `longjmp`). In this example, the environment is empty, meaning that it does not contain any name–value pairs. The `longjmp` function acts like a `goto` in that it transfers control. However, unlike `goto`, the `longjmp` function also restores the original environment (captured when `setjmp` was called) to the point where control is transferred. The `longjmp` function never returns. Instead, when `longjmp` is called, the call to `setjmp` sharing the buffer passed in each invocation returns (line 7), but this time with the value passed as a second argument to `longjmp` (in this case 5; line 9). Notice the lines of output from line 15 onward contain `x = 5`. Thus, the `setjmp` and `longjmp` functions communicate through a shared `struct` buffer of type `jmp_buf` that represents the captured environment.

When used in the manner just described in the same function (i.e., `main`) and with an empty environment, `setjmp` and `longjump` act like a label and a `goto`, respectively, and effect a simple nonlocal transfer of control. The captured environment is unnecessary in this example; that is, it simply serves to convey the semantics of `setjmp/longjump`.

The `setjmp` function is similar to `call/cc`; the `longjmp` function is similar to `(k v)` (i.e., it invokes the continuation captured in `k` with the value `v`); and `jmp_buf env` is similar to the captured continuation `k` (Table 13.2). Recall that a closure is a pair consisting of an *expression* [e.g., `(lambda (y) (+ x y))`] and an *environment* [e.g., `(x 8)`]. In other words, a closure is program code that “remembers” its lexical environment. A continuation is also a closure: The “what to do with the return value” is the expression component of the closure, and the environment to be restored after the transfer of control is the environment

Semantics	Scheme	C
captures branch point and environment	<code>call/cc</code>	<code>setjmp</code>
restores branch point and environment	<code>(k v)</code>	<code>longjmp</code>
environment	<code>k</code>	<code>jmp_buf env</code>

Table 13.2 Facilities for Global Transfer of Control in Scheme Vis-à-Vis C

component. The `call/cc` function returns a closure that, when called, never returns.

There is, however, a fundamental difference between `setjmp/longjmp` and `call/cc`. This difference is a consequence of the location where Scheme and C store closures in the run-time system, or alternatively the extent of closures in Scheme and C. Consider the following C program using `setjmp/longjmp`, which is an attempt to replicate the `factorial` example using `call/cc` in Scheme in Section 13.3.2 to help illustrate this difference:

```

1  $ cat factorial.c
2  #include<stdio.h>
3  #include<setjmp.h>
4
5  jmp_buf env;
6
7  int factorial(int n) {
8      int x;
9
10     if (n == 0) {
11         x = setjmp(env);
12         printf("Inside the factorial function.\n");
13         if (x == 0)
14             return 1;    /* normal base of recursion */
15         else
16             return x;    /* new base of recursion passed from longjmp */
17     } else
18         return n*factorial(n-1);
19 }
20
21 int main() {
22     printf ("%d\n", factorial(5));
23     longjmp(env, 3); /* (k 3) */
24 }
25 $
26 $ gcc factorial.c
27 $ ./a.out
28 Inside the factorial function.
29 120
30 Inside the factorial function.
31 Segmentation fault: 11

```

In this example, unlike in the simple example at the beginning of Section 13.4.2, the environment captured through `setjmp` comes into focus. Here, the `factorial` function invokes `setjmp` in the base case (line 11) where its parameter `n` is 0 (line 10). It then returns normally back through all of the recursive calls, progressively computing the factorial (i.e., performing the multiplications) as the activation records for `factorial` pop off the stack. By the time control returns to `main` at line 22 where the factorial is printed, those stack frames for `factorial` are gone. The invocation of `longjmp` on line 23 seeks to transfer control back to the invocation of `factorial` corresponding to the base case (when the parameter `n` is 0) and to return from the call to `setjmp` on line 11 with the value 3, effectively changing the base of the recursion from 1 to 3 and ultimately returning 360. However, when `longjmp` is called at line 23, `main` is the only function on the stack. The invocation of `longjmp` on line 23 is tantamount to jumping to a phantom stack frame, meaning a stack frame that is no longer there (Figure 13.4).

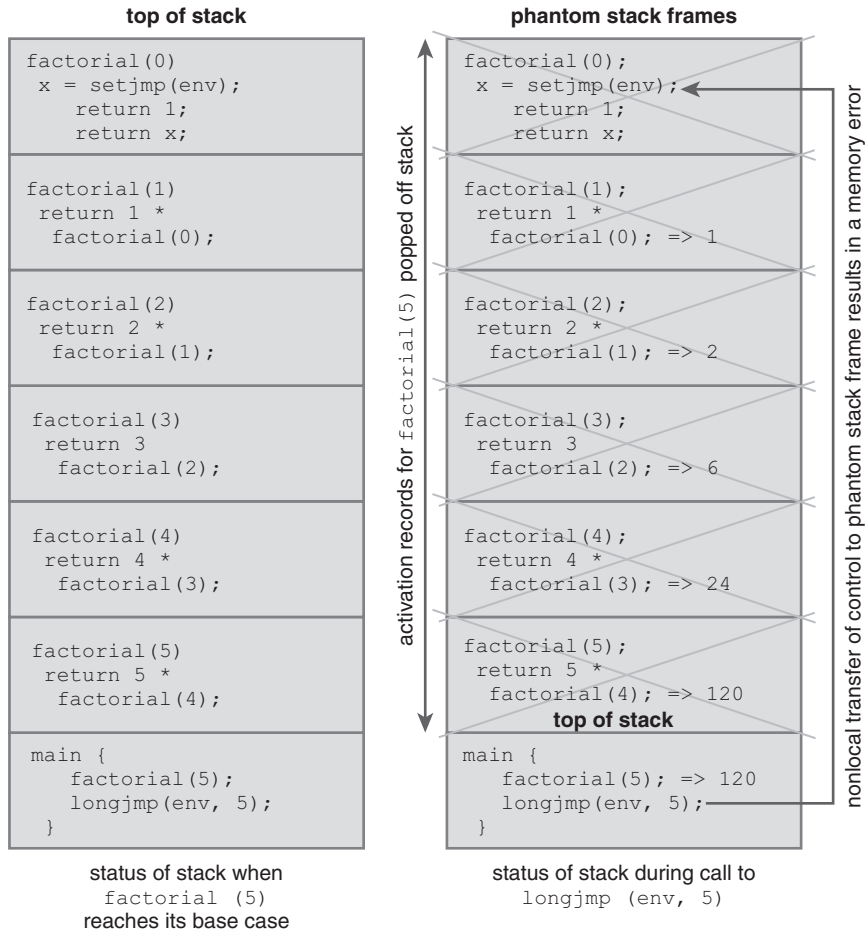


Figure 13.4 The run-time stacks in the factorial example in C.

Thus, the nonlocal transfer of control through the use of `set jmp/long jmp` is limited to frames that are still active on the stack. Using these functions, we can only jump to code that is active and, therefore, has a limited extent. For instance, we can make a nonlocal exit from several functions in a single jump, as we did in the second definition of `product` using `call/cc` in Section 13.3.1:

```

1 $ cat jumpstack.c
2 #include<stdio.h>
3 #include<set jmp.h>
4
5 jmp_buf env;
6
7 int d(int x) {
8     /* exceptional case; need to break out,
9      but do not want to return back through
10     all of the calls on the stack */
11     fprintf(stderr, "Jumping back to main without ");

```

```

12     fprintf(stderr, "returning through c, b, and a ");
13     fprintf(stderr, "on the stack.\n");
14     longjmp(env, -1);
15 }
16
17 int c(int x) {
18     return 3 + d(x*3);
19 }
20
21 int b(int x) {
22     return 2 * c(x+2);
23 }
24
25 int a(int x) {
26     return 1 + b(x+1);
27 }
28
29 int main() {
30     if (setjmp(env) != 0)
31         fprintf(stderr, "Error case.\n");
32     else
33         a(1);
34 }
35 $ gcc jumpstack.c
36 $ ./a.out
37 Jumping back to main without returning
38 through c, b, and a on the stack.
39 Error case.

```

Here, we can jump directly back to `main` because the activation record for `main` is *still active on the run-time stack* (i.e., it still exists). By doing so, we bypass the functions `a`, `b`, and `c`. The stack frames for `d`, `c`, `b`, and `a` are removed from the stack and disposed of properly as if each function had exited normally, in that order, when the `longjmp` happens. In other words, `setjmp/longjmp` can be used to jump down the stack, but not back up it.

The `setjmp` function is the analog of a statement `:label`, whereas the `longjmp` function is the analog of the `goto` statement. The main difference between a `:label/goto` and the `setjmp/longjmp` pair is that `longjmp` cleans up the stack in addition to transferring control; `goto` just transfers control.

Let us compare the `factorial` example in Section 13.4.2 with this example. In the `factorial` example, we attempt to jump from `main` directly back to a stack frame for the last invocation of `factorial` (i.e., for the base case where `n` is 0), which no longer exists. Here, we are jumping directly back to the stack frame for `main`, from the stack frame for `d`, which still exists on the stack because it is waiting for `d`, `c`, `b`, and `a` to return normally and complete the continuation of the computation. At the time `d` is called [as `d(12)`], the stack is `main → a → b → c → d`, where the stack grows left-to-right. Thus, the top of the stack is on the right. The continuation of pending computations is

$$\begin{array}{rcl}
 1 + & \text{return value of } b(1+1) & = \\
 1 + & (2 * \text{return value of } c(2+2)) & = \\
 1 + & (2 * (3 + \text{return value of } d(4*3))) & = \\
 1 + & (2 * (3 + \text{return value of } d(12))) &
 \end{array}$$

This scenario is illustrated through the stacks presented in Figure 13.5.

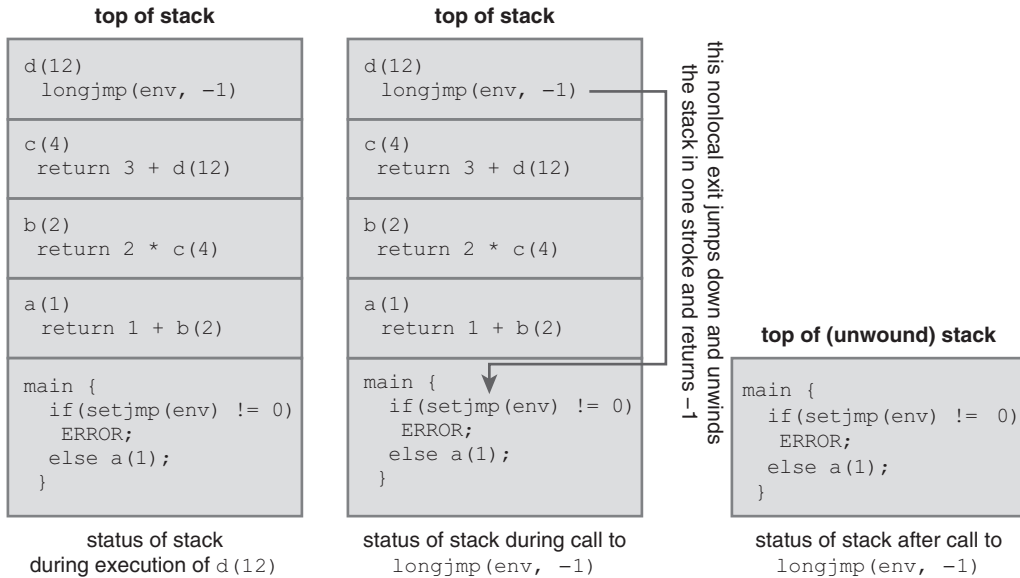


Figure 13.5 The run-time stacks in the `jumpstack.c` example.

The key difference between `setjmp/longjmp` and `call/cc` is that closures (or stack frames) have limited extent in C, while they have unlimited extent in Scheme because closures are allocated from the stack in C and from the heap in Scheme. When a first-class continuation is captured through `call/cc`, that continuation remembers the entire execution state of the program at the time it was created (i.e., at the time `call/cc` was invoked) and can resume the program later even if the stack frames have since seemingly disappeared (i.e., been deallocated or garbage collected).

The `setjmp/longjmp` functions operate by manipulating the stack pointer, rather than by actually saving the stack. Once a function in C has returned, the memory occupied by its stack frame, which contained its parameters and local variables, is reclaimed by the system. In contrast, a continuation captured with the `call/cc` function in Scheme has access to the entire stack, so it can restore the stack at any time later when the continuation is invoked. This discussion is reminiscent of the examples of first-class closures returned from functions that “remembered” their lexical context even though it no longer existed because the activation record for the function that created and returned the closure had been popped off the run-time stack (Section 6.10).

In the Scheme example of `factorial` using `call/cc` in Section 13.3.2, the invocations to `redo` always return without error with the correct answer. Once the continuation of the base case of `factorial` is captured through `call/cc` and assigned to `redo` (with the `set!` operator), it can be called (i.e., followed, activated, or continued) at any time, including after all of the calls to `factorial` have returned and, therefore, after all of the activation records for `factorial`

	Facility	Semantics
least flexible/ ↑ general ↓ most flexible/ general	<code>:label</code> and <code>goto</code>	only nonlocal transfer of control <i>within</i> a single function; does not clean up the stack
	<code>set jmp/long jmp</code> in C	nonlocal transfer of control both <i>within</i> and <i>between</i> functions <i>currently on the stack</i> (i.e., active extent) + restored context/environment; + unwinds the stack, but does not restore it
	<code>call/cc</code> and <code>(k v)</code> in Scheme	nonlocal transfer of control both <i>within</i> and <i>between</i> <i>any functions</i> + restored context/environment + unwinds and restores the stack

Table 13.3 Summary of Methods for Nonlocally Transferring Program Control

have popped off the stack. Whenever that continuation is called, we are transferred directly into the middle of the base case call to `factorial`, which is executing normally, with the illusion of all of its parent activation records still on the stack waiting for the call to the base case to terminate. Moreover, that continuation can be reused as many times as desired without error—the same is not possible in C. In essence, the `set jmp` and `long jmp` functions represent a middle ground between the unwieldiness of `gotos` and the generality of `call/cc` for nonlocal transfer of control (Table 13.3).

The important point to observe here is that the combination of `(call/cc (lambda (k) ...))` and `(k v)` does not just capture the current continuation and transfer control, respectively. Instead, `(call/cc (lambda (k) ...))` captures the current continuation, *including the environment and the status of the stack*, and `(k v)` transfers control *while restoring the environment and the stack*. The `set jmp` function captures the environment, but does not capture the status of the stack. Consequently, the `long jmp` function, unlike `(k v)`, requires any stack frame to which it is to jump to be active. Thus, the `set jmp` and `long jmp` functions can be implemented in Scheme using first-class continuations to simulate their semantics (Programming Exercise 13.4.8), illustrating the generality, power, and flexibility of first-class continuations.

Nonetheless, the `set jmp` and `long jmp` functions are helpful for exception handling within this limitation. The following is a common programming idiom for using these functions for exception handling:

```

if (set jmp(env) == 0)
  /* protected code block;
    call long jmp when an exception is encountered */
else

```

```

    /* exception handler;
       return point from a longjmp */
}

```

A return value of 0 for `set jmp` indicates a normal return, while a non-zero return value indicates a return from `longjmp`. If `longjmp` is called anywhere within the protected block, or in any function called within that block, then `set jmp` will return (again), causing control to be transferred to the exception handler. Again, a call to `longjmp` after the protected code block completes (and pops off the stack) is undefined and generally results in a memory error.

Conceptual Exercises for Section 13.4

Exercise 13.4.1 Explain why Scheme does not suffer from the problem demonstrated in the factorial C program in Section 13.4.2.

Exercise 13.4.2 Assume closures and, by extension, local variables in Scheme do not have unlimited extent. Describe an implementation approach for `call/cc` that would support transfer of control to stack frames which seemingly no longer exist, as demonstrated in the `factorial` example using `call/cc` in Section 13.3.2.

Programming Exercises for Section 13.4

Exercise 13.4.3 Use `set jmp/longjmp` to complete Programming Exercise 13.3.6 in C. Represent a list in C as an array of characters. The `member1` function in C must be recursive. It can also accept the size of the list and the current index as arguments:

```
int member1(int a, char lst[], int length, int start_index)
```

Exercise 13.4.4 Use `set jmp/longjmp` to complete Programming Exercise 13.3.8 in C.

Exercise 13.4.5 Write a C program with three functions: `main`, `A`, and `B`. The `main` function calls `A`, which then calls `B`. Low-level computation that might result in an error is performed in functions `A` and `B`. All error handling is done in `main`. Use `set jmp` and `longjmp` for error handling. The `main` function must be able to discern which of the other two functions (i.e., `A` or `B`) generated the error. *Hint:* Use a `switch` statement.

Exercise 13.4.6 The Common Lisp functions `catch` and `throw` have nearly the same semantics as `set jmp` and `longjmp` in C, respectively. Moreover, `catch` and `throw` expressions in Common Lisp can be easily translated into equivalent Scheme expressions involving `(call/cc (lambda (k) ...))` and `(k v)`, respectively (Haynes and Friedman 1987, p. footnote †, p. 11):

Common Lisp	Scheme
(catch <i>id expr</i>)	(call/cc (lambda (<i>id</i>) <i>expr</i>))
(throw <i>id result</i>)	(<i>id result</i>)

Replicate the `jumpstack.c` C program in Section 13.4.2 in Common Lisp using `catch` and `throw`. Use an implementation of Common Lisp available from <https://clisp.org>.

Exercise 13.4.7 Complete Programming Exercise 13.4.5 in Common Lisp using `catch` and `throw`.

Exercise 13.4.8 Define the functions `setjmp` and `longjmp` in Scheme with the same functional signatures as they have in C. Use a Scheme vector to store the `jmp_buf`.

Exercise 13.4.9 Solve Programming Exercise 13.4.5 in Scheme using the Scheme functions `setjmp` and `longjmp` defined in Programming Exercise 13.4.8. Do not invoke the `call/cc` function outside of the `setjmp` function.

Exercise 13.4.10 Replicate the `jumpstack.c` C program in Section 13.4.2 in Scheme using the Scheme functions `setjmp` and `longjmp` defined in Programming Exercise 13.4.8. Do not invoke the `call/cc` function outside of the `setjmp` function.

Exercise 13.4.11 When the C function `longjmp` is called, control is transferred directly to the call to the function `setjmp` that is closest to the call to `longjmp` that uses the same `jmp_buf`. Write a C program as an experiment to determine if “closest” means “closest lexically” or “closest on the run-time stack.” In other words, can we determine the point to which control is transferred by simply examining the source code of the program (i.e., statically) or must we run the program (i.e., dynamically)? You may need to compile with the `-fnested-functions` option to `gcc`.

Exercise 13.4.12 Complete Programming Exercise 13.4.11 in Scheme using the Scheme functions `setjmp` and `longjmp` defined in Programming Exercise 13.4.8. Do not invoke the `call/cc` function outside of the `setjmp` function.

13.5 Levels of Exception Handling in Programming Languages: A Summary

Thus far, we have discussed first-class continuations primarily in the context of handling exceptions in programming. Exception handling is a convenient place to start with continuations because it involves transfer of control. In this section, we summarize the mechanisms in programming languages for handling exceptions.

13.5.1 Function Calls

Passing error codes as return values through the run-time stack is a primitive, low-level way of handling exceptions. Consider a program where `main` calls `A`, which calls `B`, which generates an exception. The function `B` can return an error code to `A`, which in turn can return that error back to the `main` program, which can report the error to the user. A problem with this approach is that functions usually return result values, not control information like error codes. While it is possible to return both a return value and an error code through a variety of mechanisms (e.g., reference parameters), doing so integrates too tightly the code for normal processing with that for exception handling. That tight coupling increases the complexity of a program and turns error handling into a global property of program design rather than a cleanly separated property concentrated in an independent program unit, as are other constituent program components. Moreover, error codes, which are typically generated by low-level routines, must be passed through each intermediate function all the way down the stack to the `main` program. Lastly, once activation records have been popped off the stack, control cannot be transferred back to the function that generated the exception. The approach to exception handling that entails passing error codes up the function-call chain is encoded/sketched in C in the following programming idiom:

```
int B() {
    /* perform some low-level computation */
    /* return valid result from B or an error code from B */
}

int A() {
    int result;
    /* perform some computation */
    if (error)
        return error code from A;
    else {
        result = B();

        if (result of B is an error code) {
            /* process the error here in function A */
            /* or */
            /* pass the error up the call chain by returning it */
        }

        /* here result could be a valid result or
           an error code from B */
        return result;
    }
}

int main() {
    switch (A()) {

        /* dispatch to exception handler for function A */
        case 1:
            handlerForExceptionIn_A();

        /* dispatch to exception handler for function B */
        case 2:
```

```

        handlerForExceptionIn_B();
    }
}

```

13.5.2 Lexically Scoped Exceptions: `break` and `continue`

The `break` and `continue` statements in Python, Java, and C can be used to raise a lexically scoped exception. *Lexically scoped exceptions* can be raised as long as the lexical parent of the block that raises the exception is available to catch it. Thus, lexically scoped exceptions are a structured type of `goto`, in that they can be used only for *local exits*. The following is a simple example of a lexically scoped exception in Python:

```

1  >>> i = 1
2  >>> while i <= 10:
3  ...     print(i)
4  ...     if i == 3:
5  ...         break
6  ...     i += 1
7  ...
8  1
9  2
10 3

```

The `while` loop on lines 2–6 executes only three times because the `break` on line 5 terminates the execution of loop when `i` equals 3.

13.5.3 Stack Unwinding/Crawling

The use of dedicated functions such as `setjmp` and `longjmp` to install and transfer control to nonlocal exit handlers obviates the need to pass control information, including error codes, through each intermediate function all the way down the stack to the main program. This approach to exception handling is often referred to as *stack unwinding*, *stack unraveling*, or *stack crawling*. However, once activation records have been popped off the run-time stack, control cannot be transferred back to the function that caused the exception. This approach also makes handling exceptions less of a global property of program design than function calls are, because return values are used solely for normal, nonexceptional results, while the statement that transfers control (e.g., `longjmp`) communicates the error code. This approach to exception handling is encoded/sketched in C in the following programming idiom:

```

int B() {
    /* perform some computation */
    if (error)
        longjmp(env, 2);
    else
        /* return normal result */
}

```

```

int A() {
    /* perform some computation */
    if (error)
        longjmp(env, 1);
    else
        return B();
}

int main() {
    switch (setjmp(env)) {

        /* protected code block */
        case 0:
            A();
            break;

        /* dispatch to exception handler for function A */
        case 1:
            handlerForExceptionIn_A();

        /* dispatch to exception handler for function B */
        case 2:
            handlerForExceptionIn_B();
    }
}

```

The `setjmp/longjmp` suite of functions can be considered a primitive exception-handling system, especially when used in conjunction with signal handlers.

13.5.4 Dynamically Scoped Exceptions: Exception-Handling Systems

Some programming languages include systems for handling exceptions (e.g., Java, Python, C++). Exceptions can be matched to a handler by name or object. In languages in which the exception raised is an object (e.g., Python or Java), contextual information about the exception (e.g., where it was raised) can be passed to the handler. This approach also makes handling exceptions much less a global property of the program design than stack unwinding/crawling is, because exception handling can be localized into individual classes and objects and control is transferred automatically. Languages that transfer control to the handler only after the activation records between the function that raised the exception and the function that defines the handler (i.e., the entire control context) are popped off the stack use *terminating semantics*. Most programming languages, including Python and Java, use terminating semantics. The terminating-semantics approach to exception-handling systems is encoded/sketched in Python in the following programming idiom:

```

import traceback

class SampleException(Exception):
    def __init__(self, msg):
        self.msg = msg

    def getMessage(self):
        return self.msg

```

```

def A():
    print("Inside A.")
    B()

def B():
    print("Inside B.")
    try:
        C()
    except SampleException as e:
        print(e.getMessage())
        print(str(e))
        traceback.print_exc(limit=None)
        # jump back to D here so the computation can complete

def C():
    print("Inside C.")
    D()

def D():
    print("Inside D.")
    raise SampleException("D raised an exception.")
    # do more computation here

print("Main program.")
A()

```

The output of this program is

```

Main program.
Inside A.
Inside B.
Inside C.
Inside D.
D raised an exception.
D raised an exception.
Traceback (most recent call last):
  File "exception_example.py", line 17, in B
    C()
  File "exception_example.py", line 26, in C
    D()
  File "exception_example.py", line 30, in D
    raise SampleException("D raised an exception.")
SampleException: D raised an exception.

```

Calling semantics are sometimes referred to as *resumable semantics* or *resumable exceptions*.

13.5.5 First-Class Continuations

Function calls, stack unwinding/crawling, and exception-handling systems that use terminating semantics make no provision for returning to the point where the original transfer of control took place—that is, where the exception is raised or calling semantics. During exception handling, once the activation records between the function that raised the exception and the handler are popped off the stack, they are not pushed back on. These approaches allow control to be transferred down the stack, but not back up it. Thus, these mechanisms are intended for

Control Mechanisms for Exception Handling	
high level	<i>first-class continuations</i> , with heap-allocated activation records
	<i>dynamically scoped exceptions: exception-handling systems</i> (e.g., <code>try: except</code> blocks in Python)
	<i>stack unwinding/crawling</i> ; nonlocal exit handlers (i.e., without passing error codes as return values through the stack; e.g., <code>set jmp/long jmp</code>)
	<i>lexically scoped exceptions</i> (e.g., <code>break</code> and <code>continue</code>) for local exits
	<i>function calls</i> (i.e., pass error codes as return values through the stack)
low level	

Table 13.4 Mechanisms for Handling Exceptions in Programming Languages

nonlocal exits and, unlike first-class continuations, are limited in their use for implementing other types of control structures (e.g., the breakpoint illustrated in the `factorial` example in Section 13.3.2). In contrast, *first-class continuations* with heap-allocated activation records can be used as the basis for a general, high-level mechanism for exception handling in programming languages. For instance, first-class continuations can be used to build an exception-handling system using calling semantics. First-class continuations with heap-allocated activation records have been referred to as *reinvocable continuations* or *reentrant continuations*, whereas *escape continuations* can only be used to escape the current control context to a surrounding one (e.g., exception-handling systems with terminating semantics or `set jmp/long jmp` in C). The first-class continuation approach to simple exception handling is encoded/sketched in Scheme in the following programming idiom:

```
(call/cc
  (lambda (break)
    ;; perform some computations
    ;; if an exception happens, invoke (break ...)
    ;; perform more computations
  ))
```

Table 13.4 summarizes the mechanisms for handling exceptions in the programming languages discussed in this section.

Programming Exercises for Section 13.5

Exercise 13.5.1 Modify the solution to Programming Exercise 13.4.10 so that the main program jumps back to where the original exception occurred in function `d` after it handles the exception in the main program. Recall that this action is impossible in programming languages that use stack-based control mechanisms. To make the program cleaner, the functions `a`, `b`, `c`, and `d` need not take any arguments. The functions `a`, `b`, and `c` can simply call the functions `b`, `c`, and `d`, respectively. Do not invoke the `call/cc` function outside of the `set jmp` function.

Exercise 13.5.2 Rewrite the Python program in Section 13.5.4, which demonstrates the programming idiom for the terminating semantics of exception-handling systems, in Java.

13.6 Control Abstraction

Granting a programmer access to the underlying continuation in the interpreter in the presence of heap-allocated activation records empowers the programmer to implement a wide range of control structures. In this section, we demonstrate control abstraction through first-class continuations—an essential ingredient for control abstraction in programming languages. *Control abstraction* refers to the facilities and support that a language provides the programmer to concretely model and manipulate the control of a program (e.g., `goto` in C). Facilities for affecting the control of a program can be categorized by the scope of program control they affect: *local* (e.g., sequence, conditionals, and repetition) or *global* (e.g., `goto`, `break`, function calls, exception-handling systems, and continuations). However, aside from continuations in Scheme, none of these entities for affecting program control have first-class status. While facilities for data and procedural abstraction abound abundantly in programming languages, less attention is paid to the equally important concept of control abstraction. Figure 13.6 depicts these three types of abstraction in programming languages and underscores the underemphasis of control abstraction. Table 13.5, which is modified from Pérez-Quinones (1996, p. 109), captures the slightly different view of the dichotomy between data and control abstraction in programming languages.

Support for control abstraction in programming languages is necessary if a programmer requires control structures beyond the traditional mechanisms provided by the language, including sequential statements, conditionals, repetition,

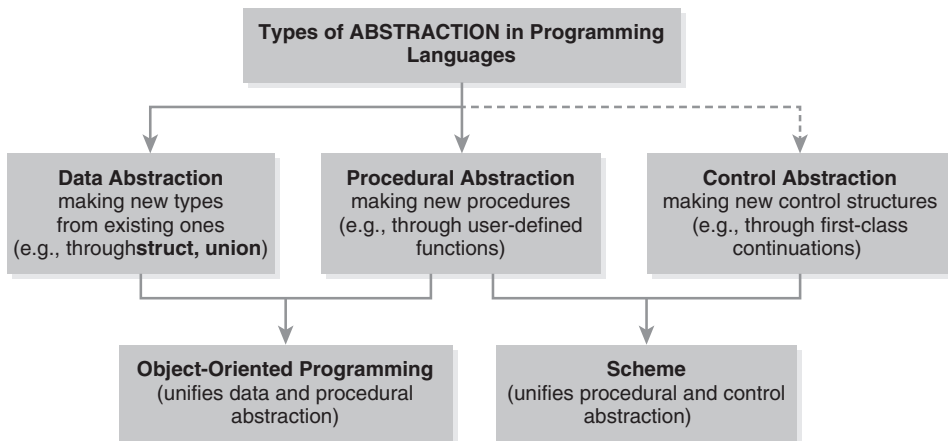
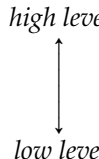


Figure 13.6 Data and procedural abstraction with control abstraction as an afterthought.



Data Abstraction	Control Abstraction
object-oriented classes; structures with data access	first-class continuations
offset-based access direct memory access	function call stack unwinding/crawling goto

Table 13.5 Levels of Data and Control Abstraction in Programming Languages (Pérez-Quiñones 1996, p. 109)

and function calls. Moreover, the ability to leverage our understanding of how control is fundamentally imparted to a program as a basis for building new control structures facilitates an improved understanding of traditional control structures.

A criticism of facilities for capturing a first-class continuation (e.g., `call/cc`) is that they are only necessary for dealing with the problems endemic to a functional style of programming. In other words, they might mitigate the effects of recursion for repetition in programming (e.g., breaking out of layers of recursion), but are not really needed in languages whose control flows along a sequential execution of statements and that support repetition through iterative control structures (e.g., a while loop). This perspective views `call/cc` primarily as a mechanism to break out of recursion in a clean way (i.e., jumping several activation records down the stack), which is unnecessary in languages whose primary mode of repetition is iteration.

This criticism also highlights a fundamental difference between functional and imperative programming. In imperative programming, we generally program iteratively and, therefore, have no need for a first-class continuation. However, the perspective just mentioned presumes that a first-class continuation is only intended for nonlocal exits for exception handling or, more generally, for jumping down the run-time stack. That is a limited view of a first-class continuation. While using a first-class continuation for nonlocal exits is common practice, and we used nonlocal exits to initially demonstrate the use of `call/cc`, exceptional handling is only one instance of a much more general use of first-class continuations—namely, for *control abstraction*. While some languages provide a variety of (typically low-level) mechanisms to transfer control (e.g., `gotos`, function calls, stack unwinding or crawling), other languages recognize the benefits of and provide general facilities for control abstraction. In addition to nonlocal exits for exception handling, first-class continuations can be used to create new control abstractions. For instance, we demonstrate their use for implementing coroutines in the following subsection.

13.6.1 Coroutines

A *coroutine* is a function whose execution can be suspended and resumed in cooperation with other coroutines. Coroutines are an instance of *cooperative multitasking* or *nonpreemptive multitasking*—the coroutine itself, and not some external factor, decides when to suspend its execution. In this sense, coroutines

cooperatively collaborate to solve a problem within a single process. The following is an implementation of coroutines from Dybvig (2009, pp. 76–77) with stylistic modifications:

```

1  ;;; a simple implementation of coroutines in Scheme
2
3  (define ready-queue '())
4
5  (define spawn-coroutine
6    (lambda (coroutine)
7      (set! ready-queue (append ready-queue (cons coroutine '()))))
8
9  (define start-next-ready-coroutine
10   (lambda ()
11     ;; take first coroutine off ready queue
12     (let ((coroutine (car ready-queue)))
13       ;; remove first coroutine from ready queue
14       (set! ready-queue (cdr ready-queue))
15       ;; start thunk
16       (coroutine)))
17
18  (define pause-coroutine
19   (lambda ()
20     (call/cc
21      (lambda (resume)
22        (spawn-coroutine (lambda () (resume "ignored")))
23        (start-next-ready-coroutine))))
24
25  (define new-coroutine
26   (lambda (c)
27     (lambda ()
28       (letrec ((f (lambda ()
29                     (pause-coroutine)
30                     (display c)
31                     (f))))
32         (f))))
33
34  ;; create eleven coroutines and start the first
35  (spawn-coroutine (new-coroutine "c"))
36  (spawn-coroutine (new-coroutine "o"))
37  (spawn-coroutine (new-coroutine "o"))
38  (spawn-coroutine (new-coroutine "p"))
39  (spawn-coroutine (new-coroutine "e"))
40  (spawn-coroutine (new-coroutine "r"))
41  (spawn-coroutine (new-coroutine "a"))
42  (spawn-coroutine (new-coroutine "t"))
43  (spawn-coroutine (new-coroutine "e"))
44  (spawn-coroutine (new-coroutine "."))
45  (spawn-coroutine (new-coroutine "\n"))
46
47  (start-next-ready-coroutine)

```

Each coroutine is represented as an anonymous, argumentless function—or thunk—whose body consists of the operations to be interleaved (lines 27–32). The ready queue (i.e., queue of coroutines ready to run) is represented as list (line 3). The `spawn-coroutine` function (lines 5–7) accepts a thunk representing the coroutine (returned from the `new-coroutine` function) as an argument and appends it to the end of the ready queue (line 7). The `start-next-ready-coroutine` function removes a coroutine from the front of the ready queue (lines 12–14) and starts it by invoking the

function representing the coroutine (line 16). The `pause-coroutine` function captures the current continuation using `call/cc`; packages it as a thunk, `(lambda () (resume "ignored"))`, or, in other words, as a coroutine; and places it on the ready queue (line 22). Since the body of each coroutine is a series of statements evaluated for side effect (lines 29–31) rather than a group of expressions waiting to return, the value passed to the continuation of any of those statements is ignored and never used; that is, no other expression is waiting on that value to complete its computation. Thus, we pass `"ignored"` to `resume` (line 22). Each coroutine has three operations: It starts by suspending itself with `(pause-coroutine)` (line 29), then it prints a character (line 30), and finally it calls itself recursively (line 31). The 11 coroutines created and spawned (lines 35–45) cooperate to print `cooperate.` followed by a newline repeatedly, each printing one character:

```
cooperate.
cooperate.
cooperate.
cooperate.
cooperate.
...
```

Note that these coroutines are cooperatively (i.e., nonpreemptively) scheduled because they suspend themselves after each atomic operation—in this case, printing a character. Coroutines are nonpreemptive and exist at the program level; threads, however, are preemptive and exist at the operating system level. Thus, unlike threads, multiple coroutines in a program cannot utilize more than one of the system's cores. The Lua and Kotlin programming languages support coroutines.

Recall that Ruby supports first-class continuations. The following is a Ruby analog of this Scheme implementation of coroutines:

```
require "continuation" # for callcc
require "thread"       # for Queue

$readyQ = Queue.new

def spawn_coroutine(coroutine)
  $readyQ.push(coroutine)
end

def start_next_ready_coroutine()
  # check for non-empty queue
  coroutine = $readyQ.pop
  coroutine.call()
end

def pause_coroutine()
  callcc{|cc|
    $readyQ.push(cc)
    start_next_ready_coroutine() }
end

def new_coroutine(c)
  f = Proc.new {
    pause_coroutine()
  }
```

```

        print(c)
        f.call()
    }
end

spawn_coroutine(new_coroutine("c"))
spawn_coroutine(new_coroutine("o"))
spawn_coroutine(new_coroutine("o"))
spawn_coroutine(new_coroutine("p"))
spawn_coroutine(new_coroutine("e"))
spawn_coroutine(new_coroutine("x"))
spawn_coroutine(new_coroutine("a"))
spawn_coroutine(new_coroutine("t"))
spawn_coroutine(new_coroutine("e"))
spawn_coroutine(new_coroutine("."))
spawn_coroutine(new_coroutine("\n"))
start_next_ready_coroutine()

```

This implementation of coroutines also underscores and isolates the primary difference between `(call/cc (lambda (k) ...))/(k v)` and `setjmp/longjmp`. A coroutine, in general, has its own stack and program counter. In Scheme, there is only one run-time stack. However, since `call/cc` captures the current continuation *including the stack* at the time `call/cc` is invoked, we can implement coroutines. Specifically, we create multiple functions that dispatch and perform work while pushing and popping activation records on and off the stack. Anytime we invoke `call/cc`, we save a snapshot of the stack at the time of the call. The results, in effect, are coroutines. The same is not possible with `setjmp/longjmp` in C because the individual representation of the stack for each coroutine, which is essential to a coroutine, cannot be captured.

13.6.2 Applications of First-Class Continuations

So far, we have described the following applications of first-class continuations:

- programming abnormal flows of control (e.g., nonlocal exits)
- breakpoints and backtracking (as used in debuggers)
- coroutines (Haynes, Friedman, and Wand 1986) or cooperative, non-preemptive multitasking (Section 13.6.1)

The following are additional applications of continuations:

- threads or noncooperative, preemptive multitasking (Krishnamurthi 2017, Section 14.6.3)
- generators (Section 14.6.2) or iterators (Coyle and Crogono 1991)—see also Friedman and Felleisen (1996b, Chapter 19)
- lazy evaluation (e.g., call-by-name parameters) (Wang 1990)
- callbacks (Section 13.9)
- web servers (i.e., structuring/maintaining interactions between web servers and users) (Graunke et al. 2001; Queinnec 2000)
- human–computer dialogs in user interface software (Pérez-Quñones 1996; Quan et al. 2003)

13.6.3 The Power of First-Class Continuations

So far in this chapter we have demonstrated that first-class continuations are at least as powerful as constructs for nonlocal transfers of control for exception handling since all nonlocal exit constructs (e.g., `setjmp/longjmp`) can be expressed using continuations.

First-class continuations allow the programmer to define any sequential control-flow construct. Any sequential control abstraction (e.g., `gotos`, conditionals, repetition, coroutines) can be defined using first-class continuations (Haynes, Friedman, and Wand 1986, p. 143). The corollary is that continuations are yet another primitive from which to build language features or, in other words, new (specialized) languages, that can be used to solve the particular computing problem at hand. Hence, the importance and merit of first-class continuations are as a general mechanism for control abstraction.

We believe that the primary responsibility of language designers is to provide a flexible basis for the creation of abstractions suitable for various classes of problems. The presence of first[-]class continuations in Scheme ... provides such a basis for the creation of control abstractions ... By using a clean base language with powerful and orthogonal reflection mechanisms (such as `call/cc`), the programmer is able to create control structures far better tuned to the problem at hand. (Haynes, Friedman, and Wand 1986, p. 152)

The power of first-class continuations is derived from both their first-class nature and the ability to call a continuation from outside of its stack lifetime.

“Coroutines, threads, and generators are all conceptually similar: they are all mechanisms to create ‘many little stacks’ instead of having a single, global stack” (Krishnamurthi 2017, p. 122). Further, notice that continuations, call-by-name/need parameters (i.e., lazy evaluation), and coroutines conceptually share common complementary operations for suspending and resuming computation (Table 13.6). Both coroutines (Section 13.6.1) and call-by-name/need parameters can be implemented with continuations (Wang 1990).

Concept	Complementary Operations	
continuations	capture [e.g., <code>(call/cc (λ(k) ...))</code>]	replace [e.g., <code>(k ...)</code>]
↑ <i>lazy evaluation</i> and <i>coroutines</i> can be implemented with <i>continuations</i> ↑		
lazy evaluation	freeze [e.g., <code>(delay ...)</code>]	thaw [e.g., <code>(force ...)</code>]
coroutines	yield [e.g., <code>pause-coroutine</code>]	resume [e.g., <code>start-next-ready-coroutine</code>]

Table 13.6 Different Sides of the Same Coin: Call-By-Name/Need Parameters, Continuations, and Coroutines Share Conceptually Common Complementary Operations

Conceptual Exercises for Section 13.6

Exercise 13.6.1 Explore the relationship of control abstraction to higher-order functions. Can higher-order functions be used to define new control structures in languages without direct support for control abstraction? Explain.

Exercise 13.6.2 (Dybvig 2009, Exercise 3.3.3, p. 77) Explain what happens if a coroutine created by `spawn-coroutine` in the implementation of coroutines in Section 13.6.1 terminates normally (i.e., simply returns without calling `pause-coroutine` again) as demonstrated in the following program. Also, explain why `a` is repeatedly printed twice on each line of output after the first line of output from the following program:

```
> (spawn-coroutine (lambda () (letrec ((f
  (lambda () (pause-coroutine) (display "a") (f)))) (f))))
> (spawn-coroutine (lambda () (letrec ((f
  (lambda () (pause-coroutine) (display "b")))) (f))))
> (spawn-coroutine (lambda () (letrec ((f
  (lambda () (pause-coroutine) (newline) (f)))) (f))))
> (start-next-ready-coroutine)
aba
aa
aa
aa
...
```

Exercise 13.6.3 The following is a proposed solution to Programming Exercise 13.6.9:

```
> (define quit
  (lambda ()
    (cond
      ((null? ready-queue) "end")
      (else (start-next-ready-coroutine)))))

> (spawn-coroutine (lambda () (pause-coroutine) (display "a") (quit)))
> (spawn-coroutine (lambda () (pause-coroutine) (display "b") (quit)))
> (spawn-coroutine (lambda () (pause-coroutine) (display "c") (quit)))
> (spawn-coroutine (lambda () (pause-coroutine) (newline) (quit)))

> (start-next-ready-coroutine)
abc
cba
"end"
>
```

As observed in the output, this proposed solution is not correct. Explain why it is incorrect. Also, explain why the second line of output is the first line of output reversed. *Hint:* Use the Racket debugging facility.

Exercise 13.6.4 Consider the following Scheme program, which appears in Feeley (2004) with minor modifications:

```
(define fail
  (lambda () 'end))
```



```

(define in-range
  (lambda (a b)
    (call/cc
      (lambda (k)
        (enumerate a b k)))))

(define enumerate
  (lambda (a b k)
    (if (> a b)
        (fail)
        (let ((save fail))
          (set! fail
                (lambda ()
                  ;; restore fail to its immediate previous value
                  (set! fail save)
                  (enumerate (+ a 1) b k)))
            (k a))))))

(let ((x (in-range 0 9))
      (y (in-range 0 9))
      (z (in-range 0 9)))
  (write x)
  (write y)
  (write z)
  (newline)
  (fail))

```

This program uses first-class continuations through `call/cc` for backtracking. The continuations are used to simulate a triply nested `for` loop to print the three consecutive digits from 000 to 999:

```

000
001
002
003
...
996
997
998
999

```

This program is the Scheme analog of the following C program:

```

#include<stdio.h>

int main() {
    int i, j, k;

    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)
            for (k = 0; k < 10; k++)
                printf ("%d%d%d\n", i, j, k);
}

```

Trace the Scheme program manually or use the tracing (step-through) feature in the built-in Racket debugging facility to help develop an understanding of how this program functions.

Provide an explanation of how the Scheme program works. Do not restate the obvious (e.g., “the `in-range` function invokes `call/cc` with `lambda (k) ...`”). Instead, provide insight into how this program works.

Programming Exercises for Section 13.6

Exercise 13.6.5 Use `call/cc` to write a Scheme program that prints the integers from 0 to 9 (one per line) once in a loop using iteration. Do not use either recursion or a list.

Exercise 13.6.6 Use `call/cc` to define a while control construct in Scheme without recursion (e.g., `letrec`). Specifically, define a Scheme function `while-loop` that accepts two S-expressions representing Scheme code as arguments, where the first is a loop condition and the second is a loop body. Use the following template for your function and include the missing lines of code (represented as `...`):

```
1 (define ns (make-base-namespace))
2 (eval '(define i 0) ns)
3
4 (define while-loop
5   (lambda (condition body)
6     ...))
```

The following call to `while-loop` prints the integers 0 through 9, one per line, without recursion (e.g., `letrec`):

```
> (while-loop '(< i 10) '(begin
                        (write i)
                        (newline)
                        (set! i (+ i 1))))
0
1
2
3
4
5
6
7
8
9
```

Include lines 1–2 in your program so that calls to `eval` (in the definition of `while-loop`) find bindings for both the `<` function and the identifier `i` in the environment from this example.

Exercise 13.6.7 Define the `while-loop` function from Programming Exercise 13.6.6 without using assignment (i.e., `set!`) and, therefore, without exploiting side effect.

Exercise 13.6.8 The following are two *coroutines* that cooperate to print `I love Lucy.`:

```

(define coroutine1
  (lambda ()
    (display "I ")
    (pause)
    (display "Lucy.")))

(define coroutine2
  (lambda ()
    (display "love ")
    (pause)
    (newline)))

```

The first coroutine prints `I` and `Lucy.` and the second coroutine prints `love` and a newline. The activities of these coroutines are coordinated (i.e., synchronized) by the use of the function `pause`, so that the interleaving of their output operations writes an intelligible sentence to standard output: `I love Lucy.`

Use continuations to provide definitions for `pause` and `resume`, without using recursion (e.g., `letrec`), so that the following main program prints `I love Lucy.`:

```

(define readyq (cons coroutine1 (cons coroutine2 '())))
(resume)

```

Exercise 13.6.9 (Dybvig 2009, Exercise 3.3.3, p. 77) Define a function `quit` in the implementation of coroutines in Section 13.6.1 that allows a coroutine to terminate gracefully without affecting the other coroutines in the program. Be sure to handle the case in which the only remaining coroutine terminates through `quit`.

Exercise 13.6.10 Modify the program from Conceptual Exercise 13.6.4 so that it prints the x , y , and z values where $4 \leq x, y, z \leq 12$, and $x^2 = y^2 + z^2$.

Exercise 13.6.11 Implement the program from Conceptual Exercise 13.6.4 in Ruby using the `callcc` facility.

13.7 Tail Recursion

13.7.1 Recursive Control Behavior

Thus far in our presentation of recursive, functional programming, we have primarily used *recursive control behavior*, where the definition of a recursive function naturally reflects the recursive specification of the function. For instance, consider the following definition of a `factorial` function in Scheme, which naturally mirrors the mathematical definition of a factorial $n! = n * (n - 1)!$:

```

1 (define factorial
2   (lambda (n)
3     (cond
4       ((zero? n) 1) ; base case
5       (else (* n (factorial (- n 1))))) ; inductive step

```

Each call to `factorial` is made with a promise to multiply the value returned by `n` at the time of the call. Examining the run-time behavior of this function with respect to the stack reveals the essence of recursive control behavior:

```

1  (factorial 5)
2  (* 5 (factorial 4))
3  (* 5 (* 4 (factorial 3)))
4  (* 5 (* 4 (* 3 (factorial 2))))
5  (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
6  (* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0)))))) ; base case
7  (* 5 (* 4 (* 3 (* 2 (* 1 1)))))
8  (* 5 (* 4 (* 3 (* 2 1))))
9  (* 5 (* 4 (* 3 2)))
10 (* 5 (* 4 6))
11 (* 5 24)
12 120

```

Notice how execution of this function requires an ever-increasing amount of memory (on the run-time stack) to store the control context as the depth of the recursion increases. In other words, `factorial` is progressively invoked in an ever larger control context as the computation proceeds. That situation occurs because the recursive call to `factorial` is in *operand position*—the return value of each recursive call to `factorial` becomes the second operand to the multiplication operator. The interpreter must save the context around each recursive call because it needs to remember that after the evaluation of the function invocation, the interpreter still needs to finish evaluating the operands and execute the outer call—in this case, the waiting multiplication. Thus, there is a continuation waiting for each recursive call to `factorial` to return. That continuation grows (lines 1–5) until the base case is reached (i.e., `n = 0`; line 6). The computation required to actually compute the factorial is performed as these pending multiplications execute while the activation records for the recursive calls to `factorial` pop off the stack (lines 7–12). Rotating the textual depiction of the control context 90 degrees to the left reveals a parabola capturing the change in the size of the stack as time proceeds during the function execution. Figure 13.7 (left) illustrates this parabola, which describes the general pattern of recursive control behavior.⁵ A function whose control context grows in this manner exhibits *recursive control behavior*. Most recursively defined functions follow this execution pattern.

A key advantage of recursive control behavior is that the definition of the function reflects its specification; a disadvantage is that the amount of memory required to invoke the function is unbounded. However, we can define a recursive version of `factorial` that does not cause the control context to grow; in other words, this version does not require an unbounded amount of memory.

5. This shape is comparable to the contour of an ADRS (Attack–Decay–Sustain–Release) envelope, which depicts changes in the sound of an acoustic musical instrument over time, without the decay phase: The growth of the stack is the analog of the attack phase, the base case is the analog of the sustain phase, and the computation performed as activation records pop off the stack corresponds to the release phase.

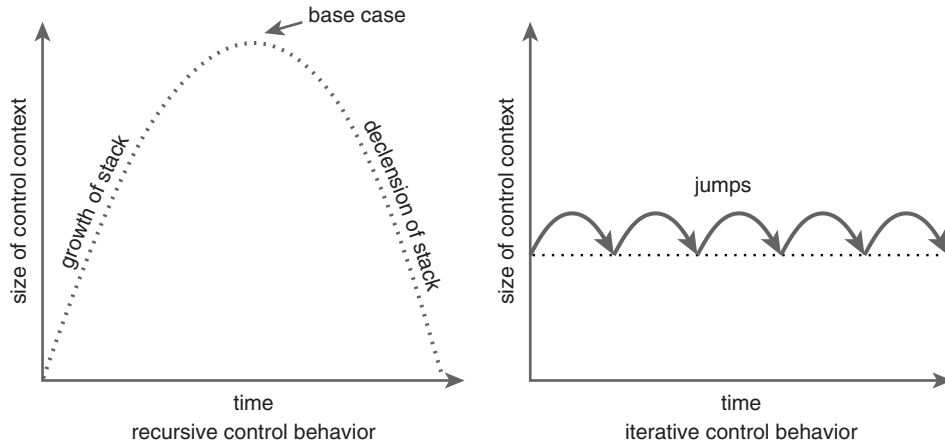


Figure 13.7 Recursive control behavior (left) vis-à-vis iterative control behavior (right).

13.7.2 Iterative Control Behavior

Consider an alternative definition of a `factorial` function:

```

1 (define factorial
2   (lambda (n)
3     (letrec ((fact
4               (lambda (n a)
5                 (cond
6                   ((zero? n) a)
7                   (else (fact (- n 1) (* n a)))))) ; a tail call
8       (fact n 1))))

```

This version defines a nested, recursive function `fact` that accepts an additional parameter `a`, which serves as an *accumulator*. Unlike in the first definition, in this version of `factorial`, successive calls to `fact` do not communicate through a return value (i.e., the factorial resulting from each smaller instance of the problem). Instead, the successive recursive calls now communicate through the additional accumulator parameter.

On line 7, notice that no computation is waiting for each recursive call to `fact` to return; that is, the recursive call to `factorial` is no longer in operand position. In other words, when `fact` calls itself, it does so at the *tail end* of a call to `fact`. Such a recursive call is said to be in *tail position*—in contrast to operand position in which the recursive call to `factorial` is found in the first version—and referred to as a tail call. A function call is a *tail call* if there is no promise to do anything with the returned value. In this version of `factorial`, no promise is made to do anything with the return value other than return it as the result of the current call to `fact`. When the tail call invokes the same function in which it occurs, the approach is referred to as *tail recursion*. Thus, the tail call in this revised version of the `factorial` function uses tail recursion.

The following is a depiction of the control context of a sample execution of this new definition of `factorial`:

```
(factorial 5)
(fact 5 1)
(fact 4 5)
(fact 3 20)
(fact 2 60)
(fact 1 120)
(fact 0 120)
120
```

Figure 13.7 (right) illustrates this graph. Unlike with the execution pattern of the first definition of `factorial`, rotating this textual depiction of the control context 90 degrees to the left reveals a straight line, which indicates the control context remains constant as the function executes. That pattern is a result of *iterative control behavior*, where a recursive function uses a bounded control context. In this case, the function has the potential to run in constant memory space and without the use of a run-time stack because a “procedure call that does not grow control context is the same as a jump” (Friedman, Wand, and Haynes 2001, p. 262). (The strategy used to define this revised version of `factorial` is introduced in Section 5.6.3—through the definition of a list `reverse` function—as *Design Guideline 7: Difference Lists Technique*.)

The use of word *tail* in this context is slightly deceptive because it is not used in the lexicographical context of the function, but rather in the run-time context. In other words, a function that calls itself at the tail end of its definition lexicographically is not necessarily a tail call. For instance, consider line 5 in the first definition of `factorial` in Section 13.7.1 (repeated here): `(else (* n (factorial (- n 1))))`). The recursive call to `factorial` in this line of code appears to be the last step of the function because it is positioned at the rightmost end of the function definition lexicographically, but it is not the final step. The key to determining whether a call is in tail or operand position is the pending continuation. If there is a continuation waiting for the recursive call to return, then the call is in operand position; otherwise, it is in tail position.

As we conclude this section, let us examine two new (tail-recursive) definitions of the `product` function from Section 13.3.1. The following definition is the tail-recursive version of the definition without a nonlocal exit for the exceptional case (i.e., a zero in the input list) from that section:

```
(define product
  (lambda (lon)
    (letrec ((P (lambda (a l)
                  (cond
                    ;; base case
                    ((null? l) a)
                    ;; exceptional case; abnormal = normal flow of control
                    ((zero? (car l)) 0)
                    ;; inductive case; normal flow of control
                    (else (P (* (car l) a) (cdr l)))))))
      (P 1 lon))))
```

While this function is tail recursive and exhibits iterative control behavior, it may perform unnecessary multiplications if the input list contains a zero. The following definition is the tail-recursive version of the definition using a continuation captured with `call/cc` to perform a nonlocal exit in the exceptional case from Section 13.3.1:

```
(define product
  (lambda (lon)
    (call/cc
      ;; break stores the current continuation
      (lambda (break)
        (letrec ((P (lambda (a l)
                      (cond
                        ;; base case
                        ((null? l) a)
                        ;; exceptional case; abnormal != normal flow of control
                        ((zero? (car l)) (break 0))
                        ;; inductive case; normal flow of control
                        (else (P (* (car l) a) (cdr l)))))))
          (P 1 lon))))))
```

This definition, like the first one, is tail recursive, exhibits iterative control behavior, and may perform unnecessary multiplications if the input list contains a zero. However, this version avoids returning through all of the activation records built up on the call stack when a zero is encountered in the list.

13.7.3 Tail-Call Optimization

If a recursive function defined using tail recursion exhibits iterative control behavior, it has the *potential* to run in constant memory space. The use of tail recursion implies that no computations are waiting for the return value of each recursive call, which in turn means the function that made the recursive call can be popped off the run-time stack. However, even though tail recursion eliminates the buildup of pending computations on the run-time stack waiting to complete once the base case is reached, the activation records for each recursive tail call are still on the stack. Each activation record simply receives the return value from the function it calls and returns this value to the function that called it.

Tail-call optimization (TCO) eliminates the implicit function return in a tail call and eliminates the need for a run-time stack.⁶ Thus, TCO enables (recursive) functions to run in constant space—rendering recursion as efficient as iteration. The Scheme, ML, and Lua programming languages use TCO. Languages supporting functional programming can be implemented using CPS and TCO in concert (Appel 1992).

Note that TCO is not just applicable to tail-recursive calls. It is applicable to *all* tail calls—even non-recursive ones.⁷ As a consequence, a stack is unnecessary for

6. Tail-call optimization is also referred to as *tail-call elimination*. Since the caller jumps to the callee, the tail call is essentially eliminated.

7. It is tail-call optimization, not tail-recursion optimization.

a language to support functions. Thus, TCO should be used not just in languages where recursion is the primary means of repetition (e.g., Scheme and ML), but in any language that has functions. Consider the following `isodd` and `iseven` Python functions:

```
>>> def isodd (n):
...     if n == 0:
...         return False
...     else:
...         return iseven (n-1)
...
>>> def iseven (n):
...     if n == 0:
...         return True
...     else:
...         return isodd (n-1)
...
>>> print (iseven(1000000000))
...
RecursionError: maximum recursion depth exceeded in comparison
```

The call to `isodd` in the body of the definition of `iseven` is not tail recursion—it is simply a tail call. The same is true for the call to `iseven` in the body of `isodd`. Thus, neither of these functions is recursive independently of each other (i.e., neither function has a call to itself). They are just mutually dependent on each other or *mutually recursive*. Since Python does not use TCO on these non-recursive functions, this program does not run in constant memory space or without a stack.

The Scheme rendition of this Python program runs in constant space without a stack:

```
> (letrec ((iseven? (lambda (n) (if (zero? n) #t (isodd? (- n 1)))))
      (isodd? (lambda (n) (if (zero? n) #f (iseven? (- n 1)))))
      (iseven? 1000000000))
#t
```

Thus, not only can TCO be used to optimize non-recursive functions, but it should be applied so that the programmer can use both individual non-recursive functions and recursion without paying a performance penalty.

Tail-call optimization makes functions using only tail calls iterative (in run-time behavior) and, therefore, more efficient. The revised definition of `factorial` using tail recursion and exhibiting iterative control behavior does not have a growing control context, so it now has the potential to be optimized to run in constant space. However, it no longer mirrors the recursive specification of the problem. By using tail recursion, we trade off function readability/writability for the possibility of space efficiency. Even so, it is possible to make recursion iterative while maintaining the correspondence of the code to the mathematical definition of the function (Section 13.8). Table 13.7 summarizes the relationship between the type of function call and the control behavior of a function.

The programming technique called *trampolining* (i.e., converting a program to *trampolined style*) can be used to achieve the same effect as tail-call optimization

Functions with non-tail calls exhibit recursive control behavior:

Non-tail calls imply recursive control behavior.

Functions with tail calls exhibit iterative control behavior:

Tail calls imply iterative control behavior.

Iterative control behavior is not sufficient to eliminate the run-time stack.

Iterative control behavior + tail-call optimization = no run-time stack needed.

Table 13.7 Non-tail Calls/Recursive Control Behavior Vis-à-Vis Tail Calls/Iterative Control Behavior

in a language that does not implement TCO. The underlying idea is to replace a tail-recursive call to a function with a thunk to invoke that function. The thunk is then subsequently applied in a loop. Consider the following trampolined version of the previous odd/even program in Python that would not run:

```

1  from collections import namedtuple
2
3  Thunk = namedtuple('Thunk', 'func args')
4
5  def trampoline(x):
6      while (isinstance(x, Thunk)):
7          x = x.func(*x.args)
8      return x
9
10 def isoddtampoline(n):
11     if n == 0:
12         return False
13     else:
14         return Thunk(func=iseventrampoline, args=[n-1])
15
16 def iseventrampoline(n):
17     if n == 0:
18         return True
19     else:
20         return Thunk(func=isoddtampoline, args=[n-1])
21
22 def isodd(n):
23     return trampoline(Thunk(func=isoddtampoline, args=[n]))
24
25 def iseven(n):
26     return trampoline(Thunk(func=iseventrampoline, args=[n]))

```

In this program, `Thunk` is a `namedtuple`, which behaves like an unnamed tuple, but with field names (line 3). We use this unnamed tuple to create the thunks that obviate the would-be recursive calls to `isodd` and `iseven` (lines 14 and 20, respectively). In lines 5–8, the function `trampoline` performs the computation iteratively, thereby acting as a trampoline. Therefore, we are able to write tail calls that execute without a stack:

```

>>> print (iseven(1000000000))
True

```

13.7.4 Space Complexity and Lazy Evaluation

There is an interesting relationship between *tail recursion* and *lazy evaluation* in regard to the space complexity of a program. Programmers of lazy languages must have a greater awareness of the space complexity of a program. Consider the following function `len` defined using tail recursion in Haskell:

```
1 Prelude> :{
2 Prelude| len [] acc = acc
3 Prelude| len (x:xs) acc = len xs (acc + 1)
4 Prelude| :}
```

Invoking this tail-recursive definition of `len` in Haskell results in a stack overflow:

```
5 Prelude> len [1..1000000000] 0
6 *** Exception: stack overflow
```

The following is a trace of the expansion of the calls to `len`:

```
len [1,2,3..20000] 0
len [2,3..20000] (0 + 1)
len [3..20000] (0 + 1 + 1)
len [..20000] (0 + 1 + 1 + 1)
len [20000] (0 + 1 + 1 + 1 ... + 1)
len [] (0 + 1 + 1 + 1 ... + 1)
20000
```

This function is tail recursive and appears to run in constant memory space—the stack never grows beyond one frame. However, the size of the second argument to `len` is expanding because of the lazy (as opposed to eager) evaluation strategy used. Although the interpreter no longer must save the pending computations—in this case, the additions—on the stack, the interpreter stores a new thunk for the expression `(acc + 1)` for every recursive call to `len`. Forcing the evaluation of the second parameter to `len` (i.e., making the second parameter to `len` strict) prevents the stack overflow. We can force a parameter to be strict by prefacing it with `$!` (as demonstrated in Section 12.5.5):

```
Prelude> :{
Prelude| len [] acc = acc
Prelude| len (x:xs) acc = len xs $! (acc + 1)
Prelude| :}
Prelude> len [1..1000000000] 0
1000000000
```

The following trace illustrates how the evaluation of the second parameter to `len` is forced for each recursive call:

```
len [1,2,3..1000000000] 0
len [2,3..1000000000] (0 + 1)
len [2,3..1000000000] 1
len [3..1000000000] (1 + 1)
len [3..1000000000] 2
len [..1000000000] (2 + 1)
len [..1000000000] 3
```

```

len [1000000000] (999999998 + 1)
len [1000000000] 999999999
len [] (999999999 + 1)
len [] 1000000000
1000000000

```

In general, it is often recommended to make an accumulator parameter strict when defining a tail-recursive function in a lazy language.

The recursive pattern used in this definition of `len` is encapsulated in the higher-order folding functions. The accumulator parameter is the analog of the initial value passed to `foldl` or `foldr`. Since the combining function for `len` [i.e., `(\x acc -> acc+1)`] does not use the elements in the input list, we can define `len` using either `foldl` or `foldr`:

```
1 len = foldr (\x acc -> acc+1) 0
```

Even though this definition of `len` uses the accumulator approach in the combining function passed to `foldr` (i.e., its first parameter), its invocation results in a stack overflow:

```

2 Prelude> len 0 [1..1000000000]
3 *** Exception: stack overflow
4
5 Prelude> foldr (\x acc -> acc+1) 0 [1..1000000000]
6 *** Exception: stack overflow

```

This is because `foldr` *is not* defined using tail recursion:

```

foldr f i [] = i
foldr f i (x:xs) = f x (foldr f i xs)

foldr (\x acc -> acc+1) 0 [1..1000000000]
f 1 (foldr f 0 [2..1000000000])
f 1 (f 2 (foldr f 0 [3..1000000000]))
f 1 (f 2 ... (foldr f 0 [1000000000]))
f 1 (f 2 ... (f 1000000000 (foldr f 0 [])))
f 1 (f 2 ... (f 1000000000 0))
f 1 (f 2 ... 1)
f 1 1000000000
1000000000

```

Conversely, `foldl` *is* defined using tail recursion:

```

foldl f i [] = i
foldl f i (x:xs) = foldl f (f i x) xs

```

Thus, we can define a more space-efficient version of `len` using `foldl`:

```
1 Prelude> len = foldl (\acc x -> acc+1) 0
```

Notice that in this definition of `len`, we must reverse the order of the parameters to the combining function (i.e., `acc` and `x`). However, this version produces a stack overflow:

```

2 Prelude> len 0 [1..1000000000]
3 *** Exception: stack overflow
4
5 Prelude> foldl (\acc x -> acc+1) 0 [1..1000000000]
6 *** Exception: stack overflow

```

The following is a trace of this invocation of `len`:

```

foldl (\acc x -> acc+1) 0 [1..1000000000]
foldl f (f 0 1) [2..1000000000]
foldl f (f (f 0 1) 2) [3..1000000000]
foldl f (f ... (f (f 0 1) 2) ... 999999999) [1000000000]
foldl f (f (f ... (f (f 0 1) 2) ... 999999999) 1000000000) []
(f (f ... (f (f 0 1) 2) ... 999999999) 1000000000)
(f (f ... (f (1 2) ... 999999999) 1000000000)
(f (f ... (2 ... 999999999) 1000000000)
(f (f 999999998 999999999) 1000000000)
(f 999999999 1000000000)
1000000000

```

While `foldl` does use tail recursion, it also uses lazy evaluation. Thus, this invocation of `len` results in a stack overflow because a thunk is created for the second parameter to `foldl`—that is, the evaluation of the combining function `(f i x)`—for every recursive call and the second parameter continues to grow. The invocation of `len` builds up a lengthy chain of thunks that will eventually evaluate to the length of the list rather than maintaining a running length. Thus, this version of `len` behaves the same as the first version of `len` in this subsection.

To solve this problem, we need a version of `foldl` that is both tail recursive *and* strict in its second parameter:

```

1 Prelude> :{
2 Prelude| foldl' f i [] = i
3 Prelude| foldl' f i (x:xs) = (foldl' f $! f i x) xs
4 Prelude| :}
5
6 Prelude> :type foldl'
7 foldl' :: (a -> t -> a) -> a -> [t] -> a

```

Consider the following invocation of `foldl'`:

```

8 Prelude> foldl' (\acc x -> acc+1) 0 [1..1000000000]
9 1000000000

```

The following is a trace of this invocation of `foldl'`:

```

foldl' (\acc x -> acc+1) 0 [1..1000000000]
foldl' f 1 [2..1000000000]
foldl' f (f 1 2) [3..1000000000]
foldl' f 2 [3..1000000000]
...
foldl' f (f 999999998 999999999) [1000000000]
foldl' f 999999999 [1000000000]
foldl' f (f 999999999 1000000000) []
foldl' f 1000000000 []
1000000000

```

While `foldr` should be avoided for computing the length of a list because it is not defined using tail recursion, `foldr` should not be avoided in all cases. For instance, consider the following function, which determines whether all elements of a list are `True`:

```
1 Prelude> allTrue = foldr (&&) True
```

Since `(&&)` is non-strict in its second parameter, use of `foldr` obviates further exploration of the list as soon as a `False` is encountered:

```
2 Prelude> allTrue [False, True, True, True]
3 False
```

The following is a trace of this invocation of `allTrue`:

```
Prelude> foldr (&&) True (False : [True, True, True])
False

Prelude> False && (foldr (&&) True [True, True, True])
False
```

In this case, `foldr` does not build up the ability to perform the remaining computations. The same is not true of `foldl'`. For instance:

```
Prelude> foldl' (&&) True [False, True, True, True]
False
```

The following is a trace of this invocation of `foldl'`:

```
foldl' (&&) True [False, True, True, True]
foldl' (&&) (True && False) [True, True, True]
foldl' (&&) False [True, True, True]
foldl' (&&) (False && True) [True, True]
foldl' (&&) False [True, True]
foldl' (&&) (False && True) [True]
foldl' (&&) False [True]
foldl' (&&) (False && True) []
foldl' (&&) False []
False
```

Even though this version runs in constant space because `foldl'` is defined using tail recursion, it examines every element of the input list. Thus, `foldr` is preferred in this case. Similarly, the built-in Haskell function `concat` uses `foldr` even though `foldr` is not defined using tail recursion:

```
1 Prelude> concat = foldr (++) []
2
3 Prelude> :type concat
4 concat :: Foldable t => t [a] -> [a]
```

The following is an invocation of `concat`:

```
5 Prelude> concat [[1], [2], [3], [4], [5]]
6 [1, 2, 3, 4, 5]
```

Tracing this invocation of `concat` reveals why `foldr` is used in its definition:

```

7  Prelude> foldr (++) [] ([1]:[2],[3],[4],[5])
8    [1,2,3,4,5]
9
10 Prelude> [1] ++ (foldr (++) [] [[2],[3],[4],[5]])
11    [1,2,3,4,5]
12
13 Prelude> (1:[]) ++ (foldr (++) [] [[2],[3],[4],[5]])
14    [1,2,3,4,5]
15
16 Prelude> 1 : ([] ++ (foldr (++) [] [[2],[3],[4],[5]]))
17    [1,2,3,4,5]
```

Unlike the expansion for the invocation of the definition of `len` using `foldr` in this subsection, the expression on line 16 is as far as the interpreter will evaluate the expression until the program seeks to examine an element in the tail of the result. Since we can garbage collect the first cons cell of this result before we traverse the second, `concat` not only runs in constant stack space, but also accommodates infinite lists. By contrast, neither `foldl'` nor `foldl` can handle infinite lists because the left-recursion in the definition of either would lead to infinite recursion. For instance, the following invocation of `foldl` does not terminate (until the stack overflows):

```

Prelude> foldl (&&) False (repeat False)
^CInterrupted.
```

(Note that `repeat e` is an infinite list, where every element is `e`.) However, the following invocation of `foldr` returns `False` immediately:

```

Prelude> foldr (&&) False (True:False:(repeat False))
False
```

Since `(&&)` is non-strict in its second parameter, we do not have to evaluate the rest of the `foldr` expression to determine the result of `allTrue`. Similarly, since `(++)` is non-strict in its second parameter, we do not have to evaluate the rest of the `foldr` expression to determine the head of the result of `concat`. However, because the combining function `(\acc x -> acc+1)` in `len` must run on every element of the list before a list length can be computed, we require the result of the entire `foldr` to compute a final length. Thus, in that case, `foldl'` is a preferable choice.

Table 13.8 summarizes these `fold` higher-order functions with respect to evaluation strategy in eager and lazy languages. Defining tail-recursive functions in languages with a lazy evaluation strategy requires more attention than doing so in languages with an eager evaluation strategy. Using `foldl'` requires constant stack space, but necessitates a complete expansion even for combining functions that are non-strict in their second parameter. However, even though `foldr` is not defined using tail recursion, it can run efficiently if the combining function is non-strict in its second parameter. More generally, the space complexity of lazy programs is complex.

Eager Language		Lazy Language	
<code>foldr</code>	non-tail recursive and strict	<code>foldr</code>	non-tail recursive and non-strict
<code>foldl</code>	tail recursive and strict	<code>foldl</code>	tail recursive and non-strict
		<code>foldl'</code>	tail recursive and strict

Table 13.8 Summary of Higher-Order `fold` Functions with Respect to Eager and Lazy Evaluation

We offer some general guidelines for when `foldr`, `foldl`, and `foldl'` are most appropriate in designing functions (assuming the use of each function results in the same value).

Guidelines for the Use of `foldr`, `foldl`, and `foldl'`

- In a language using eager evaluation, when both `foldl` and `foldr` produce the same result, `foldl` is more efficient because it uses tail recursion and, therefore, runs in constant stack space.
- In a language using lazy evaluation, when both `foldl'` and `foldr` produce the same result, examine the context:
 - If the combining function passed as the second argument to the higher-order folding function is strict and the input list is finite, always use `foldl'` so that the function will run in constant space because `foldl'` is both tail recursive *and* strict (unlike `foldl`, which is tail recursive and non-strict). Passing such a function to `foldr` will always require linear stack space, so it should be avoided.
 - If the combining function passed as the second argument to the higher-order folding function is strict and the input list is infinite, always use `foldr`. While the function will not run in constant space (like `foldl'`), it will return a result, unlike `foldl'`, which will run forever, albeit in constant space.
 - If the combining function passed as the second argument to the higher-order folding function is non-strict, always use `foldr` to support both the streaming of the input list, where only a part of the list must reside in memory at a time, and infinite lists. In this situation, if `foldl'` is used, it will never return a result, though the function will run in constant memory space.
 - In general, avoid the use of the function `foldl`.

These guidelines are presented as a decision tree in Figure 13.8.

Programming Exercises for Section 13.7

Exercise 13.7.1 Unlike a language with an eager evaluation strategy, in a lazy language, even if the operator to be folded is associative `foldl'` and `foldr` may not be used interchangeably depending on the context. Demonstrate this by

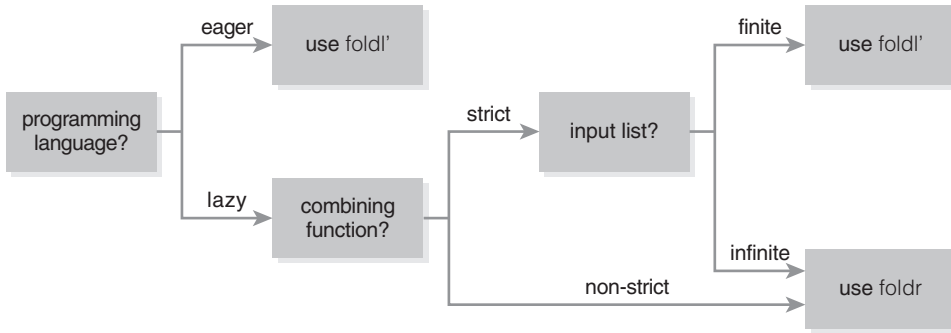


Figure 13.8 Decision tree for the use of `foldr`, `foldl`, and `foldl'` in designing functions (assuming the use of each function results in the same value).

folding the same associative operator [e.g., `(++)`] across the same list with the same initial value using `foldl'` and `foldr`. Use a different associative operator than any of those already given in this section. Use program comments to clarify your demonstration. *Hint:* Use `repeat` in conjunction with `take` to generate finite lists to be used as test lists in your example; use `repeat` to generate infinite lists to be used as test lists in your example and `take` to generate output from an infinite list that has been processed.

Exercise 13.7.2 Explain why `map1 f = foldr ((:).f) []` in Haskell can be used as a replacement for the built-in Haskell function `map`, but `map1 f = foldl ((:).f) []` cannot.

Exercise 13.7.3 Demonstrate how to overflow the control stack in Haskell using `foldr` with a function that is made strict in its second argument with `$!`.

Exercise 13.7.4 Define a *recursive* Scheme function `square` using tail recursion that accepts only a positive integer `n` and returns the square of `n` (i.e., n^2). Your definition of `square` must contain only one recursive helper function bound in a `letrec` expression that does not require an unbounded amount of memory.

Exercise 13.7.5 Define a recursive Scheme function `member-tail` that accepts an atom `a` and a list of atoms `lat` and returns the integer position of `a` in `lat` (using zero-based indexing) if `a` is a member of `lat` and `#f` otherwise. Your definition of `member-tail` must use *tail recursion*. See examples in Programming Exercise 13.3.6.

Exercise 13.7.6 The *Fibonacci* series 0, 1, 1, 2, 3, 5, 8, 13, 21, ... begins with the numbers 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers. The Fibonacci series occurs in nature and, in particular, describes a form of a spiral. The ratio of the successive Fibonacci numbers converges on a constant value of 1.618.... This number, too, repeatedly occurs in nature and has been called the golden ratio or the golden mean. Humans tend to find the golden mean aesthetically pleasing. Architects

often design windows, rooms, and buildings with a golden mean length/width ratio.

Define a Scheme function `fibonacci`, using only one *tail call*, that accepts a non-negative integer `n` and returns the `n`th Fibonacci number. Your definition of `fibonacci` must run in $O(n)$ and $O(1)$ time and space, respectively. You may define one helper function, but it also must use only one tail call. Do not use more than 10 lines of code. Your function must be invocable.

Examples:

```
> (fibonacci 0)
0
> (fibonacci 1)
1
> (fibonacci 2)
1
> (fibonacci 3)
2
> (fibonacci 4)
3
> (fibonacci 5)
5
> (fibonacci 6)
8
> (fibonacci 7)
13
> (fibonacci 8)
21
> (fibonacci 20)
6765
```

Exercise 13.7.7 Complete Programming Exercise 13.7.6 in Haskell or ML.

Exercise 13.7.8 Define a `factorial` function in Haskell using a higher-order function and one line of code. The `factorial` function accepts only a number `n` and returns `n!`. Your function must be as efficient in space as possible.

Exercise 13.7.9 Define a function `insertionsort` in Haskell that accepts only a list of integers, insertion sorts that list, and returns the sorted list. Specifically, first define a function `insert` with fewer than five lines of code that accepts only an integer and a sorted list of integers, in that order, and inserts the first argument in its sorted position in the list in the second argument. Then define `insertionsort` in one line of code using this helper function and a higher-order function. Your function must be as efficient as possible in both time and space. *Hint:* Investigate the use of `scanr` to trace the progressive use of `insert` to sort the list.

13.8 Continuation-Passing Style

13.8.1 Introduction

We can make all function calls tail calls by first encapsulating any computation remaining after each call—the “what to do next”—into an explicit, reified

continuation and then passing that continuation as an extra argument in each tail call. In other words, we can make the *implicit* continuation of each called function *explicit* by packaging it as an additional argument passed in each function call. This approach is called *continuation-passing style* (CPS), as opposed to *direct style*. We begin by presenting some examples to acclimate readers to the idea of passing an additional argument to each function, with that argument capturing the continuation of the call to the function. Consider the following function definitions:

```

1  > (define +cps
2      (lambda (x y k)
3        (k (+ x y))))
4
5  > (define *cps
6      (lambda (x y k)
7        (k (* x y))))
8
9  > (* 3 (+ 1 2)) ; direct style (i.e., non-CPS)
10 9
11 > (+cps 1 2 (lambda (returnvalue)
12     (*cps 3 returnvalue (lambda (x) x)))) ; CPS
13 9

```

Here, `+cps` and `*cps` are the CPS analogs of the `+` and `*` operators, respectively, and each accepts an additional parameter representing the continuation. When `+cps` is invoked on line 11, the third parameter specifies how to continue the computation. Specifically, the third parameter is a `lambda` expression that indicates what should be done with the return value of the invocation of `+cps`. In this case, the return value is passed to `*cps` with 3. Notice that the continuation of `*cps` is the identity function because we simply want to return the value. Consider the following expression:

```

1  (let* ((inc (lambda (n) (+ 1 n)))
2         (a (lambda (n) (* 2 (inc n))))
3         (b (lambda (n) (a (* 4 n)))))
4    (+ 3 (b 5)))

```

The function `b` calls the function `a` in tail position on line 3. As a result, the continuation of `a` is the same as that of `b`. In other words, `b` does not perform any additional work with the return value of `a`. The same is not true of the call to the function `inc` in the function `a` on line 2. The call to `inc` on line 2 is in operand position. Thus, when `a` receives the result of `inc`, the function `a` performs an additional computation—in this case, a multiplication by 2—before returning to its continuation. Here, the implicit continuation of the call to

- `inc` is `(lambda (v) (+ 3 (* 2 v)))`
- `a` is `(lambda (v) (+ 3 v))`
- `b` is `(lambda (v) (+ 3 v))`

We can rewrite this entire `letrec` expression in CPS by replacing these implicit continuations with explicit `lambda` expressions:

```

1  (let* ((inc (lambda (n k) (k (+ 1 n))))
2        (a (lambda (n k) (inc n (lambda (v) (k (* 2 v))))))
3        (b (lambda (n k) (a (* 4 n) k))))
4    (b 5 (lambda (v) (+ 3 v))))

```

When `k` is called on line 1, it is bound to the continuation of `inc`: `(lambda (v) (+ 3 (* 2 v)))`. Notice that an explicit continuation in CPS is represented as a λ -expression. While functions defined in CPS are, in general, less readable/writable than those defined using direct style, CPS implies all tail calls and, in turn, use of TCO.

Notice also that the functions written in direct style used here as examples are non-recursive. When rewritten in CPS, they are still non-recursive. However, they make only tail calls, which can be eliminated with TCO—obviating the need for a run-time stack, even for non-recursive functions. Moreover, abnormal flows of control can be programmed in CPS.

13.8.2 A Growing Stack or a Growing Continuation

While `factorial` is a simple function, defining it in CPS is instructive for better understanding the essence of CPS. Consider the following definition of a `factorial` function using recursive control behavior:

```

(define factorial
  (lambda (n)
    (cond
      ((zero? n) 1)
      (else (* n (factorial (- n 1)))))))

```

The following is an attempt at a CPS rendition of this function:⁸

```

1  (define factorial
2    (letrec
3      ((fact-cps (lambda (n growing-k)
4        (cond
5          ((eqv? n 1) (growing-k 1))
6          (else (fact-cps (- n 1) ; a tail call
7            (lambda (rtval) (* rtval (growing-k n)))))))
8      (lambda (n)
9        (cond
10         ((zero? n) 1)
11         (else (fact-cps n (lambda (x) x)))))))

```

The most critical lines of code in this definition are lines 6 and 7 where the recursive call is made and the explicit continuation is passed, respectively. Lines 6–7 conceptually indicate: take the result of $(n-1)!$ and continue the computation by *first* continuing the computation of $n!$ with n and then multiplying the result by $(n-1)!$. In other words, when we call `(growing-k n)`, we are passing the input parameter to `fact-cps` in an *unmodified* state to its continuation. This approach is tantamount to writing `(lambda (x k) (k x))`. The following series of expansions demonstrates the unnaturalness of this approach:

⁸ The `factorial` functions presented in this section are not entirely defined in CPS because the primitive functions (e.g., `zero?`, `*`, and `-`) are not defined in CPS. See Section 13.8.3 and Programming Exercise 13.10.26.

```

(factorial 3)

(fact-cps 3 (lambda (x) x))

(fact-cps 2 (lambda (rtnval) (* rtnval ((lambda (x) x) 3))))

(fact-cps 1 (lambda (rtnval) (* rtnval
  ((lambda (rtnval) (* rtnval ((lambda (x) x) 3))) 2))))

((lambda (rtnval) (* rtnval
  ((lambda (rtnval) (* rtnval ((lambda (x) x) 3))) 2))) 1)

(* 1 ((lambda (rtnval) (* rtnval ((lambda (x) x) 3))) 2))

(* 1 (* 2 ((lambda (x) x) 3)))

(* 1 (* 2 3))

(* 1 6)

6

```

While defined using tail recursion, this first version of `fact-cps` runs contrary to the spirit of CPS. The definition does not embrace the naturalness of the continuation of the computation.

Consider replacing lines 6–7 in this first version of `fact-cps` with the following lines:

```

6           (else (fact-cps (- n 1) ; a tail call
7               (lambda (rtnval) (growing-k (* rtnval n))))))

```

This second definition of `fact-cps` maintains the natural continuation of the computation. Lines 6–7 conceptually indicate: take the result of $(n-1)!$ and continue the computation by *first* multiplying $(n-1)!$ by n and then passing that result to the continuation of $n!$. The following series of expansions demonstrates the run-time behavior of this version:

```

(factorial 3)

(fact-cps 3 (lambda (x) x))

(fact-cps 2 (lambda (rtnval) ((lambda (x) x) (* rtnval 3))))

(fact-cps 1 (lambda (rtnval) ((lambda (rtnval)
  ((lambda (x) x) (* rtnval 3))) (* rtnval 2))))

((lambda (rtnval) ((lambda (rtnval)
  ((lambda (x) x) (* rtnval 3))) (* rtnval 2))) 1)

((lambda (rtnval) ((lambda (x) x) (* rtnval 3))) (* 1 2))

((lambda (rtnval) ((lambda (x) x) (* rtnval 3))) 2)

((lambda (x) x) (* 2 3))

((lambda (x) x) 6)

6

```

This definition of `fact-cps` both uses tail recursion—`fact-cps` is always on the leftmost side of any expression in the expansion—and maintains the natural

continuation of the computation. However, this second version grows the passed continuation `growing-k` in each successive recursive call. (The first version of `fact-cps` incidentally does too.) Thus, while the second version is more naturally CPS, it is not space efficient. In an attempt to keep the run-time stack of constant size (through the use of CPS), we have shifted the source of the space inefficiency from a growing stack to a growing continuation. The continuation argument is a closure representation of the stack (Section 9.8.2).

Thus, this second version of `fact-cps` demonstrates that use of tail recursion is not sufficient to guarantee space efficiency at run-time. Even though both calls to `fact-cps` and `growing-k` are in tail position (lines 6 and 7, respectively), the run-time behavior of `fact-cps` is essentially the same as that of the non-CPS version of `factorial` given at the beginning of this subsection—the expansion of the run-time behavior of each function shares the same shape. The use of continuation-passing style in this second version of `fact-cps` explicitly reifies the run-time stack in the interpreter and passes it as an additional parameter to each recursive call. Just as the stack grows when running a function defined using recursive control behavior, in the `fact-cps` function the additional parameter representing the continuation—the analog of the stack—is also growing because it is encapsulating the continuation from the prior call.

Let us reconsider the definition of a `factorial` function using tail recursion (in Section 13.7.2 and repeated here):

```

1 (define factorial
2   (lambda (n)
3     (letrec ((fact
4               (lambda (n a)
5                 (cond
6                   ((zero? n) a)
7                   (else (fact (- n 1) (* n a))))))) ; a tail call
8     (fact n 1))))

```

This function is not written using CPS, but is defined using tail recursion. The following is a CPS rendition of this version of `factorial`:

```

1 (define factorial
2   (lambda (n)
3     (letrec ((fact-cps
4               (lambda (n a constant-k)
5                 (cond
6                   ((zero? n) (constant-k a))
7                   (else (fact-cps (- n 1) (* n a) constant-k)))))) ; a CPS tail call
8     (fact-cps n 1 (lambda (x) x))))

```

Here, unlike the first version of the `fact-cps` function defined previously, this third version does not grow the passed continuation `constant-k`. In this version, the continuation passed is constant across the recursive calls to `fact-cps` (line 7):

```

(factorial 3)

(fact-cps 3 1 (lambda (x) x))

(fact-cps 2 3 (lambda (x) x))

(fact-cps 1 6 (lambda (x) x))

```

```
(fact-cps 0 6 (lambda (x) x))

((lambda (x) x) 6)

6
```

A constant continuation passed in a tail call is necessary for efficient space performance. Passing a constant continuation in a non-tail call is insufficient. For instance, consider replacing lines 6–7 in the first version of the `fact-cps` with the following lines (and renaming the continuation parameter from `growing-k` to `constant-k`):

```
6          (else (constant-k (fact-cps (- n 1) ; a non-tail call
7                (lambda (rtnval) (* rtnval n)))))))))
```

Since `constant-k` is not embedded in the continuation passed to each recursive call to `fact-cps`, the continuation is not growing. However, in this fourth version of `fact-cps`, the recursive call to `fact-cps` is not in tail position. Without a tail call, the function displays recursive control behavior, where the stack grows unbounded. Thus, the use of a constant continuation in a non-tail call is insufficient.

In summary, continuation-passing style implies the use of a tail call, but the use of a tail call does not necessarily imply a continuation argument that is bounded throughout execution (e.g., the second version of `fact-cps` in Section 13.8.2) or the use of CPS at all (e.g., `factorial` in Section 13.7.2). We desire a function embracing the spirit of CPS where, ideally, the continuation passed in the tail call is bounded. The third version of `fact-cps` meets these criteria—see the row labeled “Third/ideal version” in Table 13.9. Of course, as with any tail calls, we should apply TCO to eliminate the need for a run-time stack to execute functions written in CPS. Table 13.9 summarizes the versions of the `fact-cps` function presented here through these properties. Table 13.10 highlights some key points about interplay of tail recursion/calls, recursive/iterative control behavior, TCO, and CPS.

13.8.3 An All-or-Nothing Proposition

Consider the following definition of a remainder function using CPS:

```
1 > (define remainder-cps
2   (lambda (n d k)
3     (cond
4       ((< n d) (k n))
5       (else (remainder-cps (- n d) d k))))))
6
7 > (remainder-cps 7 3 (lambda (x) x))
8 1
```

Notice that the primitive operators used in the definition of `remainder-cps` (e.g., `<` on line 4 and `-` on line 5) are not written in CPS. To maximize the benefits of CPS discussed in this chapter, all function calls in a program should use CPS. In other words, continuation-passing style is an all-or-nothing proposition, especially to obviate the need for a run-time stack of activation records. The following is a complete CPS rendition of the `remainder-cps` function, including definitions of

Version of fact-cps in Section 13.8.2	fact-cps is in Tail Position	Call to Continuation	(Nongrowing) Constant Continuation	CPS
First version	✓	×	×	×
Second version	✓	✓	×	✓
Third/ideal version	✓	✓	✓	✓
Fourth version	×	✓	✓	×

Table 13.9 Properties of the four versions of `fact-cps` presented in Section 13.8.2.

<i>Iterative control behavior maintains a bounded control context.</i>
<i>Tail-call optimization eliminates the need for a run-time call stack.</i>
<i>A function can exhibit iterative control behavior, but still needs a call stack to run.</i>
<i>Tail-call optimization can and should be applied to all tail calls, not just recursive ones.</i>
<i>Continuation-passing style implies the use of a tail call.</i>
<i>Neither tail recursion nor a non-recursive tail call implies CPS or a bounded continuation argument.</i>

Table 13.10 Interplay of Tail Recursion/Calls, Recursive/Iterative Control Behavior, Tail-Call Optimization, and Continuation-Passing Style

the less than and subtraction operators in CPS (the `<cps` and `-cps` functions on lines 1–3 and 5–7, respectively):

```

1 (define <cps
2   (lambda (x y k)
3     (k (< x y))))
4
5 (define -cps
6   (lambda (x y k)
7     (k (- x y))))
8
9 (define remainder-cps
10  (lambda (n d k)
11    (<cps n d (lambda (bool)
12               (cond
13                 (bool (k n))
14                 (else (-cps n d
15                           (lambda (rtnval) (remainder-cps rtnval d k))))))))))
15

```

For purposes of clarity of presentation, the primitives used in this chapter are not defined in CPS (Programming Exercise 13.10.26).

13.8.4 Trade-off Between Time and Space Complexity

Defining the `product` function that invokes `call/cc` in Section 13.3.1 using CPS, while retaining the feature that no multiplications are performed if a zero

is encountered in the list, is instructive for highlighting differences in the use of `call/cc` and CPS. The following is a definition of `product-cps`:

```

1  (define product-cps
2    (lambda (lon k)
3      (let ((break k))
4        (letrec ((P
5                   (lambda (l growing-k)
6                     (cond
7                       ((null? l) (growing-k 1))
8                       ((zero? (car l)) (break
9                                     "Encountered a zero in the input list."))
10                      (else (P (cdr l)
11                              (lambda (x) (growing-k (* (car l) x))))))))
12      (P lon k))))))

```

As is customary with CPS, the `product-cps` function accepts an additional parameter `k` representing the continuation (line 2). Here, however, we program two continuations: the normal continuation that grows and computes a series of multiplications once the base case is reached (lines 5, 7, and 11), and another continuation to break out to the main program if a zero is encountered in the input list (line 3). The original, pristine continuation passed into `product-cps`—the identity function—is the continuation that returns the return value of `product-cps` to the main program. On line 3, we bind that continuation with the identifier `break`. Thus, on line 3, we have two continuations: the normal continuation `k` and the exception continuation `break`—though on line 3 they are both the identity function. Next, we define a nested, recursive function `P`, also using CPS, that accepts a list and a continuation—this time the continuation is called `growing-k` (line 5). The `growing-k` continuation grows with each recursive call made (line 11) until the base case is reached (line 7). If a zero is encountered (line 8), the continuation `break` is followed with a string representing an error message. We pass the string "Encountered a zero in the input list." to `break`, rather than 0, to reinforce that the `break` continuation is followed. In other words, the current continuation is replaced with the `break` continuation:

```

> (product-cps '(1 2 3 4 5) (lambda (x) x))
120

> (product-cps '(1 2 0 4 5) (lambda (x) x))
"Encountered a zero in the input list."
>

```

Neither the definition of `product-cps` nor the definition of `product` using `call/cc` in Section 13.3.1 performs any multiplications if the input list contains a zero. However, if the input list does not include any zeros, neither version is space efficient. Even though the CPS version uses tail recursion (line 10), the passed continuation is growing toward the base case. Thus, there is a trade-off between *time complexity* and *space complexity*. If we desire to avoid any multiplications until we determine that the list does not contain a zero, we must build up the steps potentially needed to perform a series of multiplications—the growing passed

continuation—which we will not invoke until we have determined the input list does not include a zero. This approach is time efficient, but not space efficient. In contrast, if we desire the function to run in constant space, we must perform the multiplications as the recursion proceeds (line 10 in the following definition):

```

1 (define product-cps
2   (lambda (lon k)
3     (let ((break k))
4       (letrec ((P
5                 (lambda (l a constant-k)
6                   (cond
7                     ((null? l) (constant-k a))
8                     ((zero? (car l)) (break
9                                     "Encountered a zero in the input list."))
10                    (else (P (cdr l) (* (car l) a) constant-k))))))
11     (P lon 1 k))))

```

Here, the passed continuation `constant-k` remains constant across recursive calls to `P`. Hence, we renamed the passed continuation from `growing-k` to `constant-k`. This approach is space efficient, but not time efficient. Also, because `constant-k` never grows, it remains the same as `break` throughout the recursive calls to `P`. Thus, we can eliminate `break`:

```

(define product-cps
  (lambda (lon k)
    (letrec ((P (lambda (l a k)
                  (cond
                    ((null? l) (k a))
                    ((zero? (car l))
                     (k "Encountered a zero in the input list."))
                    (else (P (cdr l) (* (car l) a) k))))))
      (P lon 1 k))))

```

Table 13.11 summarizes the similarities and differences in these three versions of a product function.

We conclude our discussion of the time-space trade-off by stating:

- We can be *time efficient* by waiting until we know for certain that we will not encounter any exceptions before beginning the necessary computation. This requires us to store the pending computations on the call stack or

Version of product Using	Time Efficient: No Multiplications If a Zero Present	Control Behavior	Space Efficient: Runs in Constant Space
call/cc (second version in Section 13.3.1)	✓	recursive	×
CPS (first version in Section 13.8.4)	✓	iterative	×
CPS (second version in Section 13.8.4)	×	iterative	✓

Table 13.11 Properties Present and Absent in the `call/cc` and CPS Versions of the `product` Function. Notice that we cannot be both time and space efficient.

in a continuation parameter (e.g., the second version of `factorial` in Section 13.8.2 or the first version of `product-cps` in Section 13.8.4.

- Alternatively, we can be *space efficient* by incrementally computing intermediate results (in, for example, an accumulator parameter) when we are uncertain about the prospects of encountering any exceptional situations as we do so. This was the case with the third definition of `factorial` in Section 13.8.2 and the second definition of `product-cps` in Section 13.8.4.

It is challenging to do both (see Table 13.14 in Section 13.12: Chapter Summary).

13.8.5 `call/cc` Vis-à-Vis CPS

The `call/cc` and CPS versions of the `product` function (in Section 13.3.1 and in Section 13.8.4, respectively) are instructive for highlighting differences in the use of `call/cc` and CPS. The CPS versions provide two notable degrees of freedom.

- **The function can accept more than one continuation.** Any function defined using CPS can accept more than one continuation. For instance, we can define `product-cps` as follows, rendering the *normal* and *exceptional* continuations more salient:

```

1  (define product-cps
2    (lambda (lon k break)
3      (letrec ((P
4                 (lambda (l normal-k)
5                   (cond
6                     ((null? l) (normal-k 1))
7                     ((zero? (car l)) (break
8                                     "Encountered a zero in the input list."))
9                     (else (P (cdr l)
10                             (lambda (x)
11                               (normal-k (* (car l) x)))))))
12      (P lon k))))

```

In this version, the second and third parameters (`k` and `break`) represent the normal and exceptional continuations, respectively:

```

1  > (product-cps '(1 2 3 4 5) (lambda (x) x) (lambda (x) x))
2  120
3
4  > (product-cps '(1 2 0 4 5) (lambda (x) x) (lambda (x) x))
5  "Encountered a zero in the input list."
6
7  > (product-cps '(1 2 0 4 5) (lambda (x) x)
8    (lambda (x) (cons "Error message: " (cons x '()))))
9  ("Error message: " "Encountered a zero in the input list.")
10
11 > (product-cps '(1 2 0 4 5) (lambda (x) x) list)
12 ("Encountered a zero in the input list.")

```

In the last invocation to `product-cps` (line 11), `break` is bound to the built-in Scheme `list` function at the time of the call.

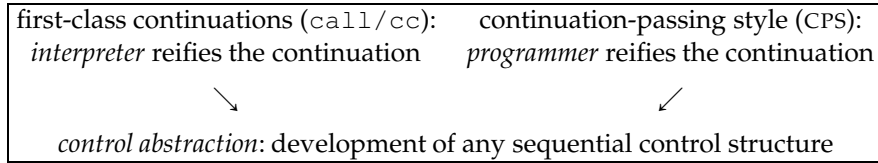


Figure 13.9 Both `call/cc` and CPS involve reification and support control abstraction.

- **Any continuation can accept more than one argument.** Any continuation passed to a function defined using CPS can accept more than one argument because the programmer is defining the function that represents the continuation (rather than the interpreter reifying and returning it as a unary function, as is the case with `call/cc`). The same is not possible with `call/cc`—though it can be simulated (Programming Exercise 13.10.15). For instance, we can replace lines 7–8 in the definition of `product-cps` given in this subsection with

```
((zero? (car l)) (break 0
  "Encountered a zero in the input list."))
```

Now `break` accepts two arguments: the result of the evaluation of the product of the input list (i.e., here, 0) and an error message:

```
> (product-cps '(1 2 0 4 5) (lambda (x) x) list)
(0 "Encountered a zero in the input list.")
```

This approach helps us cleanly factor the code to handle successful execution from that for unsuccessful execution (i.e., the exception).

Figure 13.9 compares `call/cc` and CPS through reification.

13.9 Callbacks

A *callback* is simply a reference to a function, which is typically used to return control flow back to another part of the program. The concept of a callback is related to continuation-passing style. Consider the following Scheme program in direct style:

```
1 > (let* ((dictionnaire '(poire pomme))
2         (addWord (lambda (word)
3                   ;; add word to dictionary
4                   (set! dictionnaire (cons word dictionnaire))))
5       (getDictionnaire (lambda () dictionnaire)))
6   (begin
7     (addWord 'pamplemousse)
8     (getDictionnaire)))
9
10 '(pamplemousse poire pomme)
```

The main program (lines 6–8) calls `addWord` (to add a word to the dictionary; line 7), followed by `getDictionnaire` (to get the dictionary; line 8). The following is the rendering of this program in CPS using a callback:

```

11 > (let* ((dictionnaire '(poire pomme))
12         (addWord (lambda (word callback)
13                     ;; add word to dictionary
14                     (set! dictionnaire (cons word dictionnaire))
15                     (callback))) ; call callback
16         (getDictionnaire (lambda () dictionnaire)))
17   (addWord 'pamplemousse getDictionnaire))
18
19 ' (pamplemousse poire pomme)

```

This expression uses CPS without recursion. The callback (`getDictionnaire`) is the continuation of the computation (of the main program), which is explicitly packaged in an argument and passed on line 17. Then the function that receives the callback as an argument—the *caller* of the callback (`addWord`)—calls it in tail position on line 15. Control flows *back* to the *callback* function.

Assume the two functions called in the main program on lines 7 and 8 run in separate threads; in other words, the call to `getDictionnaire` starts before the call to `addWord` returns. In this scenario, `getDictionnaire` may return `' (poire pomme)` before `addWord` returns. However, the version using a callback does not suffer due to the use of CPS. It is as if the main program says to the `addWord` function: “I need you to add a word to the dictionary so that when I call `getDictionnaire` it will return the most recent dictionary.” The `addWord` function replies: “Sure, but it is going to take quite some time for me to add the word. Are you sure you want to wait?” The main program replies: “No, I don’t. I’ll pass `getDictionnaire` to you and you can call it back yourself when you have what it needs to do its job.”

Callbacks find utility in user interface and web programming, where a callback is stored/registered in user interface components like buttons so that it can be called when a component is engaged (e.g., clicked). The idea is that the callback is an event handler; the main program contains the event loop that listens for events (e.g., a mouse click); and the function that is passed the callback (i.e., the caller) installs/registers the callback as an event handler in the component or with the (operating) system. The following program sketches this approach:

```

1  ;;; this function is defined in the UI toolkit API
2  (define installHandler
3    (lambda (eventhandler)
4      ;; install/register callback function
5    ))

1  ;;; programmer defines this mouse click handler function
2  (define handleClick
3    (lambda ()
4      ;; actions to perform when button is clicked
5      ...))
6
7  (define main
8    (lambda ()

```

```
9      (begin
10      ...
11      (installHandler handleClick) ; install callback function
12      (start-event-loop)))
```

This type of callback is called a *deferred callback* because its execution is deferred until the event that triggers its invocation occurs. Sometimes callbacks used this way are also referred to as *asynchronous callbacks* because the callback (`handleClick`) is invoked *asynchronously* or “at any time” in response to the (mouse click) event.

In an object-oriented context, the UI component is an object and its event handlers are defined as methods in the class of which the UI component is an instance. The methods to install/register custom event handlers (i.e., callback installation methods) are also part of this class. When a programmer desires to install a custom event handler, either (1) the programmer calls the installation method and passes a callback to it, and the installation method stores a pointer to that callback in an instance variable of the object, or (2) the programmer creates a subclass and overrides the default event handler.

Programming with callbacks is an inversion of the traditional programming practice with an API. Typically, an application program calls functions in a language library or API to make use of the abstractions that the API provides as they relate to the application. With callbacks, the API invokes callbacks the programmer defines and installs.

13.10 CPS Transformation

A program written using recursive control behavior can be *mechanically* rewritten in CPS (i.e., iterative control behavior), and that mechanical process is called CPS conversion. The main idea in CPS conversion is to transform the program so that implicit continuations are represented as *closures* manipulated by the program. The informal steps involved in this process are:

1. Add a formal parameter representing the continuation to each `lambda` expression.
2. Pass an anonymous function representing a continuation in each function call.
3. Use the passed continuation in the body of each function definition to return a value.

The CPS conversion involves a set of rewrite rules from a variety of syntactic constructs (e.g., conditional expressions or function applications) to their equivalent forms in continuation-passing style (Feeley 2004). As a result of this systematic conversion, all non-tail calls in the original program are translated into tail calls in the converted program, where the continuation of the non-tail call is packaged and passed as a closure, leaving the call in tail position. Since each function call is in tail position, each function call can be translated as a jump using tail-call optimization (see the right side of Figure 13.7).

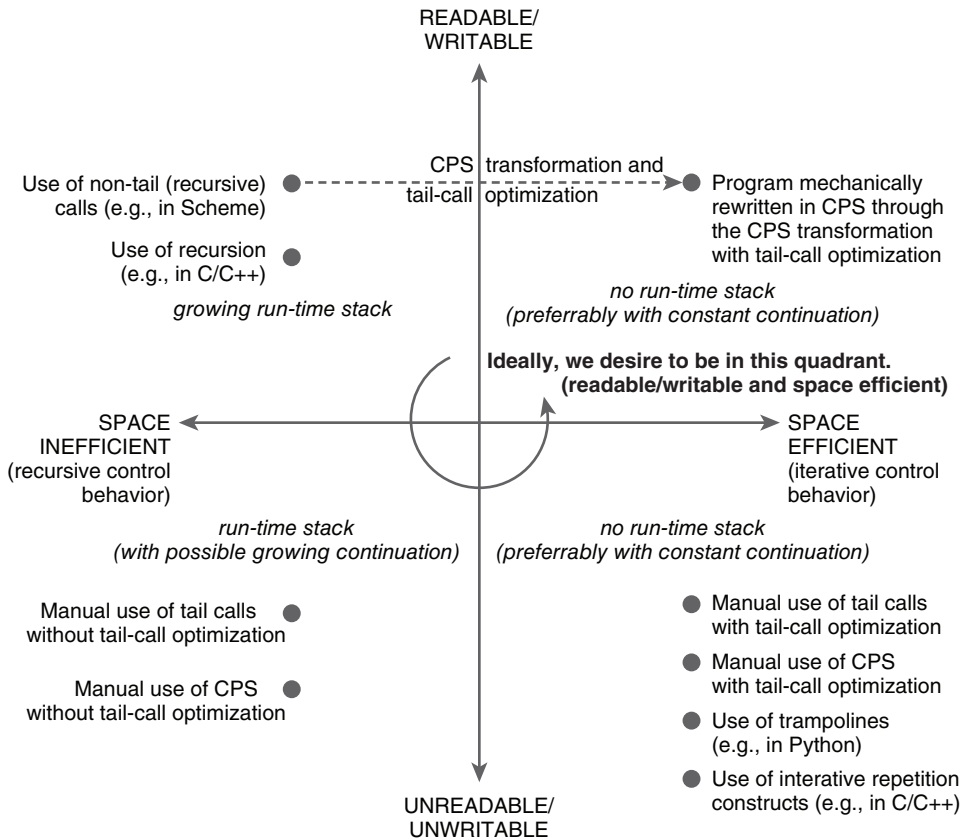


Figure 13.10 Program readability/writability vis-à-vis space complexity axes: (top left) writable and space inefficient: programs using non-tail (recursive) calls; (bottom left) unwritable and space inefficient: programs using tail calls, including CPS, but without tail-call optimization (TCO); (bottom right) unwritable and space efficient: programs using tail calls, including CPS, with TCO, exhibiting iterative control behavior; and (top right) writable and efficient: programs using non-tail (recursive) calls mechanically converted to the use of all tail calls through the CPS transformation, with TCO, exhibiting iterative control behavior. The curved arrow at the origin indicates the order in which these approaches are presented in the text.

Continuation-passing style with tail-call optimization renders recursion as efficient as iteration. Thus, the CPS transformation applied to a program exhibiting recursive control behavior leads to a program that exhibits iterative control behavior and was both originally readable and writable (see the top-right quadrant of Figure 13.10). In other words, the CPS transformation (from recursive control behavior to iterative control behavior) concomitantly supports run-time efficiency and the preservation of the symmetry between the program code for functions and the mathematical definitions of those functions during programming. Table 13.12

Control Behavior	Advantage	Disadvantage
Recursive	function specification and definition reflect each other; readable/writable	unbounded memory space
Iterative	bounded control context; potential to run in constant memory space	function specification and definition do not reflect each other; unreadable/unwritable
Recursive Using CPS Transformation with Tail-Call Optimization	function specification and definition reflect each other; readable/writable; run-time stack unnecessary	

Table 13.12 Advantages and Disadvantages of Functions Exhibiting Recursive Control Behavior, Iterative Control Behavior, and Recursive Control Behavior with CPS Transformation

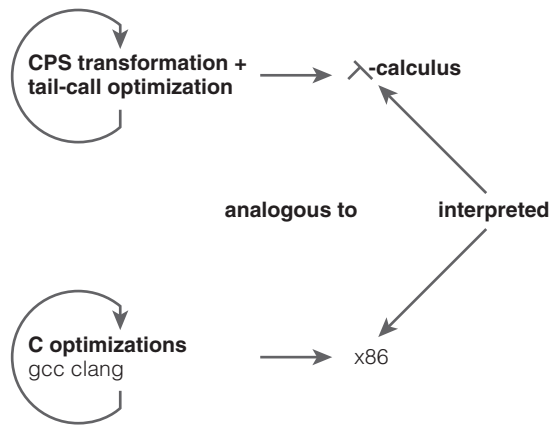


Figure 13.11 CPS transformation and tail-call optimization with subsequent low-level `letrec/let*/let-to-lambda` transformations can be viewed as compilation optimizations akin to those performed by C compilers (e.g., `gcc` or `clang`).

summarizes the advantages and disadvantages of recursive/iterative control behavior and CPS with TCO. The CPS transformation and subsequent tail-call optimization are conceptually analogous to compilation optimizations performed by C compilers such as `gcc` or `clang` (Figure 13.11).

13.10.1 Defining `call/cc` in Continuation-Passing Style

The `call/cc` function can be defined in CPS. Consider the following expression:

```

> (+ 1
   (call/cc
    (lambda (capturedk) (+ 2 (capturedk 3)))))
4

```

Translating this expression into CPS leads to

```
> (call/cc-cps
   (lambda (capturedk normal-k)
     (capturedk 3 (lambda (result) (+ 2 result))))
   (lambda (result) (+ 1 result)))
4
```

All we have done so far is make the implicit continuation waiting for `call/cc` to return [i.e., `(lambda (result) (+ 1 result))`] and the implicit continuation waiting for `capturedk` to return [i.e., `(lambda (result) (+ 2 result))`] explicit. What `call/cc-cps` must do is:

1. Invoke—*call*—its first argument.
2. Pass to it a continuation that ignores any pending computations between the invocation of `call/cc-cps` and the invocation of the captured continuation `capturedk`.

```
1 (define call/cc-cps
2   (lambda (f normal-k)
3
4     (let ((reified-current-continuation
5           ;; replace current continuation with captured continuation:
6           ;; replace current continuation currentk_tobeignored with
7           ;; with captured continuation normal-k of call/cc-cps
8           (lambda (result currentk_tobeignored)
9             (normal-k result))))
10
11     (f reified-current-continuation normal-k))))
```

The expression on line 11 calls the first argument to `call/cc-cps` (i.e., the function `f`; step 1) and passes to it the reified continuation of the invocation of `call/cc-cps` (i.e., `reified-current-continuation`; step 2) created on lines 4–9. When `call/cc-cps` is invoked:

- `f` is

```
(lambda (capturedk normal-k)
  (capturedk 3 (lambda (result) (+ 2 result))))
```
- `normal-k` is `(lambda (result) (+ 1 result))`

The `call/cc-cps` function invokes `f` and passes to it a function—`reified-current-continuation`—that replaces the continuation of `f` [i.e., `(lambda (result) (+ 2 result))`] with the continuation of `call/cc-cps` [i.e., `normal-k = (lambda (result) (+ 1 result))`]. It appears as if the value of the argument `normal-k` passed on line 11 to the function `f` is insignificant because `normal-k` is never used in the body of the `f`. For instance, we can replace line 11 with `(f reified-current-continuation "ignore")` and the expression will still return 4. However, consider another example:

```
1 > (+ 1
2   (call/cc
```



```

3      (let ((f (lambda (x) (+ x 10))))
4          (lambda (capturedk) (+ 2 (f 10)))))
5  23

```

Unlike in the prior example, the continuation captured through `call/cc` is never invoked in this example; that is, the captured continuation `capturedk` is not invoked on line 4. Translating this expression into CPS leads to

```

1  > (call/cc-cps
2      (let ((f (lambda (x normal-k) (normal-k (+ x 10)))))
3          (lambda (capturedk normal-k)
4              (normal-k (f 10 (lambda (result) (+ 2 result))))))
5      (lambda (result) (+ 1 result)))
6  23

```

Again, the continuation `capturedk` passed as the second argument to `call/cc-cps` on line 5 is never invoked in the body (line 4) of the first argument to `call/cc-cps` (lines 2–4). Thus, in this example, the value of the argument `normal-k` passed on line 11 in the definition of `call/cc-cps` to the function `f` is significant because `normal-k` is used in the body of the `f`. If we replace line 11 with `(f reified-current-continuation "ignore"))`, the expression will not return 23. A simplified version of the `call/cc-cps` function is

```

(define call/cc-cps
  (lambda (f k)
    (f (lambda (return_value ignore)
         (k return_value)) k)))

```

Here are two additional examples of invocations of `call/cc-cps`, along with the analogous `call/cc` examples:

```

> ;; invokes the captured continuation break (CPS)
> (call/cc-cps
  (lambda (break normal-k)
    (break 5 (lambda (return_value) (+ return_value 2))))
  (lambda (return_value) (+ return_value 1)))
6

> ;; invokes the captured continuation break (direct style; non-CPS)
> (+ 1
  (call/cc (lambda (break)
             (+ 2 (break 5)))))
6

> ;; does not invoke the captured continuation break (CPS)
> (call/cc-cps
  (lambda (break normal-k)
    (normal-k (+ 2 5))))
  (lambda (return_value) (+ return_value 1)))
8

> ;; does not invoke the captured continuation break
> ;; (direct style; non-CPS)
> (+ 1
  (call/cc (lambda (break)
             (+ 2 5)))))
8

```

Since first-class continuations can be implemented from first principles in Scheme, the `call/cc` function is technically unnecessary. So why is `call/cc` included in Scheme and other languages supporting first-class continuations?

Unfortunately, the procedures resulting from the conversion process are often difficult to understand. The argument that [first-class] continuations need not be added to the Scheme language is factually correct. It has as much validity as the statement that “the names of the formal parameters can be chosen arbitrarily.” And both of these arguments have the same basic flaw: the form in which a statement is written can have a major impact on how easily a person can understand the statement. While understanding that the language does not inherently need any extensions to support programming using [first-class] continuations, the Scheme community nevertheless chose to add one operation [i.e., `call/cc`] to the language to ease the chore. (Miller 1987, p. 209)

Conceptual Exercises for Sections 13.8–13.10

Exercise 13.10.1 Consider the following expression:

```
1 (let ((mult (lambda (x y) (* x y))))
2   (let ((square (lambda (x) (mult x x))))
3     (write (+ (square 10) 1))))
```

(a) Reify the continuation of the invocation `(square 10)` on line 3.

(b) Rewrite this expression using *continuation-passing style*.

Exercise 13.10.2 Reconsider the first definition of `product-cps` given in Section 13.8.4. Show the body of the continuation `growing-k` when it is used on line 7 when `product-cps` is called as `(product-cps ' (1 2 3 4 5) (lambda (x) x))`.

Exercise 13.10.3 Does the following definition of `product` perform any unnecessary multiplications? If so, explain how and why. If not, explain why not.

```
(define product
  (lambda (l)
    (letrec ((P (lambda (lon break)
                  (cond
                    ((null? lon) (break 1))
                    ((zero? (car lon)) (break 0))
                    (else (P (cdr lon)
                           (lambda (returnvalue)
                             (break (* (car lon) returnvalue)))))))
              (P l (lambda (x) x)))))
```

Exercise 13.10.4 Explain what CPS offers that `call/cc` does not.

Exercise 13.10.5 Consider the following Scheme program:

```
(define stackBuilder
  (lambda(x)
    (cond
      ((eqv? 0 x) "DONE" )
      (else (cons '() (stackBuilder (- x 1)))))))

(define stackBuilderCPS
  (lambda(x k)
    (let ((break k))
      (letrec ((helper (lambda (x k)
                          (cond
                            ((eqv? 0 x) (break "DONE") )
                            (else (helper (- x 1)
                                           (lambda (rv) (cons '() rv)))))))
                (helper x k))))))

(define stackBuilder-cc
  (lambda(x)
    (call/cc
      (lambda(k)
        (letrec ((helper (lambda(x)
                            (cond
                              ((eqv? 0 x) (k "DONE"))
                              (else (helper (- x 1)))))))
          (helper x))))))

(stackBuilder 10)
(stackBuilderCPS 10 (lambda(x) x))
(stackBuilder-cc 10)
```

Run this program in the Racket debugger and step through each of the three different calls to `stackBuilder`, `stackBuilderCPS`, and `stackBuilder-cc`. In particular, observe the growth, or lack thereof, of the stack in the upper right-hand corner of the debugger. What do you notice? Report the details of your observations of the behavior and dynamics of the stack.

Exercise 13.10.6 Compare and contrast *first-class continuations* (captured through a facility like `call/cc`) and *continuation-passing style*. What are the advantages and disadvantages of each? Are there situations where one is preferred over the other? Explain.

Programming Exercises for Sections 13.8–13.10

Table 13.13 presents a mapping from the greatest common divisor exercises here to some of the essential aspects of CPS.

Exercise 13.10.7 Rewrite the following Scheme expression in *continuation-passing style*:

```
(let ((f (lambda (x) (* 3 (+ x 1)))))
  (+ (* (f 32) 2) 1))
```

Programming Exercise	Start from	Input		Nonlocal Exit for		Tail Recursion	No Unnecessary Operations Computed	Constant Space Complexity; Static Continuation
		LoN	S-Expression	1 in List	Intermediate gcd = 1			
13.10.17	N/A	✓	×	✓	×	✓	✓	×
13.10.18	13.10.17	✓	×	✓	✓	✓	✓	×
13.10.19	N/A	×	✓	✓	×	✓	✓	×
13.10.20	13.10.19	×	✓	✓	✓	✓	✓	×
13.10.21	N/A	✓	×	✓	✓	✓	×	✓
13.10.22	13.10.21	✓	×	✓	✓	✓	×	✓
13.10.23	N/A	×	✓	✓	×	✓	×	✓
13.10.24	13.10.23	×	✓	✓	✓	✓	×	✓

Table 13.13 Mapping from the Greatest Common Divisor Exercises in This Section to the Essential Aspects of Continuation-Passing Style

Exercise 13.10.8 Rewrite the following Scheme expression in *continuation-passing style*:

```
(let ((g (lambda (x) (+ x 1))))
  (let ((f (lambda (x) (* 3 (g x)))))
    (g (* (f 32) 2))))
```

Exercise 13.10.9 Define a recursive Scheme function `member1` that accepts only an atom `a` and a list of atoms `lat` and returns the integer position of `a` in `lat` (using zero-based indexing) if `a` is a member of `lat` and `#f` otherwise. Your definition of `member1` must use *continuation-passing style* to compute the position of the element, if found, in the list. Your definition must not use `call/cc`. In addition, your definition of `member1` must not return back through all the recursive calls when the element `a` is not found in the list `lat`. Your function must not perform any unnecessary operations, but need not return in constant space. Use the following template for your function and include the missing lines of code (represented as `. . .`):

```
(define member1
  (lambda (a lat)
    (letrec ((member-cps (lambda (ll break)
      (letrec ((M (lambda (l k)
        (cond
          ...
          ...
          ...))))
        ...))))
      (member-cps lat (lambda (x) x)))))
```

See the examples in Programming Exercise 13.3.6.

Exercise 13.10.10 Define a recursive Scheme function `member1` that accepts only an atom `a` and a list of atoms `lat` and returns the integer position of `a` in `lat` (using zero-based indexing) if `a` is a member of `lat` and `#f` otherwise. Your definition of `member1` must use *continuation-passing style*, but the passed continuation must not grow. Thus, the function must run in constant space. Your definition must not use `call/cc`. In addition, your definition of `member1` must not return back through all the recursive calls when the element `a` is not found in the list `lat`. Your function must run in constant space, but need not avoid all unnecessary operations. Use the following template for your function and include the missing lines of code (represented as `. . .`):

```
(define member1
  (lambda (a lat)
    (letrec ((member-cps (lambda (l ... break)
      (cond
        ...
        ...
        ...))))
      (member-cps lat ... (lambda (x) x)))))
```

See the examples in Programming Exercise 13.3.6.

Exercise 13.10.11 Define a Scheme function `fibonacci`, using *continuation-passing style*, that accepts a non-negative integer `n` and returns the `n`th Fibonacci number (whose description is given in Programming Exercise 13.7.6). Your definition of `fibonacci` must run in $O(n)$ and $O(1)$ time and space, respectively. Use the following template for your function and include the missing lines of code (represented as `. . .`):

```
(define fibonacci
  (lambda (n)
    (letrec ((fibonacci-cps (lambda (n prev curr k)
                              (cond
                               . . .
                               . . .))))
      (fibonacci-cps n (lambda (x) x))))))
```

Do not use `call/cc` in your function definition. See the examples in Programming Exercise 13.7.6.

Exercise 13.10.12 Define a Scheme function `int/cps` that performs integer division. The function must accept four parameters: the two integers to divide and success and failure continuations. The failure continuation is followed when the divisor is zero. The success continuation accepts two values—the quotient and remainder—and is used otherwise. Use the built-in Scheme function `quotient`.

Examples:

```
> (int/cps 5 3 list (lambda (x) x))
(1 2)
> (int/cps 5 0 list (lambda (x) x))
"divide by zero"
> (int/cps 6 2 list (lambda (x) x))
(3 0)
```

Exercise 13.10.13 Redefine the first version of the Scheme function `product-cps` in Section 13.8.4 as `product`, a function that accepts a *variable* number of arguments and returns the product of them. Define `product` using *continuation-passing style* such that no multiplications are performed if any of the list elements are zero. Your definition must not use `call/cc`. The nested function `P` from the first version in Section 13.8.4 is named `product-cps` in this revised definition.

Examples:

```
> (product 1 2 3 4 5 6)
720
> (product 1 2 3 0 4 5 6)
"Encountered a zero in the input list."
```

Exercise 13.10.14 Redefine the Scheme function `product-cps` in Section 13.8.4 as `product`, a function that accepts a *variable* number of arguments and returns the product of them. Define `product` using *continuation-passing style*. The function

must run in constant space. The nested function `P` from the version in Section 13.8.4 is named `product-cps` in this revised definition.

Exercise 13.10.15 Consider the following definition of `product-cps`:

```
(define product-cps
  (lambda (lon k break)
    (letrec ((P
              (lambda (l normal-k)
                (cond
                 ((null? l) (normal-k 1))
                 ((zero? (car l)) (break 0
                                     "Encountered a zero in the input list.))
                 (else (P (cdr l)
                          (lambda (x) (normal-k (* (car l) x)))))))
              (P lon k))))))
```

When a zero is encountered in the input list, this function returns with two values: 0 and a string. Recall that the ability to continue with multiple values is an advantage of CPS over `call/cc`.

Redefine this function using *direct style* (i.e., in non-CPS fashion) with `call/cc`. While it is not possible to pass more than one value to a continuation captured with `call/cc`, figure out how to simulate this behavior to achieve the following result when a zero is encountered in the list:

```
> (product-cps '(1 2 0 4 5))
(0 "Encountered a zero in the input list.")
```

Exercise 13.10.16 Define a Scheme function `product` that accepts only a list of numbers and returns the product of them. Your definition must not perform any multiplications if any of the list elements is zero. Your definition must not use `call/cc` or *continuation-passing style*. Moreover, the call stack may grow only *once* to the length of the list plus one (for the original function).

Exercise 13.10.17 Define a Scheme function `gcd-lon` using *continuation-passing style*. The function accepts only a non-empty list of *positive, non-zero* integers, and contains a nested function `gcd-lon1` that accepts only a non-empty list of *positive, non-zero* integers and a continuation (in that order) and returns the greatest common divisor of the integers. During computation of the greatest common divisor, if a 1 is encountered in the list, return the string "1: encountered a 1 in the list" immediately without *ever* calling `gcd-cps` and before performing any arithmetic computations. Use only *tail recursion*. Use the following template for your function and include the missing lines of code (represented as ...):

```
(define gcd-lon
  (lambda (lon)
    (let ((main (lambda (ll break)
                  (letrec ((gcd-lon1
                          (letrec ((gcd-cps (lambda (u v k)
                                              (cond
```

```

      ((zero? v) (k u))
      (else (gcd-cps v (remainder u v) k))))))
(lambda (l k)
  (cond
    (...)
    (...)
    (else ...))))))
(gcd-lon1 ll break))))
(main lon (lambda (x) x))))

```

Do not use `call/cc` in your function definition.

Examples:

```

> (gcd-lon '(20 48 32 1))
"1: encountered a 1 in the list"
> (gcd-lon '(4 32 12 8 16))
4
> (gcd-lon '(4 32 1 12 8 16))
"1: encountered a 1 in the list"
> (gcd-lon '(4 8 11 11))
1

```

For additional examples, see the examples in Programming Exercise 13.3.13.

Exercise 13.10.18 Modify the solution to Programming Exercise 13.10.17 so that if a 1 is ever computed as the result of an intermediate call to `gcd-cps`, the string "1: computed an intermediary gcd = 1" is returned immediately before performing any additional arithmetic computations. Use the function template given in Programming Exercise 13.10.17.

Examples:

```

> (gcd-lon '(20 48 32 1))
"1: encountered a 1 in the list"
> (gcd-lon '(4 32 12 8 16))
4
> (gcd-lon '(4 32 1 12 8 16))
"1: encountered a 1 in the list"
> (gcd-lon '(4 8 11 11))
"1: computed an intermediary gcd = 1"

```

For additional examples, see the examples in Programming Exercise 13.3.14.

Exercise 13.10.19 Define a Scheme function `gcd*` using *continuation-passing style*. The function accepts only a non-empty *S-expression* of *positive, non-zero* integers that contains no empty lists, and contains a nested function `gcd*1` that accepts only a non-empty list of *positive, non-zero* integers and a continuation (in that order) and returns the greatest common divisor of the integers. During computation of the greatest common divisor, if a 1 is encountered in the list, return the string "1: encountered a 1 in the S-expression" immediately without *ever* calling `gcd-cps` and before performing any arithmetic computations. Use only *tail recursion*. Use the following template for your function and include the missing lines of code (represented as `...`):


```

(define gcd*
  (lambda (l)
    (let ((main (lambda (ll break)
      (letrec ((gcd*1
        (lambda (gcd-cps (lambda (u v k)
          (cond
            ((zero? v) (k u))
            (else (gcd-cps v (remainder u v) k))))))
        (lambda (l k)
          (cond
            (...
              (cond
                (...
                  (else ...))
                (...
                  (else ...))))))
          (gcd*1 ll break))))))
    (main l (lambda (x) x))))

```

Examples:

```

> (gcd* '((36 12 48) (((24 36) 6 54 240))))
6
> (gcd* '((((20)))) 48 (32) 1))
"1: encountered a 1 in the S-expression"
> (gcd* '((4 (32 12) 8) ((16))))
4
> (gcd* '((4 32 1) (12 (8) 16)))
"1: encountered a 1 in the S-expression"
> (gcd* '(4 8 (((11 11))))))
1
> (gcd* '(((4 8)) (11)))
1

```

For additional examples, see the examples in Programming Exercise 13.3.15.

Exercise 13.10.20 Modify the solution to Programming Exercise 13.10.19 so that if a 1 is ever computed as the result of an intermediate call to `gcd-cps`, the string "1: computed an intermediary gcd = 1" is returned immediately before performing any additional arithmetic computations. Use the function template given in Programming Exercise 13.10.19.

Examples:

```

> (gcd* '((36 12 48) (((24 36) 6 54 240))))
6
> (gcd* '((((20)))) 48 (32) 1))
"1: encountered a 1 in the S-expression"
> (gcd* '((4 (32 12) 8) ((16))))
4
> (gcd* '((4 32 1) (12 (8) 16)))
"1: encountered a 1 in the S-expression"
> (gcd* '(4 8 (((11 11))))))
"1: computed an intermediary gcd = 1"
> (gcd* '(((4 8)) (11)))
"1: computed an intermediary gcd = 1"

```

For additional examples, see the examples in Programming Exercise 13.3.16.

Exercise 13.10.21 Define a Scheme function `gcd-lon` using *continuation-passing style*. The function accepts only a non-empty list of *positive, non-zero* integers, and contains a nested function `gcd-lon1` that accepts only a non-empty list of *positive, non-zero* integers, an accumulator, and a continuation (in that order) and returns the greatest common divisor of the integers. During computation of the greatest common divisor, if a 1 is encountered in the list, return the string "1: encountered a 1 in the list" immediately. Use only *tail recursion*. Your continuation parameter must not grow and your function must run in constant space. Use the following template for your function and include the missing lines of code (represented as `...`):

```
(define gcd-lon
  (lambda (lon)
    (let ((main (lambda (ll break)
                  (letrec ((gcd-lon1
                           (letrec ((gcd-cps (lambda (u v k)
                                             (cond
                                                ((zero? v) (k u))
                                                (else (gcd-cps v (remainder u v) k))))))
                          (lambda (l a k)
                            (cond
                             (...)
                             (...)
                             (...)
                             (else ...))))))
                        (gcd-lon1 ll ... break))))))
      (main lon (lambda (x) x))))
```

Do not use `call/cc` in your function definition. See the examples in Programming Exercise 13.3.13.

Exercise 13.10.22 Modify the solution to Programming Exercise 13.10.21 so that if a 1 is ever computed as the result of an intermediate call to `gcd-cps`, the string "1: computed an intermediary gcd = 1" is returned immediately. Use the following template for your function and include the missing lines of code (represented as `...`):

```
(define gcd-lon
  (lambda (lon)
    (let ((main (lambda (ll break)
                  (letrec ((gcd-lon1
                           (letrec ((gcd-cps (lambda (u v k)
                                             (cond
                                                ((zero? v) (k u))
                                                (else (gcd-cps v (remainder u v) k))))))
                          (lambda (l a k)
                            (cond
                             (...)
                             (...)
                             (...)
                             (...)
                             (else ...))))))
                        (gcd-lon1 ll ... break))))))
      (main lon (lambda (x) x))))
```

See the examples in Programming Exercise 13.3.14.

Exercise 13.10.23 Define a Scheme function `gcd*` using *continuation-passing style*. The function accepts only a non-empty *S-expression* of *positive, non-zero* integers that contains no empty lists, and contains a nested function `gcd*1` that accepts only a non-empty list of *positive, non-zero* integers, an accumulator, and a continuation (in that order) and returns the greatest common divisor of the integers. During computation of the greatest common divisor, if a 1 is encountered in the list, return the string "1: encountered a 1 in the S-expression" immediately. Use only *tail recursion*. Your continuation parameter must not grow and your function must run in constant space. Use the following template for your function and include the missing lines of code (represented as `. . .`):

```
(define gcd*
  (lambda (l)
    (let ((main (lambda (ll break)
                  (letrec ((gcd*1
                           (letrec ((gcd-cps (lambda (u v k)
                                             (cond
                                                ((zero? v) (k u))
                                                (else (gcd-cps v (remainder u v) k))))))
                          (lambda (l a k)
                            (cond
                             ((number? (car l))
                              (cond
                               (...)
                               (...)
                               (else ...)))
                             (...)
                             (else ...))))))
                    (gcd*1 ll ... break))))))
    (main l (lambda (x) x))))
```

See the examples in Programming Exercise 13.3.15.

Exercise 13.10.24 Modify the solution to Programming Exercise 13.10.23 so that if a 1 is ever computed as the result of an intermediate call to `gcd-cps`, the string "1: computed an intermediary gcd = 1" is returned immediately. Use the following template for your function and include the missing lines of code (represented as `. . .`):

```
(define gcd*
  (lambda (l)
    (let ((main (lambda (ll break)
                  (letrec ((gcd*1
                           (letrec ((gcd-cps (lambda (u v k)
                                             (cond
                                                ((zero? v) (k u))
                                                (else (gcd-cps v (remainder u v) k))))))
                          (lambda (l a k)
                            (cond
                             (...)
                             (else ...))))))
                    (gcd*1 ll ... break))))))
    (main l (lambda (x) x))))
```

```

                (else ...))))))
      (gcd*1 11 ... break))))
    (main 1 (lambda (x) x))))

```

See the examples in Programming Exercise 13.3.16.

Exercise 13.10.25 Use *continuation-passing style* to define a while loop in Scheme *without* recursion (e.g., `letrec`). Specifically, define a Scheme function `while-loop` that accepts a condition and a body—both as Scheme expressions—and implements a while loop. Define `while-loop` using *continuation-passing style*. Your definition must not use either recursion or `call/cc`. Use the following template for your function and include the missing lines of code (represented as `...`):

```

(define while-loop
  (lambda (condition body)
    (let ((W (lambda (k)
                ...)))
      (W ...))))

```

See the example in Programming Exercise 13.6.6.

Exercise 13.10.26 Define a Scheme function `cps-primitive-transformer` that accepts a Scheme primitive (e.g., `+` or `*`) as an argument and returns a version of that primitive in *continuation-passing style*. For example:

```

> (define *cps (cps-primitive-transformer *))
> (define +cps (cps-primitive-transformer +))

> (+cps 1 2 (lambda (returnvalue)
              (*cps 3 returnvalue (lambda (x) x))))
9

```

Exercise 13.10.27 Consider the Scheme program in Section 13.6.1 that represents an implementation of coroutines using `call/cc`. Rewrite this program using the `call/cc-cps` function defined in Section 13.10.1 as a replacement of `call/cc`.

13.11 Thematic Takeaways

- *First-class continuations* are ideal for programming *abnormal flows of control* (e.g., nonlocal exits) and, more generally, for *control abstraction*—implementing user-defined control abstractions.
- The `call/cc` function captures the current continuation with a representation of the environment, including the run-time stack, at the time `call/cc` is invoked.
- Unlike `goto`, continuation replacement in Scheme [i.e., `(k v)`] is not just a transfer of control, but also a restoration of the environment, including the run-time stack, at the time the continuation was captured.

- *First-class continuations* are sufficient to create a variety of control abstractions, including any desired sequential control structure (Haynes, Friedman, and Wand 1986, p. 143).
- It is the unlimited extent of closures that unleashes the power of first-class continuations for control abstraction. The unlimited lifetime of closures enables control to be transferred to stack frames—called *heap-allocated stack frames*—that seemingly no longer exist.
- A limited extent of closures puts a limit on the scope of control abstraction possible through application of operators for transfer of control (e.g., `set jmp/long jmp` in C) and restricts their use for handling exceptions to, for example, nonlocal exits.
- Using *first-class continuations* to create new control structures and abstractions is an art requiring creativity.
- Use of *tail recursion* trades off function *writability* for improved *space complexity*.
- The `call/cc` function automatically reifies the implicit continuation that the programmer of a function using CPS manually reifies.
- In a program written in *continuation-passing style*, the continuation of every function call is passed as an additional argument representing the continuation of the call. In consequence, every function call is in *tail position*.
- In *continuation-passing style*, the continuation passed to the function defined using CPS must both exclusively use tail calls and be invoked in tail position itself.
- Continuation-passing style implies tail calls, but tail calls do not imply continuation-passing style.
- Tail-call optimization eliminates the run-time stack, thereby enabling (recursive) functions to run in constant space—and rendering recursion as efficient as iteration.
- A stack is unnecessary for a language to support functions.
- Tail-call optimization is applicable to all tail calls, not just tail-recursive ones.
- There is a trade-off between *time complexity* and *space complexity* in programming (Table 13.14).

13.12 Chapter Summary

This chapter is concerned with imparting *control* to a computer program. We used *first-class continuations* (captured, for example, through `call/cc`), *tail recursion*, and *continuation-passing style* to tease out ideas about control and how to affect it in programming. While evaluating an expression, the interpreter must keep track of what to do with the return value of the expression it is currently evaluating. The actions entailed in the “what to do with the return value” are the pending computations or the *continuation* of the computation. A *continuation* is a one-argument function that represents the remainder of a computation from a given point in a program. The argument passed to a continuation is the return value

Approach	To Program Abnormal Flow of Control	To Program Normal Flow of Control	Time Efficient	Control Behavior	Constant Continuation Parameter	Space Efficient: Runs in Constant Space	Example(s)
call/cc without tail recursion	use captured k	use call stack	✓	recursive	N/A	×	second version of product in Section 13.3.1
call/cc with tail recursion	use captured k	use accumulator parameter	×	iterative	N/A	✓	second version of product in Section 13.7.2
tail recursion without CPS or call/cc	must return through call stack	use accumulator parameter	×	iterative	N/A	✓	first version of product in Section 13.7.2
CPS (implies tail call)	use passed break, e.g., identity	use passed growing-k	✓	iterative	×	×	first version of product in Section 13.8.4
CPS (implies tail call)	use passed break, e.g., identity	use constant continuation and accumulator parameter; e.g., (constant-k a)	×	iterative	✓	✓	second version of product in Section 13.8.4

Table 13.14 The Approaches to Function Definition as Related to Control Presented in This Chapter Based on the Presence and Absence of a Variety of Desired Properties. Theme: We cannot be *both* time and space efficient.

of the prior computation—the one value for which the continuation is waiting to complete the next computation.

The `call/cc` function in Scheme captures the current continuation *with a representation of the environment, including the run-time stack*, at the time `call/cc` is invoked. The expression `(k v)`, where `k` is a first-class continuation captured through `(call/cc (lambda (k) ...))` and `v` is a value, does not just transfer program control. The expression `(k v)` transfers program control **and restores the environment, including the stack, that was active at the time `call/cc` captured `k`, even if it is not active when `k` is invoked**. This capture and restoration of the call stack is the ingredient necessary for supporting the creation of a wide variety of new control constructs. More specifically, it is the unlimited extent of lexical closures that unleash the power of first-class continuations for control abstraction: The unlimited lifetime of closures enable control to be transferred to stack frames that seemingly no longer exist, called *heap-allocated stack frames*.

Mechanisms for transferring control in programming languages are typically used for handling exceptions in programming. These mechanisms include function calls, stack unwinding/crawling operators, exception-handling systems, and first-class continuations. In the absence of heap-allocated stack frames, once the stack frames between the function that caused/raised an exception and the function handling that exception have been popped off the stack, they are gone forever. For instance, the `setjmp/longjmp` stack unwinding/crawling functions in C allow a programmer to perform a nonlocal exit from several functions on the stack in a single jump. Without heap-allocated stack frames, these stack unwinding/crawling operators transfer control down the stack, but not back up it. Thus, these mechanisms are simply for nonlocal exits and, unlike first-class continuations, are limited in their support for implementing other types of control structures (e.g., breakpoints).

We have also defined recursive functions in a manner that maintains the natural correspondence between the recursive specification or mathematical definition of the function [e.g., $n! = n * (n - 1)!$] and the program code implementing the function (e.g., `factorial`). This congruence is a main theme running throughout Chapter 5. When such a function runs, the activation records for all of the recursive calls are pushed onto the run-time stack while building up pending computations. Such functions typically require an ever-increasing amount of memory and exhibit *recursive control behavior*. When the base case is reached, the computation required to compute the function is performed as these pending computations are executed while the activation records for the recursive calls pop off the stack and the memory is reclaimed. In a function using *tail recursion*, the recursive call is the last operation that the function must perform. Such a recursive call is in *tail position* [e.g., `(factorial (- n 1) (* n a))`] in contrast to *operand position* [e.g., `(* n (factorial (- n 1)))`]. A function call is a *tail call* if there is no promise to do anything with the returned value. Recursive functions using tail recursion exhibit *iterative control behavior*. However, the structure of the program code implementing a function using tail recursion no longer reflects the recursive specification of the function—the symmetry is broken.

Thus, the use of *tail recursion* trades off function *writability* for improved *space complexity*.

We can turn all function calls into tail calls by encapsulating any computation remaining after each call—the “what to do next”—into an explicit, reified continuation and passing that continuation as an extra argument in each tail call. In other words, we can make the *implicit* continuation of each called function *explicit* by packaging it as an additional argument passed in each function call. Functions written in this manner use *continuation-passing style* (CPS). The continuation that the programmer of a function using CPS manually reifies is the continuation that the `call/cc` function automatically reifies. A function defined using CPS can accept *multiple* continuations; this property helps us cleanly factor the various ways a program might complete its computation. A function defined in CPS can pass *multiple* results to its continuation; this property provides us with flexibility in communicating results to continuations.

A desired result of CPS is that the recursive function defined in CPS run in constant memory space. This means that no computations are waiting for the return value of each recursive call, which in turn means the function that made the recursive call can be popped off the stack. The growing stack of pending computations can be transmuted through CPS as a growing continuation parameter. We desire a function embracing the spirit of CPS, where, ideally, the passed continuation is not growing. Continuation-passing style with a bounded continuation parameter and *tail-call optimization* eliminates the run-time stack, thereby ensuring the recursive function can run in constant space—and rendering recursion as efficient as iteration.

There is a trade-off between *time complexity* and *space complexity* in programming. We can be either (1) *time efficient*, by waiting until we know for certain that we will not encounter any exceptions before beginning the necessary computation (which requires us to store the pending computations on the call stack or in a continuation parameter), or (2) *space efficient*, by incrementally computing intermediate results (in, for example, an accumulator parameter) in the presence of the uncertainty of encountering any exceptional situations. It is challenging to do both (Table 13.14).

Programming abnormal flows of control and *running recursive functions in constant space* are two issues that can easily get conflated in the study of program control. Continuation-passing style with tail-call optimization can be used to achieve both. Tail-call optimization realizes the constant space complexity. Passing and invoking the continuation parameter (e.g., the identity function) is used to program abnormal flows of control. If the continuation parameter is growing, then it is used to program the normal flow of control—albeit in a cluttered manner. In contrast, `call/cc` is primarily used for programming abnormal flows of control. For instance, the `call/cc` function can be used to unwind the stack in the case of exceptional values (e.g., a 0 in the list input to a product function; see the versions of `product` using `call/cc` in Sections 13.3.1 and 13.7.2). (*Programming abnormal flows of control* with first-class continuations in this manner can be easily confused with *improving time complexity* of a function by obviating the need to return through

Technique	Purpose/Effect
continuation-passing style	tail recursion
tail recursion + TCO	space efficiency
CPS + TCO	space efficiency
first-class continuations (<code>call/cc</code> or CPS) for exception handling	run-time efficiency

Table 13.15 Effects of the Techniques Discussed in This Chapter

layers of pending computations on the stack in the case of a non-tail-recursive function.) Unlike with CPS, the continuation captured through `call/cc` is neither necessary nor helpful for programming normal control flow: If the function uses a tail call, it is already capable of being run in constant space; if the function is not tail recursive, then it must not run in constant space because the stack is truly needed to perform the computation of the function. In that case, the normal flow of control in the recursive call remains uncluttered—unlike in CPS. Table 13.15 summarizes the effects of these control techniques. Table 13.14 classifies some of the example functions presented in this chapter based on factors involved in these trade-offs.

The CPS transformation and subsequent tail-call optimization applied to a program exhibiting recursive control behavior leads to a program exhibiting iterative control behavior that was both originally readable and writable (see the top-right quadrant of Figure 13.10). In other words, the CPS transformation (from recursive control behavior to iterative control behavior) maintains the natural reflection of the program code with the mathematical definition of the function.

First-class continuations, tail recursion, CPS, and tail-call optimization bring us more fully into the third layer of functional programming: *More Efficient and Abstract Functional Programming* (shown in Figure 5.10).

13.13 Notes and Further Reading

An efficient implementation of first-class continuations in Scheme is given in Hieb, Dybvig, and Bruggeman (1990). The language specification of Scheme requires implementations to implement tail-call optimization (Sperber et al. 2010). For an overview of control abstractions in programming languages, especially as related to user-interface software and the implementation of human–computer dialogs, we refer the reader to Pérez-Quinones (1996, Chapter 4). For more information about the CPS transformation, we refer the reader to Feeley (2004), Friedman, Wand, and Haynes (2001, Chapter 8), and Friedman and Wand (2008, Chapter 6). The term *coroutine* was first used by Melvin E. Conway (1963).

Chapter 14

Logic Programming

- (1) No ducks waltz;
- (2) No officers ever decline to waltz;
- (3) All my poultry are ducks.¹

- (1) Every one who is sane can do Logic;
- (2) No lunatics are fit to serve on a jury;
- (3) None of your sons can do Logic.²

(sets of Concrete Propositions, proposed as Premisses for Sorites. Conclusions to be found—in footnotes)

— Lewis Carroll, *Symbolic Logic, Part I: Elementary* (1896)

The more I think about language, the more it amazes me that people ever understand each other at all.

— Kurt Gödel

For now, what is important is not finding the answer, but looking for it.

— Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid* (1979)

IN contrast to an imperative style of programming, where the programmer specifies *how* to compute a solution to a problem, in a declarative style of programming, the programmer specifies *what* they want to compute, and the system uses a built-in search strategy to compute a solution. A simple and perhaps familiar example of declarative programming is the use of an embedded regular expression language within a programming language. For instance, when a programmer writes the Python expression `([a-z])([a-z])[a-z]\2\1`, the

1. My poultry are not officers.

2. None of your sons are fit to serve on a jury.

programmer is declaring what they want to match—in this case, strings consisting of five lowercase alphabetical character palindromes—and not how to match those strings using `for` loops and string manipulation functions. In this chapter, we study the foundation of declarative programming³ in symbolic logic and Prolog—the most classical and widely studied programming language supporting a logic/declarative style of programming.

14.1 Chapter Objectives

- Establish an elementary understanding of *predicate calculus* and *resolution*.
- Discuss *logic/declarative programming*.
- Explore programming in Prolog.
- Explore programming in CLIPS.

14.2 Propositional Calculus

A background in symbolic logic is essential to understanding how logic programs are constructed and executed. Symbolic logic is a formal system involving both a syntax by which propositions and relationships between propositions are expressed and formal methods by which new propositions can be deduced from axioms (i.e., propositions asserted to be true). The goal of symbolic logic is to provide a formal apparatus by which the validity of these new propositions can be verified. Multiple systems of symbolic logic exist, which offer varying degrees of expressivity in describing and manipulating propositions. A *proposition* is a statement that is either true or false (e.g., “Pascal is a philosopher”). Propositional logic involves the use of symbols (e.g., p , q , and r) for expressing propositions. The simplest form of a proposition is an *atomic proposition*. For example, the symbol p could represent the atomic proposition “Pascal is a philosopher.” *Compound propositions* can be formed by connecting two or more atomic propositions with *logical operators* (Table 14.1):

$$\begin{aligned} &\neg p \vee q \\ &p \vee q \supset r \\ &p \vee \neg q \supset r \end{aligned}$$

The precedence of these operators is implied in their top-down presentation (i.e., highest to lowest) in Table 14.1:

$$\begin{aligned} (\neg p \vee q) &\equiv ((\neg p) \vee q) \\ (p \vee q \supset r) &\equiv ((p \vee q) \supset r) \\ (p \vee \neg q \supset r) &\equiv ((p \vee (\neg q)) \supset r) \end{aligned}$$

3. We use the terms *logic programming* and *declarative programming* interchangeably in this chapter.

Logical Concept	Symbol	Example	Semantics
Negation	$\neg$	$\neg p$	<i>not p</i>
Conjunction	$\wedge$	$p \wedge q$	<i>p and q</i>
Disjunction	$\vee$	$p \vee q$	<i>p or q</i>
Implication	$\supset$	$p \supset q$	<i>p implies q</i>
Implication	$\subset$	$p \subset q$	<i>q implies p</i>
Biconditional	$\Longleftrightarrow$	$p \Longleftrightarrow q$	<i>p if and only if q</i>
Entailment (or <i>semantic consequence</i>)	$\models$	$\alpha \models \beta$	(read left-to-right) α entails β (read right-to-left) β follows from α (read right-to-left) β is a <i>semantic consequence</i> of α
Logical Equivalence	$\equiv$	$\alpha \equiv \beta$	α is logically equivalent to β

Table 14.1 Logical Concepts and Operators or Connectors

p	$\neg p$	q	$p \supset q$	$\neg p \vee q$	$(p \supset q) \Longleftrightarrow (\neg p \vee q)$
T	F	T	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	T	F	T	T	T

Table 14.2 Truth Table Proof of the *Logical Equivalence* $p \supset q \equiv \neg p \vee q$

The truth table presented in Table 14.2 proves the logical equivalence between $p \supset q$ and $\neg p \vee q$.

A *model* of a proposition in formal logic is a row of the truth table. *Entailment*, which is a semantic concept in formal logic, means that all of the models that make the left-hand side of the entailment symbol ($\models$) true also make the right-hand side true. For instance, $p \wedge q \models p \vee q$, which reads left to right “ $p \wedge q$ entails $p \vee q$ ” and reads right to left “ $p \vee q$ is a *semantic consequence* of $p \wedge q$.” Notice that $p \vee q \models p \wedge q$ is not true because some models that make the proposition on the left-hand side true (e.g., the second and third rows of the truth table) do not make the proposition on the right-hand side true.

While implication and entailment are different concepts, they are easily confused. *Implication* is a function or connective operator that establishes a conditional relationship between two propositions. *Entailment* is a semantic relation that establishes a consequence relationship between a set of propositions and a proposition.

Implication: $\phi \supset \psi$ is true if and only if $\neg \phi \vee \psi$ is true.

Entailment: $\Gamma \models \psi$ is true if and only if every model that makes all $\phi \in \Gamma$ true, makes ψ true.

p	q	$p \wedge q$	$p \vee q$	$(p \wedge q) \supset (p \vee q)$
T	T	T	T	T
T	F	F	T	T
F	T	F	T	T
F	F	F	F	T

Table 14.3 Truth Table Illustration of the Concept of *Entailment* in $p \wedge q \models p \vee q$

While different concepts, implication and entailment are related:

$\alpha \models \beta$ if and only if the proposition $\alpha \supset \beta$ is true for all models.⁴

This statement is called the *deduction theorem* and a proposition that is true for all models is called a *tautology* (see rightmost column in Table 14.3).

The relationship between logical equivalence ($\equiv$) and entailment ($\models$) is also notable:

$\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$

Biconditional and logical equivalence are also sometimes confused with each other. Like implication, *biconditional* establishes a (bi)conditional relationship between two propositions. Akin to entailment, *logical equivalence* is a semantic relation that establishes a (bi)consequence relationship. While different concepts, biconditional and logical equivalence (like implication and entailment) are similarly related:

$\alpha \equiv \beta$ if and only if the proposition $\alpha \iff \beta$ is true for all models.⁵

The rightmost column in Table 14.2 illustrates that $(p \supset q) \iff (\neg p \vee q)$ is a tautology since $(p \supset q) \equiv (\neg p \vee q)$.

14.3 First-Order Predicate Calculus

Logic programming is based on a system of symbolic logic called *first-order predicate calculus*,⁶ which is a formal system of symbolic logic that uses variables, predicates, quantifiers, and logical connectives to produce propositions involving terms. Predicate calculus is the foundation for logic programming as λ -calculus is the basis for functional programming (Figure 14.1). We refer to first-order predicate calculus simply as predicate calculus in this chapter. The crux of logic programming is that the programmer specifies a knowledge base of known

4. This statement can also be expressed as: $\models (\alpha \supset \beta)$ if and only if $(\alpha \models \beta)$.

5. This statement can also be expressed as: $\models (\alpha \iff \beta)$ if and only if $(\alpha \equiv \beta)$.

6. The qualifier *first-order* implies that in this system of logic, there is no means by which to reason about the predicates themselves.

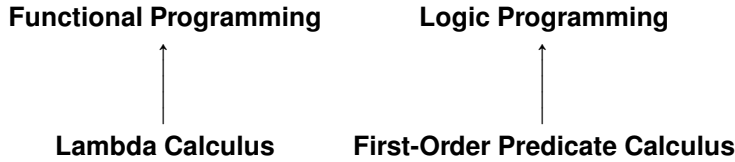


Figure 14.1 The theoretical foundations of *functional* and *logic programming* are λ -calculus and *first-order predicate calculus*, respectively.

propositions—axioms declared to be true—from which the system infers new propositions using a deductive apparatus:

representing the relevant knowledge ← predicate calculus
 ← resolution
 method for inference

14.3.1 Representing Knowledge as Predicates

In predicate calculus, propositions are represented in a formal mathematical manner as predicates. A *predicate* is a function that evaluates to true or false based on the values of the variables in it. For instance, *Philosopher(Pascal)* is a predicate, where *Philosopher* is the predicate symbol or functor and *Pascal* is the argument. Predicates can be used to represent knowledge that cannot be reasonably represented in propositional calculus. The following are examples of atomic propositions in predicate calculus:

Philosopher(Pascal).
Friend(Lucia, Leisel).

In the first example, *Philosopher* is called the *functor*. In the second example, *Lucia, Leisel* is the ordered list of *arguments*. When the functor and the ordered list of arguments are written together in the form of a function as one element of a relation, the result is called a *compound term*. The following are examples of compound propositions in predicate calculus:

$weather(raining) \vee weather(sunny) \supset carry(umbrella)$
 $weather(raining) \vee \neg weather(cloudy) \supset carry(umbrella) \equiv$
 $(weather(raining) \vee (\neg weather(cloudy))) \supset carry(umbrella)$
 $weather(raining) \supset carry(umbrella) \equiv \neg weather(raining) \vee carry(umbrella)$

The *universal* and *existential* logical *quantifiers*, $\forall$ and $\exists$, respectively, introduce *variables* into propositions (Table 14.4):

$\forall X. (presidentOfUSA(X) \supset atLeast35yearsOld(X))$
 (All presidents of the United States are at least 35 years old.)

Quantifier	Example	Semantics
Universal	$\forall X.P$	For all X , P is true.
Existential	$\exists X.P$	There exists a value of X such that P is true.

Table 14.4 Quantifiers in Predicate Calculus

$\exists X.(\text{country}(X) \wedge \text{continent}(X))$
 (There exists a country that is also a continent.)

$\exists X.(\text{drinks}(X, \text{earl grey}) \wedge \neg \text{english}(X))$
 (There exists a non-English person who drinks Earl Grey tea.)

These two logical quantifiers have the highest precedence in predicate calculus. The scope of a quantifier is limited to the atomic proposition that it precedes unless it precedes a parenthesized compound proposition, in which case it applies to the entire compound proposition.

Propositions are purely syntactic and, therefore, have no intrinsic semantics—they can mean whatever you want them to mean. In *Symbolic Logic and the Game of Logic*, Lewis Carroll wrote:

I maintain that any writer of a book is fully authorised in attaching any meaning he likes to a word or phrase he intends to use. If I find an author saying, at the beginning of his book, “Let it be understood that by the word ‘black’ I shall always mean ‘white,’ and by the word ‘white’ I shall always mean ‘black,’” I meekly accept his ruling, however injudicious I think it.

14.3.2 Conjunctive Normal Form

A proposition can be stated in multiple ways. While this redundancy is acceptable for pure symbolic logic, it poses a problem if we are to implement predicate calculus in a computer system. To simplify the process by which new propositions are deduced from known propositions, we use a standard syntactic representation for a set of well-formed formulas (*wffs*). To do so we must convert each individual wff in the set of wffs into *conjunctive normal form* (CNF), which is a representation for a proposition as a flat conjunction of disjunctions:

$$\overbrace{(t_1 \vee t_2 \vee t_3)}^{\text{a clause}} \wedge (t_4 \vee \underbrace{t_5}_{\text{a term}} \vee t_6 \vee t_7) \wedge \overbrace{(t_8 \vee t_9)}^{\text{a clause}}$$

Each parenthesized expression is called a *clause*. A *clause* is either a (1) *term* or *literal*; (2) a disjunction of two or more literals; or (3) the empty clause represented

Law	Expression
Commutative	$p \vee q \equiv q \vee p$ $p \wedge q \equiv q \wedge p$
Associative	$(p \vee q) \vee r \equiv p \vee (q \vee r)$ $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Distributive	$(p \wedge q) \vee r \equiv (p \vee r) \wedge (q \vee r)$ $(p \vee q) \wedge r \equiv (p \wedge r) \vee (q \wedge r)$
DeMorgan's	$\neg(p \vee q) \equiv \neg p \wedge \neg q$ $\neg(p \wedge q) \equiv \neg p \vee \neg q$

Table 14.5 The Commutative, Associative, and Distributive Rules of Boolean Algebra as Well as *DeMorgan's Laws* Are Helpful for Rewriting Propositions in CNF.

by the symbol $\emptyset$ or $\square$. We convert each wff in our knowledge base to a set of clauses:

$$\text{a wff} \rightarrow \text{a wff in CNF} \rightarrow \text{a set of clauses}$$

Thus, the entire knowledge base is represented as a set of clauses:

$$\text{knowledge base} \rightarrow \text{a set of wffs} \rightsquigarrow \text{a set of clauses}$$

While converting a proposition to CNF, we can use the equivalence between $p \supset q$ and $\neg p \vee q$ to eliminate $\supset$ in propositions. The commutative, associative, and distributive rules of Boolean algebra as well as *DeMorgan's Laws* are also helpful for rewriting propositions in CNF (Table 14.5). For instance, using DeMorgan's Laws we can express implication using conjunction and negation:

$$p \supset q \equiv \neg p \vee q \overset{\text{double negation}}{\equiv} \neg p \vee \neg(\neg q) \overset{\text{DeMorgan's Laws}}{\equiv} \neg(p \wedge \neg q)$$

The following are the propositions given previously expressed in CNF:

$$\begin{aligned} &(\text{atLeast35yearsOld}(X) \vee \neg \text{presidentOfUSA}(X)) \\ &(\text{drinks}(X, \text{earlgrey})) \wedge (\neg \text{english}(X)) \end{aligned}$$

Additional examples of propositions in CNF include:

$$\begin{aligned} &(\text{drinks}(\text{ray}, \text{earlgrey}) \vee \neg \text{drinks}(\text{ray}, \text{tea}) \vee \neg \text{tea}(\text{earlgrey})) \\ &(\text{siblings}(\text{christina}, \text{maria}) \vee \text{cousins}(\text{christina}, \text{maria}) \vee \\ &\neg \text{grandfather}(\text{virgil}, \text{christina}) \vee \neg \text{grandfather}(\text{virgil}, \text{maria})) \end{aligned}$$

The use of CNF has multiple advantages:

- Existential quantifiers are unnecessary.
- Universal quantifiers are implicit in the use of variables in the atomic propositions.
- No operators other than conjunction and disjunction are required.
- All predicate calculus propositions can be converted to CNF.

The purpose of representing wffs in CNF is to deduce new propositions from them. The question is: What can we logically deduce from known axioms and theorems (i.e., the knowledge base) represented in CNF (i.e., $\mathcal{KB} \models \alpha$)? To answer this question we need *rules of inference*, sometimes collectively referred to as a *deductive apparatus*. A rule of inference particularly applicable to logic programming is the *rule of resolution*. The purpose of representing a set of propositions as a set of clauses is to simplify the process of resolution.

14.4 Resolution

14.4.1 Resolution in Propositional Calculus

There are multiple rules of inference in formal systems of logic that are used to infer new propositions from given propositions. For instance, *modus ponens* is a rule of inference: $(p \wedge (p \supset q)) \supset q$ (if p implies q , and p , therefore q), often written $\frac{p, p \supset q}{q}$. Application of *modus ponens* supports the elimination of antecedents (e.g., p) from a logical proof and, therefore, is referred to as the *rule of detachment*. *Resolution* is the primary rule of inference used in logic programming. Resolution is designed to be used with propositions in CNF. It can be stated as follows:

$$\frac{\neg p \vee q, \neg q \vee r}{\neg p \vee r}$$

This rule indicates that if $\neg p \vee q$ and $\neg q \vee r$ are assumed to be true, then $\neg p \vee r$ is true. According to the rule of resolution, given two propositions (e.g., $\neg p \vee q$ and $\neg q \vee r$) where the same term (e.g., q) is present in one and negated in another, a new proposition is deduced by uniting the two original propositions without the matched term (e.g., $\neg p \vee r$). The underlying intuition being that the proposition q does not contribute to the validity of $\neg p \vee r$. The main idea in the application of resolution is to find two propositions in CNF such that the negation of a term in one is present in the other. When two such propositions are found, they can be combined with a disjunction after canceling out the matched terms in both:

Given propositions:
$\neg p \vee q$
$\neg q \vee r$
After combining the two propositions, cancel out the matching, negated terms.
$\neg p \vee q \vee \neg q \vee r$
$\neg p \vee \cancel{q} \vee \cancel{\neg q} \vee r$
Inferred proposition:
$\neg p \vee r$

Thus, given the propositions $\neg p \vee q$ and $\neg q \vee r$, we can infer $\neg p \vee r$.

14.4.2 Resolution in Predicate Calculus

Resolution in propositional calculus similarly involves matching a proposition with its negation: p and $\neg p$. Resolution in predicate calculus is not as simple because the arguments of the predicates must be considered. The structure of the following resolution proof is the same as in the prior example, except that the propositions p , q , and r are represented as binary predicates:

Given propositions:
$\neg \text{siblings}(\text{angela}, \text{rosa}) \vee \text{friends}(\text{angela}, \text{rosa})$
$\neg \text{friends}(\text{angela}, \text{rosa}) \vee \text{talkdaily}(\text{angela}, \text{rosa})$
After combining the two propositions, cancel out the matching, negated terms.
$\neg \text{siblings}(\text{angela}, \text{rosa}) \vee \text{friends}(\text{angela}, \text{rosa}) \vee \neg \text{friends}(\text{angela}, \text{rosa}) \vee \text{talkdaily}(\text{angela}, \text{rosa})$
$\neg \text{siblings}(\text{angela}, \text{rosa}) \vee \cancel{\text{friends}(\text{angela}, \text{rosa})} \vee \cancel{\neg \text{friends}(\text{angela}, \text{rosa})} \vee \text{talkdaily}(\text{angela}, \text{rosa})$
Inferred proposition:
$\neg \text{siblings}(\text{angela}, \text{rosa}) \vee \text{talkdaily}(\text{angela}, \text{rosa})$

At present, we are not concerned with any intended semantics of any propositions, but are simply exploring the mechanics of resolution. Consider the example of an application of resolution in Table 14.6.

In the prior examples, the process of resolution started with the axioms (i.e., the propositions assumed to be true), from which was produced a new, inferred proposition. This approach to the application of resolution is called *forward chaining*. The question being asked is: What new propositions can we derive from the existing propositions? An alternative use of resolution is to test a hypothesis represented as a proposition for validity. We start by adding the negation of the hypothesis to the set of axioms and then run resolution. The process or resolution continues as usual until a contradiction is found, which indicates that the hypothesis is proved to be true (i.e., it is a theorem). This process produces a *proof by refutation*. Consider a knowledge base of one axiom *commuter(lucia)*

Given propositions: $\neg \text{grandfather}(\text{virgil}, \text{christina}) \vee \neg \text{grandfather}(\text{virgil}, \text{maria}) \vee \text{siblings}(\text{christina}, \text{maria}) \vee \text{cousins}(\text{christina}, \text{maria})$ $\neg \text{siblings}(\text{maria}, \text{angela}) \vee \neg \text{siblings}(\text{christina}, \text{maria}) \vee \text{siblings}(\text{christina}, \text{angela})$	
After combining the two propositions, cancel out the matching, negated terms: $\neg \text{grandfather}(\text{virgil}, \text{christina}) \vee \neg \text{grandfather}(\text{virgil}, \text{maria}) \vee \cancel{\text{siblings}(\text{christina}, \text{maria})} \vee \text{cousins}(\text{christina}, \text{maria}) \vee$ $\neg \text{siblings}(\text{maria}, \text{angela}) \vee \neg \cancel{\text{siblings}(\text{christina}, \text{maria})} \vee \text{siblings}(\text{christina}, \text{angela})$	
Inferred proposition: $\neg \text{grandfather}(\text{virgil}, \text{christina}) \vee \neg \text{grandfather}(\text{virgil}, \text{maria}) \vee \neg \text{siblings}(\text{maria}, \text{angela}) \vee \text{cousins}(\text{christina}, \text{maria}) \vee \text{siblings}(\text{christina}, \text{angela})$	

Table 14.6 An Example Application of Resolution

and the hypothesis $\text{commuter}(\text{lucia})$. We add $\neg\text{commuter}(\text{lucia})$ to the knowledge base and run resolution:

Given propositions:	
	$\text{commuter}(\text{lucia})$
negated hypothesis:	$\neg\text{commuter}(\text{lucia})$
Combining the two propositions results in a <i>contradiction</i> !	
$\text{commuter}(\text{lucia}) \vee \neg\text{commuter}(\text{lucia})$	

Thus, the hypothesis $\text{commuter}(\text{lucia})$ is true.

The presence of variables in propositions represented as predicates makes matching propositions during the process of resolution considerably more complex than the process demonstrated in the preceding examples. The process of “matching propositions” is formally called unification. *Unification* is the activity of finding a *substitution* or mapping that, when applied, renders two terms equivalent. The substitution is said to *unify* the two terms. Unification in the presence of variables requires instantiation—the temporary binding values to variables. The instantiation is temporary because the unification process often involves *backtracking*. *Instantiation* is the process of finding values for variables that will foster unification; it recurs throughout the process of unification. Consider the example of a resolution proof by refutation involving variables in Table 14.7, where the hypothesis to be proved is $\text{rides}(\text{lucia}, \text{train})$. Since a contradiction is found, the hypothesis $\text{rides}(\text{lucia}, \text{train})$ is proved to be true.

14.5 From Predicate Calculus to Logic Programming

14.5.1 Clausal Form

To prepare propositions in CNF for use in logic programming, we must further simplify their form, with the ultimate goal being to simplify the resolution process. Consider the following proposition expressed in CNF:

$$\overbrace{(\neg A_1 \vee \neg A_2 \cdots \vee \neg A_m \vee B_1 \vee B_2 \vee \cdots \vee B_n)}^{\text{clause } c_1} \wedge \overbrace{(t_1 \vee t_2)}^{\text{clause } c_2} \wedge \underbrace{(t_3)}_{\text{a term}}^{\text{clause } c_3}$$

We convert each clause in this expression into *clausal form*, which is a standard and simplified syntactic form for propositions:

$$\overbrace{B_1 \vee B_2 \vee \cdots \vee B_n}^{\text{consequent}} \subset \overbrace{A_1 \wedge \underbrace{A_2}_{\text{a term}} \wedge \cdots \wedge A_m}^{\text{antecedent}}$$

The A s and B s are called *terms*. The left-hand side (i.e., the expression before the $\subset$ symbol) is called the *consequent*; the right-hand side (i.e., the expression after the $\subset$ symbol) is called the *antecedent*. The intuitive interpretation of a proposition in

Knowledge Base	
clause 1:	$\neg \text{commuter}(x) \vee \neg \text{doesnot have}(x, \text{car}) \vee \text{rides}(x, \text{bus})$
clause 2:	$\neg \text{commuter}(x) \vee \neg \text{doesnot have}(x, \text{bicycle}) \vee \text{rides}(x, \text{train})$
clause 3:	$\text{commuter}(\text{lucia})$
clause 4:	$\text{doesnot have}(\text{lucia}, \text{bicycle})$
clause 5:	$\neg \text{rides}(\text{lucia}, \text{train})$ (negated hypothesis)
Resolution Proof by Refutation	
Using clauses 2 and 5:	$\neg \text{commuter}(x) \vee \neg \text{doesnot have}(x, \text{bicycle}) \vee \text{rides}(x, \text{train}) \vee \neg \text{rides}(\text{lucia}, \text{train})$
We must <i>instantiate</i> x to <i>lucia</i> to <i>unify</i> the terms to be canceled out.	
	$\neg \text{commuter}(\text{lucia}) \vee \neg \text{doesnot have}(\text{lucia}, \text{bicycle}) \vee \text{rides}(\text{lucia}, \text{train}) \vee \neg \text{rides}(\text{lucia}, \text{train})$
	$\neg \text{commuter}(\text{lucia}) \vee \neg \text{doesnot have}(\text{lucia}, \text{bicycle}) \vee \text{rides}(\text{lucia}, \text{train}) \vee \neg \text{rides}(\text{lucia}, \text{train})$
Using clause 4:	$\neg \text{commuter}(\text{lucia}) \vee \neg \text{doesnot have}(\text{lucia}, \text{bicycle})$
	$\neg \text{commuter}(\text{lucia}) \vee \neg \text{doesnot have}(\text{lucia}, \text{bicycle}) \vee \text{doesnot have}(\text{lucia}, \text{bicycle})$
	$\neg \text{commuter}(\text{lucia}) \vee \neg \text{doesnot have}(\text{lucia}, \text{bicycle}) \vee \text{doesnot have}(\text{lucia}, \text{bicycle})$
Using clause 3:	$\neg \text{commuter}(\text{lucia}) \vee \text{commuter}(\text{lucia})$ contradiction!

Table 14.7 An Example of a Resolution Proof by Refutation, Where the Propositions Therein Are Represented in CNF

clausal form is as follows: If *all* of the A s are true, then *at least one* of the B s must be true. When converting the individual clauses in an expression in CNF into clausal form, we introduce implication based on equivalence between $\neg p \vee q$ and $q \subset p$. The clauses C_1 and C_2 given previously expressed in clausal form are

$$\begin{aligned}\text{clause } C_1: & B_1 \vee B_2 \vee \cdots \vee B_n \subset A_1 \wedge A_2 \wedge \cdots \wedge A_m \\ \text{clause } C_2: & t_2 \subset \neg t_1 \\ \text{clause } C_3: & t_3\end{aligned}$$

Thus, a single proposition expressed in CNF is converted into a set of propositions in clausal form. Notice that we used the DeMorgan Law $\neg p \vee \neg q \equiv \neg(p \wedge q)$ to convert the $(\neg A_1 \vee \neg A_2 \cdots \vee \neg A_m)$ portion of clause C_2 to the antecedent of the proposition in clausal form. In particular,

$$\begin{aligned}(\neg A_1 \vee \neg A_2 \cdots \vee \neg A_m) \vee (B_1 \vee B_2 \vee \cdots \vee B_n) &\equiv \\ \neg(A_1 \wedge A_2 \cdots \wedge A_m) \vee (B_1 \vee B_2 \vee \cdots \vee B_n) &\equiv \\ (B_1 \vee B_2 \vee \cdots \vee B_n) \subset \neg(\neg(A_1 \wedge A_2 \cdots \wedge A_m)) &\equiv \\ (B_1 \vee B_2 \vee \cdots \vee B_n) \subset (A_1 \wedge A_2 \cdots \wedge A_m)\end{aligned}$$

The first of the other clauses expressed in clausal form is

$$\text{atLeast35yearsOld}(X) \subset \text{presidentOfUSA}(X)$$

(If X is/was a president of the United States, then X is/was at least 35 years old.)

Examples of other propositions in clausal form follow:

$$\begin{aligned}\text{siblings}(\text{christina}, \text{maria}) \vee \text{cousins}(\text{christina}, \text{maria}) &\subset \\ \text{grandfather}(\text{virgil}, \text{christina}) \wedge \text{grandfather}(\text{virgil}, \text{maria}) &\subset \\ \text{(If Virgil is the grandfather of Christina and Virgil is the grandfather} & \\ \text{of Maria, then Christina and Maria are either siblings or cousins.)} &\end{aligned}$$

$$\text{drinks}(\text{ray}, \text{earlgrey}) \subset \text{drinks}(\text{ray}, \text{tea}) \wedge \text{tea}(\text{earlgrey})$$

(If Ray drinks tea and Earl Grey is a type of tea, then Ray drinks Earl Grey.)

14.5.2 Horn Clauses

A restriction that can be applied to propositions in clausal form is to limit the right-hand side to at most one term. Propositions in clausal form adhering to this additional restriction are called Horn clauses. A *Horn clause* is a proposition with either exactly zero terms or one term in the consequent. Horn clauses conform to one of the three clausal forms shown in Table 14.8. A *headless Horn clause* is a proposition with no terms in the consequent (e.g., $\{\} \subset p$). A *headed Horn clause* is a proposition with exactly one atomic term in the consequent (e.g., $q \subset p$). The last proposition in clausal form in the prior subsection is a headed Horn clause. Table 14.8 provides examples of these types of Horn clauses.

Type of Horn Clause	Form	Example
headless	$false \subset B_1 \wedge \dots \wedge B_n, n \geq 1$	$false \subset \text{philosopher}(\text{Pascal})$
headed	$A \subset true$	$\text{drinks}(\text{ray}, \text{earlgrey}) \subset true$
headed	$A \subset B_1 \wedge \dots \wedge B_n, n \geq 1$	$\text{drinks}(\text{ray}, \text{earlgrey}) \subset \text{drinks}(\text{ray}, \text{tea}) \wedge \text{tea}(\text{earlgrey})$

Table 14.8 Types of Horn Clauses with Forms and Examples

14.5.3 Conversion Examples

To develop an understanding of the representation of propositions in a variety of representations, including CNF and clausal form as Horn clauses, consider the following conversion examples.

Factorial

- **Natural language specification:**

The factorial of zero is 1.

The factorial of a positive integer n is n multiplied by the factorial of $n - 1$.

- **Predicate calculus:**

$\text{factorial}(0, 1)$

$\forall n, \forall g.$

$\text{factorial}(n, n * g) \subset \neg \text{zero}(n) \wedge \neg \text{negative}(n) \wedge \text{factorial}(n - 1, g)$

- **Conjunctive normal form:**

$(\text{factorial}(0, 1)) \wedge$

$(\text{zero}(n) \vee \text{negative}(n) \vee \neg \text{factorial}(n - 1, g) \vee \text{factorial}(n, n * g))$

- **Horn clauses:**

$\text{factorial}(0, 1)$

$\text{factorial}(n, n * g) \subset \neg \text{zero}(n) \wedge \neg \text{negative}(n) \wedge \text{factorial}(n - 1, g)$

Fibonacci

- **Natural language specification:**

The first Fibonacci number is 0.

The second Fibonacci number is 1.

Any Fibonacci number n , except for the first and second, is the sum of the previous two Fibonacci numbers.

- **Predicate calculus:**

$\text{fibonacci}(1, 0)$

$\text{fibonacci}(2, 1)$

$\forall n, \forall g, \forall h.$

$\text{fibonacci}(n, g + h) \subset \neg \text{negative}(n) \wedge \neg \text{zero}(n) \wedge \neg \text{one}(n) \wedge \neg \text{two}(n) \wedge$

$\text{fibonacci}(n - 1, g) \wedge \text{fibonacci}(n - 2, h)$

- **Conjunctive normal form:**

$$\begin{aligned} & (\text{fibonacci}(1, 0)) \wedge \\ & (\text{fibonacci}(2, 1)) \wedge \\ & (\text{fibonacci}(n, g + h) \vee \text{negative}(n) \vee \text{zero}(n) \vee \text{one}(n) \vee \text{two}(n) \vee \\ & \neg \text{fibonacci}(n - 1, g) \vee \neg \text{fibonacci}(n - 2, h)) \end{aligned}$$

- **Horn clauses:**

$$\begin{aligned} & \text{fibonacci}(1, 0) \\ & \text{fibonacci}(2, 1) \\ & \text{fibonacci}(n, g + h) \subset \neg \text{negative}(n) \wedge \neg \text{zero}(n) \wedge \neg \text{one}(n) \wedge \neg \text{two}(n) \wedge \\ & \text{fibonacci}(n - 1, g) \wedge \text{fibonacci}(n - 2, h) \end{aligned}$$

Commuter

- **Natural language specification:**

For all x , if x is a commuter, then x rides either a bus or a train.

- **Predicate calculus:**

$$\forall x. (\text{rides}(x, \text{bus}) \vee \text{rides}(x, \text{train}) \subset \text{commuter}(x))$$

- **Conjunctive normal form:**

$$(\text{rides}(x, \text{bus}) \vee \text{rides}(x, \text{train}) \vee \neg \text{commuter}(x))$$

- **Horn clause:**

$$\text{rides}(x, \text{bus}) \subset \text{commuter}(x) \wedge \neg \text{rides}(x, \text{train})$$

Sibling relationship

- **Natural language specification:**

x is a sibling of y if x and y have the same mother or the same father.

- **Predicate calculus:**

$$\forall x, \forall y. ((\exists m. \text{sibling}(x, y) \subset \text{mother}(m, x) \wedge \text{mother}(m, y)) \vee (\exists f. \text{sibling}(x, y) \subset \text{father}(f, x) \wedge \text{father}(f, y)))$$

- **Conjunctive normal form:**

$$\begin{aligned} & (\neg \text{mother}(m, x) \vee \neg \text{mother}(m, y) \vee \text{sibling}(x, y)) \wedge \\ & (\neg \text{father}(f, x) \vee \neg \text{father}(f, y) \vee \text{sibling}(x, y)) \end{aligned}$$

- **Horn clauses:**

$$\begin{aligned} \text{sibling}(x, y) & \subset \text{mother}(m, x) \wedge \text{mother}(m, y) \\ \text{sibling}(x, y) & \subset \text{father}(f, x) \wedge \text{father}(f, y) \end{aligned}$$

Recall that the universal quantifier is implicit and the existential quantifier is not required in Horn clauses: All variables on the left-hand side (lhs) of the $\subset$ operator are universally quantified and those on the right-hand side (which do not appear on the lhs) are existentially quantified.

In summary, to prepare the propositions in a knowledge base for use with Prolog, we must convert the wffs in the knowledge base to a set of Horn clauses:

a set of wffs $\rightsquigarrow$ a set of Horn clauses

We arrive at the final knowledge base of Horn clauses by applying the following conversion process on each wff in the original knowledge base:

convert each clause in the CNF to clausal form

a wff $\rightarrow$ a wff in CNF $\rightarrow$ a set of clauses in clausal form $\rightarrow$ a set of Horn clauses

Since more than one Horn clause may be required to represent a single wff, the number of propositions in the original knowledge base of wffs may not equal the number of Horn clauses in the final knowledge base.

14.5.4 Motif of Logic Programming

The purpose of expressing propositions as Horn clauses is to prepare them for use in a logic programming system like Prolog. Logic programs are composed as a set of facts and rules. A *fact* is an axiom that is asserted as true. A *rule* is a declaration expressed in the form of an *if-then* statement. A *headless Horn clause* is called a *goal* (called a *hypothesis* in Section 14.4.2). A *headed Horn clause* with an empty antecedence is called a *fact*, while a *headed Horn clause* with a non-empty antecedent is called a *rule*. Note that the headless Horn clause $\{\} \subset \text{philosopher(Pascal)}$ representing a goal is the same as $\text{false} \subset \text{philosopher(Pascal)}$; and the headed Horn clause $\text{weather(raining)} \subset \{\}$ representing a fact is the same as $\text{weather(raining)} \subset \text{true}$.

In a logic programming system like Prolog the programmer declares/asserts facts and rules, and then asks questions or, in other words, pursues goals. For instance, to prove a given goal Q , the system must either

1. Find Q as a fact in the database, or
2. Find Q as the logical consequence of a sequence of propositions:

$$\begin{aligned}
 P_2 &\subset P_1 \\
 P_3 &\subset P_2 \\
 &\dots \\
 &\dots \\
 &\dots \\
 P_n &\subset P_{n-1} \\
 Q &\subset P_n
 \end{aligned}$$

14.5.5 Resolution with Propositions in Clausal Form

Forward Chaining

To apply resolution to two propositions $\mathcal{X}$ and $\mathcal{Y}$ represented in clausal form, take the disjunction of the consequences of $\mathcal{X}$ and $\mathcal{Y}$, take the conjunction of the antecedents of $\mathcal{X}$ and $\mathcal{Y}$, and cancel out the common terms on each side of the $\subset$ symbol in the new proposition:

$$\frac{\frac{q \subset p}{r \subset q}}{q \vee r \subset p \wedge q}$$

$$\frac{\cancel{q \vee r} \subset p \wedge \cancel{q}}{r \subset p}$$

Thus, given $q \subset p$ and $r \subset q$, we can infer $r \subset p$. Table 14.9 is an example of an application of resolution, where the propositions therein are represented in clausal form rather than CVF (using the example in Section 14.4.2). The new proposition inferred here indicates that “if Virgil is the grandfather of Christina and Maria, and Maria and Angela are siblings, then either Christina and Maria are cousins or Christina and Angela are siblings.”

Restricting propositions in clausal form to Horn clauses further simplifies the rule of resolution, which can be restated as follows:

$$\frac{(q \subset p), (r \subset q)}{r \subset p}$$

This rule indicates that if p implies q and q implies r , then p implies r . The mechanics of a resolution proof process over Horn clauses are slightly different than those for propositions expressed in CNF, as detailed in Section 14.4.2. In particular, given two Horn clauses $\mathcal{X}$ and $\mathcal{Y}$, if we can match the head of $\mathcal{X}$ with a term in the antecedent of clause $\mathcal{Y}$, then we can replace the matched head of $\mathcal{X}$ in the antecedent of $\mathcal{Y}$ with the antecedent of $\mathcal{X}$. Consider the following two Horn clauses $\mathcal{X}$ and $\mathcal{Y}$:

$$\begin{aligned} \mathcal{X}: \quad & \mathbf{p} \subset p_1 \wedge \cdots \wedge p_n \\ \mathcal{Y}: \quad & \mathbf{q} \subset \mathbf{p} \wedge \cdots \wedge q_{i-1} \wedge q_i \wedge q_{i+1} \wedge \cdots \wedge q_m \end{aligned}$$

Since term $\mathbf{p}$ in the antecedent of clause $\mathcal{Y}$ matches term $\mathbf{p}$ (i.e., the head of clause $\mathcal{X}$), we can infer the following new proposition:

$$\mathcal{Y}': \quad q \subset q_1 \wedge \cdots \wedge q_{i-1} \wedge p_1 \wedge \cdots \wedge p_n \wedge q_{i+1} \wedge \cdots \wedge q_m$$

where $\mathbf{p}$ in the body of proposition $\mathcal{Y}$ is replaced with $p_1 \wedge \cdots \wedge p_n$ from the body of proposition $\mathcal{X}$ to produce $\mathcal{Y}'$. Consider an application of resolution to two simple Horn clauses, $q \subset p$ and $r \subset q$:

$$\frac{\frac{q \subset p}{r \subset q}}{r \subset p}$$

$siblings(christina, maria) \vee cousins(christina, maria) \subset grandfather(virgil, christina) \wedge grandfather(virgil, maria)$ (If Virgil is the grandfather of Christina and Virgil is the grandfather of Maria, then Christina and Maria are either siblings or cousins.) $siblings(christina, angela) \subset siblings(maria, angela) \wedge siblings(christina, maria)$ (If Maria and Angela are siblings and Christina and Maria are siblings, then Christina and Angela are siblings.)
$siblings(christina, maria) \vee cousins(christina, maria) \vee grandfather(virgil, christina) \wedge grandfather(virgil, maria) \wedge$ $siblings(christina, angela) \subset siblings(maria, angela) \wedge siblings(christina, maria)$ $siblings(christina, maria) \vee cousins(christina, maria) \vee grandfather(virgil, christina) \wedge grandfather(virgil, maria) \wedge$ $siblings(christina, angela) \subset siblings(maria, angela) \wedge siblings(christina, maria)$
$cousins(christina, maria) \vee siblings(christina, angela) \subset grandfather(virgil, christina) \wedge grandfather(virgil, maria) \wedge$ $siblings(maria, angela)$

Table 14.9 An Example Application of Resolution, Where the Propositions Therein Are Represented in Clausal Form

Thus, given $q \subset p$ and $r \subset q$, we can infer $r \subset p$. Consider the following resolution example from Section 14.4.2, where the propositions are expressed as Horn clauses:

$$\begin{array}{lcl} \text{friends}(\text{angela}, \text{rosa}) & \subset & \text{siblings}(\text{angela}, \text{rosa}) \\ \text{talkdaily}(\text{angela}, \text{rosa}) & \subset & \text{friends}(\text{angela}, \text{rosa}) \\ \hline \text{talkdaily}(\text{angela}, \text{rosa}) & \subset & \text{siblings}(\text{angela}, \text{rosa}) \end{array}$$

The structure of this resolution proof is the same as the structure of the prior example, but the propositions p , q , and r are represented as binary predicates. The proof indicates that

“If Angela and Rosa are siblings, then Angela and Rosa are friends”; and
 “if Angela and Rosa are friends, then Angela and Rosa talk daily”; then
 “if Angela and Rosa are siblings, then Angela and Rosa talk daily.”

Backward Chaining

A *goal* in logic programming, which is called a hypothesis in Section 14.4.2, is expressed as a headless Horn clause and is similarly pursued through a *resolution proof by contradiction*: Assert the goal as a false fact in the database and then search for a contradiction. In particular, resolution searches the database of propositions for the head of the known Horn clause $\mathcal{P}$ that unifies with a term in the antecedent of the headless Horn goal clause $\mathcal{G}$ representing the negated goal. If a match is found, the antecedent of the Horn clause $\mathcal{P}$ whose head matched a term in the antecedent of $\mathcal{G}$ is replaced with the unified term in $\mathcal{G}$. This process continues until a contradiction is found:

$$\begin{array}{lcl} \text{a rule:} & \mathbf{p} & \subset \quad p_1 \wedge \cdots \wedge p_n \\ \text{the goal:} & \text{false} & \subset \quad \mathbf{p} \\ \hline \text{new subgoals:} & \text{false} & \subset \quad p_1 \wedge \cdots \wedge p_n \end{array}$$

We unify the body of the goal with the head of one of the known clauses, and replace the matched goal with the antecedent of the clause, creating a new list of (sub-)goals. In this example, the resolution process replaces the original goal p with the subgoals $p_1 \wedge \cdots \wedge p_n$. If, after multiple iterations of this process, a contradiction (i.e., $\text{true} \subset \text{false}$) is derived, then the goal is satisfied.

Consider a database consisting of only one fact: $\text{commuter}(\text{lucia}) \subset \text{true}$. To pursue the goal of determining if “Lucia is a commuter,” we add a negation of this proposition expressed as the headless Horn clause $\text{false} \subset \text{commuter}(\text{lucia})$ to the database and run the resolution algorithm:

$$\begin{array}{lcl} \text{a fact } \mathcal{P}: & \text{commuter}(\text{lucia}) & \subset \quad \text{true} \\ \text{a goal } \mathcal{G}: & \text{false} & \subset \quad \text{commuter}(\text{lucia}) \\ \hline & \text{Matching head of } \mathcal{P} \text{ with body of } \mathcal{G}, \\ & \text{and replacing body of } \mathcal{P} \text{ with matched body of } \mathcal{G}: \\ \hline \text{a contradiction:} & \text{false} & \subset \quad \text{true} \end{array}$$

This is a simple fact-checking example. Since the outcome of resolution is a contradiction, the goal $\mathcal{G} \text{commuter}(\text{lucia})$ is satisfied.

In contrast to the application of resolution in a *forward-chaining* manner as demonstrated in Section 14.4.2, the resolution process here attempts to prove a goal by working backward from that goal—a process called *backward chaining*. Table 14.10 is a proof using this backward-chaining style of resolution to satisfy the goal $\text{false} \subset \text{rides}(\text{lucia}, \text{train})$, where the propositions therein are expressed as Horn clauses (from the example in Section 14.4.2). Since the outcome of resolution is a contradiction, the goal $\text{rides}(\text{lucia}, \text{train})$ is satisfied. Unlike the forward-chaining proof of $\text{rides}(\text{lucia}, \text{train})$ in Section 14.4.2, here we proved the goal $\text{rides}(\text{lucia}, \text{train})$ by reasoning from the goal backward toward a contradiction. Prolog uses *backward chaining*; CLIPS uses *forward chaining* (discussed in Section 14.10).

14.5.6 Formalism Gone Awry

The implementation of resolution in a computer system is problematic. Both the order in which to search the database (e.g., top-down, bottom-up, or other) and the order in which to prove subgoals (e.g., left-to-right, right-to-left, or other) during resolution is significant. For instance, consider that in our previous example, an attempt to prove the goal $\text{false} \subset \text{rides}(\text{lucia}, \text{train})$ led to the need to prove the two subgoals: $\text{false} \subset \text{commuter}(x) \wedge \text{doesnothave}(x, \text{bicycle})$. In this example, the end result of the proof (i.e., true) is the same if we attempt to prove the subgoal $\text{commuter}(x)$ first and the subgoal $\text{doesnothave}(x, \text{bicycle})$ second, or vice versa. However, in other proofs, different orders can lead to different results (Section 14.7.1). Prolog searches its database and subgoals in a deterministic order during resolution, and programmers must be aware of the subtleties of the search process (Section 14.7.1). This violates a defining principle of declarative programming—that is, the programmer need only be concerned with the logic and leave the control (i.e., inference methods used to prove a hypothesis) up to the system. Kowalski (1979) captured the essence of logic programming with the following expression:

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

In this equation, the declaration of the facts and rules—the Logic—is independent of the Control. In other words, the construction of logic programs must be independent of program control. To be completely independent of control, predicates and the clauses therein must be evaluable either in any order or concurrently. The goal of logic programming is to make programming entirely an activity of specification, such that programmers should not have to impart control upon the program.

14.6 The Prolog Programming Language

Prolog, which stands for PROgramming in LOGic, is a language supporting a declarative/logic style of programming that was developed in the early 1970s

clause 1: clause 2: clause 3: clause 4: original goal:	Knowledge Base $rides(x, bus) \sqsubset commuter(x) \wedge doesnothave(x, car)$ $rides(x, train) \sqsubset commuter(x) \wedge doesnothave(x, bicycle)$ $commuter(lucia) \sqsubset true$ $doesnothave(lucia, bicycle) \sqsubset true$ $false \sqsubset rides(lucia, train)$
<i>new goal</i> (with two subgoals)	To use clause 2, we need <i>unification</i> and must <i>instantiate</i> <i>x</i> to <i>lucia</i>: $false \sqsubset commuter(lucia) \wedge doesnothave(lucia, bicycle)$
<i>new goal:</i>	Using clause 3: $false \sqsubset doesnothave(lucia, bicycle)$
	Using clause 4 results in a contradiction: $false \sqsubset true$

Table 14.10 An Example of a Resolution Proof Using Backward Chaining

Type of Horn Clause	Example Horn Clause	Prolog Concept	Prolog Syntax
headless	$false \subset \text{philosopher}(\text{Pascal})$	goal/query	<code>philosopher(pascal).</code>
headed	$\text{drinks}(\text{ray}, \text{earlgrey}) \subset \text{true}$	fact	<code>drinks(ray, earlgrey).</code>
headed	$\text{drinks}(\text{ray}, \text{earlgrey}) \subset$ $\text{drinks}(\text{ray}, \text{tea}) \wedge \text{tea}(\text{earlgrey})$ $\text{tea}(\text{earlgrey})$	rule	<code>drinks(ray, earlgrey) :-</code> <code>drinks(ray, tea),</code> <code>tea(earlgrey).</code>

Table 14.11 Mapping of Types of Horn Clauses to Prolog Clauses

for artificial intelligence applications. Traditionally, Prolog has been recognized as a language for artificial intelligence (AI) because of its support for logic programming, which was initially targeted at natural language processing. Since then, its use has expanded to other areas of AI, including expert systems and theorem proving. The resolution algorithm built into Prolog, along with the unification and backtracking techniques making resolution practical in a computer system, make its semantics more complex than those found in languages such as Python, Java, or Scheme.

14.6.1 Essential Prolog: Asserting Facts and Rules

In a Prolog program, knowledge is represented as *facts* and *rules* and a Prolog program consists of a set of facts and rules. A Prolog programmer asserts facts and rules in a program, and those facts and rules constitute the database or the knowledge base. Facts and rules are propositions that are represented as Horn clauses in Prolog (Table 14.11).

Facts. A *headed Horn clause* with an empty antecedence is called a *fact* in Prolog—an axiom or a proposition that is asserted as true. The fact “it is raining” can be declared in Prolog as: `weather(raining).`

Rules. A *headed Horn clause* with a non-empty antecedent is called a *rule*. A rule is a declaration that is expressed in the form of an *if-then* statement, and consists of a head (the consequent) and a body (the antecedent). We can declare the rule “if it is raining, then I carry an umbrella” in Prolog as follows: `carry(umbrella) :- weather(raining).` A rule can be thought of as a function. In Prolog, all functions are *predicates*—a function that returns true or false. (We can pass additional arguments to simulate returning values of other types.) Consider the following set of facts and rules in Prolog:

```

1  shape(circle).                /* a fact */
2  shape(square).                /* a fact */
3  shape(rectangle).            /* a fact */
4
5  rectangle(X) :- shape(square). /* a rule */
6  rectangle(X) :- shape(rectangle). /* a rule */

```

The facts on lines 1–3 assert that a circle, square, and rectangle are shapes. The two rules on lines 5–6 declare that shapes that are squares and rectangles are also rectangles. Syntactically, Prolog programs are built from terms. A *term* is either

a constant, a variable, or a structure. Constants and predicates must start with a lowercase letter, and neither have any intrinsic semantics—each means whatever the programmer wants it to mean. Variables must start with an uppercase letter or an underscore (i.e., `_`). The `X` on lines 5–6 is a variable. Recall that propositions (i.e., facts and rules) have no intrinsic semantics—each means whatever the programmer wants it to mean. Also, note that a period (`.`), not a semicolon (`;`)—which has another important function—terminates a fact and a rule.

14.6.2 Casting Horn Clauses in Prolog Syntax

The following are some of the Horn clauses given previously represented in Prolog syntax:

```
atLeast35yearsOld(X) :- presidentOfUSA(X) .

drinks(ray,earlgrey) :- drinks(ray,tea), tea(earlgrey) .

rides(X,bus) :- commuter(X), \+(rides(X,train)) .

sibling(X,Y) :- mother(M,X), mother(M,Y) .
sibling(X,Y) :- father(F,X), father(F,Y) .
```

Notice that the implication $\subset$ and conjunction $\wedge$ symbols are represented in Prolog as `:-` and `,`, respectively.

14.6.3 Running and Interacting with a Prolog Program

We use the SWI-Prolog⁷ implementation of Prolog in this chapter. There are two ways of consulting a database (i.e., compiling a Prolog program) in SWI-Prolog:

- Enter `swipl <filename>` at the (Linux) command line:

```
$ swipl first.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- make.
true.

?- halt.
$
```

- Use the built-in `consult/1`⁸ predicate (i.e., `consult(' <filename>')` or `[ <filename> ]`):

7. <https://www.swi-prolog.org>

8. The number following the `/` indicates the *arity* of the predicate. The `/<#>` is not part of the syntax of the predicate name.


```

$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- consult('first.pl').
true.

?- [first]. % abbreviated form of consult
true.

?- make.
true.

?- halt.
$

```

In either case, enter `make.` in the SWI-Prolog REPL to reconsult the loaded `prolog` program file (without exiting the interpreter), if (uncompiled) changes have been made to the program. Enter `halt.` or the EOF character (e.g., `<ctrl-D>` on Linux) to end your session with SWI-Prolog. Table 14.12 offers more information on this process.

Comments. A percent sign (i.e., `%`) introduces a single-line comment until the end of a line. C-style comments (i.e., `/* ... */`) are used for multi-line comments. Unlike in C, in Prolog multi-line comments can be nested.

Backtracking. The user can enter an `n` or `;` character to cause Prolog to *backtrack* up the search tree to find the next solution (i.e., substitution or *unification* of values to variables that leads to satisfaction of the stated goal). The built-in predicate `trace/0` allows the user to trace the resolution process (described next), including instantiations, as Prolog seeks to satisfy a goal.

Predicate	Semantics	Example
<code>make/0:</code>	reconsults/recompiles the loaded program	<code>make.</code>
<code>protocol/1:</code>	logs a transcript of the current session	<code>protocol('transcript').</code>
<code>halt/0</code> or EOF:	ends the current session	<code>halt.</code> or <code><ctrl-D></code>
<code>help/1:</code>	retrieves the manual page for a topic	<code>help(make).</code>
<code>apropos/1:</code>	searches the manual names and summaries	<code>apropos(protocol).</code>

Table 14.12 Predicates for Interacting with the SWI-Prolog Shell (i.e., REPL)

Program output. The built-in predicates `write`, `writeln`, and `nl` (for newline), with the implied semantics, write output. The programmer can include the following goal in a program to prevent Prolog from abbreviating results with ellipses:

```
set_prolog_flag(toplevel_print_options,
                [quoted(true), portray(true), max_depth(0)]).
```

The argument passed to `max_depth` indicates the maximum depth of the list to be printed. The maximum depth is 10 by default. If this value is set to 0, then the printing depth limit is turned off.

14.6.4 Resolution, Unification, and Instantiation

Once a database—a program—has been established, running the program involves asking questions or, in other words, pursuing goals. A *headless Horn clause* is called a *goal* (or *query*) in Prolog (Table 14.11). There is a distinction between a *fact* and a *goal* even though they appear in Prolog to be the same. The proposition *commuter(lucia) $\subset$ true* is a fact because its antecedence is always true. Conversely, the proposition *false $\subset$ commuter(lucia)* is a goal. Since both an empty antecedent and an empty consequent are omitted in Prolog, these two clauses can appear to be both facts or both goals. The goal *false $\subset$ commuter(x) $\wedge$ doesnothave(x, bicycle)* has two subgoals in its antecedent.

A Prolog interpreter acts as an inference engine. In Prolog, the user gives the inference engine a goal that the engine then sets out to satisfy (i.e., prove) based on the knowledge base of facts and rules (i.e., the program). In particular, when a goal is given, the inference engine attempts to match the goal with the head of a headed Horn clause, which can be either a fact or a rule. Prolog works backward from the goal using resolution to find a series of facts and rules that can be used to prove the goal (Section 14.5.5). This approach is called backward chaining because the inference engine works backward from a goal to find a path through the database sufficient to satisfy the goal. A more detailed examination of the process of resolution in Prolog is given in Section 14.7.1.

To run a program, the user supplies one or more goals, each in the form of a headless Horn clause. The activity of supplying a goal can be viewed as asking questions of the program or querying the system as one does through SQL with a database system (Section 14.7.9). Given the shape database from our previous example, we can submit the following queries:

```
1  ?- shape(circle).
2  true.
3  ?- shape(X).
4  X = circle;
5  X = square;
6  X = rectangle.
7  ?- shape(triangle).
8  false.
```

This small example involves multiple notable observations:

- Lines 1, 3, and 7 contain goals.
- A period (.), not a semicolon (;) terminates a fact, rule, or goal.
- After Prolog returns its first solution (line 4), the user can enter an ; or n character to cause Prolog to backtrack up the search tree to find the next solution (i.e., substitution of values for variables that leads to satisfaction of the stated goal), as shown on lines 5–6.
- Since an empty antecedent or consequent is omitted in the codification of a clause in Prolog, a fact and goal are syntactically indistinguishable from each other in Prolog. For instance, the clause `shape(circle) .` can be a fact [i.e., asserted proposition; $\text{shape}(\text{circle}) \subset \{\}$] or a goal [i.e., query; $\{\} \subset \text{shape}(\text{circle})$]. Thus, context is necessary to distinguish between the two. When a clause [e.g., `shape(circle) .`] is entered into a Prolog interpreter or appears on the left-hand side of a rule (i.e., the body or antecedent), then it is a goal or a subgoal, respectively. Otherwise, it is a fact.
- The case of the first letter of a term indicates whether it is interpreted as data (lowercase) or as a variable (uppercase). Variables must begin with a capital letter or an underscore. The term `circle` on line 1 is interpreted as data, while the term `X` on line 3 is interpreted as a variable.
- The goal `shape(X)` on line 3 involves a variable and returns as many values for `X` as we request for which the goal is true. Additional solutions are requested with a “;” or “n” keystroke.

Recall that the process of temporarily binding values to identifiers during resolution is called instantiation. The process of finding a substitution (i.e., a mapping) that, when applied, renders two terms equivalent is called unification and the substitution is said to unify the two terms. Two literals or constants only unify if they are the same literal:

```

1  ?- mary = mary.           3  ?- mary = martha.
2  true.                     4  false.
```

The substitution that unifies a variable with a literal or term binds the literal or term to the variable:

```

5  ?- X = mary.              14  ?- X = mary(X) .
6  X = mary.                 15  X = mary(X) .
7                               16
8  ?- mary = X.              17  ?- X = mary(Y) .
9  X = mary.                 18  X = mary(Y) .
10                             19
11  ?- X = mother(mary) .     20  ?- X = mary(name(Y)) .
12  X = mother(mary) .        21  X = mary(name(Y)) .
13
```

On lines 14–15, notice that a variable unifies with a term that contains an occurrence of the variable (see the discussion of *occurs-check* in Conceptual Exercise 14.8.3). A nested term can be unified with another term if the two

terms have the same (1) predicate name; (2) shape or nested structure; and (3) number of arguments, which can be recursively unified:

```

22  ?- name(Mary) = mother(Mary).      32  false.
23  false.                               33
24                                     34  ?- mother(olimpia, name(N)) =
25  ?- mother(olimpia,D) =              35  |   mother(M,lucia).
26  |   mother(M,lucia).                36  false.
27  D = lucia,                           37
28  M = olimpia.                         38  ?- mother(olimpia, name(N)) =
29                                     39  |   mother(M, name(lucia)).
30  ?- mother(X) =                       40  N = lucia,
31  |   mother(olimpia,lucia).           41  M = olimpia.

```

Lines 27–28 and 40–41 are substitutions that unify the clauses on lines 25–26 and 38–39, respectively. Lastly, to unify two uninstantiated variables, Prolog makes the variables aliases of each other, meaning that they point to the same memory location:

```

42  ?- Mary = Mary.                     44  ?- Mary = Martha.
43  true.                               45  Mary = Martha.

```

- If Prolog cannot prove a goal, it assumes the goal to be false. For instance, the goal `shape(triangle)` on line 7 in the first Prolog transcript given in this subsection fails (even though a triangle is a shape) because the process of resolution cannot prove it from the database—that is, there is neither a `shape(triangle) . fact` in the database nor a way to prove it from the set of facts and rules. This aspect of the inference engine in Prolog is called the *closed-world assumption* (Section 14.9.1).

The task of satisfying a goal is left to the inference engine, and not to the programmer.

14.7 Going Further in Prolog

14.7.1 Program Control in Prolog: A Binary Tree Example

The following set of facts describes a binary tree (lines 2–3). A `path` predicate is also included that defines a path between two vertices, with two rules, to be either an edge from `X` to `Y` (line 6) or a path from `X` to `Y` (line 7) through some intermediate vertex `Z` such that there is an edge from `X` to `Z` and a path from `Z` to `Y`:

```

1  /* edge(X,Y) declares there is a directed edge from vertex X to Y */
2  edge(a,b) .
3  edge(b,c) .
4
5  /* path(X,Y) declares there is a path from vertex X to Y */
6  path(X,Y) :- edge(X,Y) .
7  path(X,Y) :- edge(X,Z), path(Z,Y) .

```

Notice that the comma in the body (i.e., right-hand side) of the rule on line 7 represents conjunction. Likewise, the `:-` in that rule represents implication. Thus,

the rule $\text{path}(X, Y) :- \text{edge}(X, Z), \text{path}(Z, Y)$ is the Prolog equivalent of the Horn clause $\text{path}(X, Y) \subset \text{edge}(X, Z) \wedge \text{path}(Z, Y)$. The user can then query the program by expressing goals to determine whether the goal is true or to find all instantiations of variables that make the goal true. For instance, the goal $\text{path}(b, c)$ asks if there exists a path between vertices b and c :

```
?- path(b, c).
true .
? -
```

To prove this goal, Prolog uses resolution, which involves unification. When the goal $\text{path}(b, c)$ is given, Prolog runs its resolution algorithm with the following steps:

```
1. {} :- path(b, c).           /* the goal: a headless Horn clause */
2. {} :- edge(b, c).          /* unification using rule on line 6 */
3. {} :- {}                   /* unification using fact on line 3 */
```

During resolution, the term(s) in the body of the unified rule become subgoal(s). Consider the goal $\text{path}(X, c)$, which returns all the values of X that satisfy this goal:

```
?- path(X, c).
X = b ;
X = a ;
false.
?-
```

Prolog searches its database top-down and searches subgoals from left-to-right during resolution; thus, it constructs a search tree in a depth-first fashion. A top-down search of the database during resolution results in a unification between this goal and the head of the rule on line 6 and leads to the new goal: $\text{edge}(X, c)$. A proof of this new goal leads to additional unifications and subgoals. The entire search tree illustrating the resolution process is depicted in Figure 14.2. Source nodes in Figure 14.2 denote subgoals, and target nodes represent the body of a rule whose head unifies with the subgoal in the source. Edge labels in Figure 14.2 denote the line number of the rule involved in the unification from subgoal source to body target.

Notice that satisfaction of the goal $\text{edge}(X, c)$ involves backtracking to find alternative solutions. In particular, the solution $X=b$ is found first in the left subtree and the solution $X=a$ is found second in the right subtree. A source node with more than one outgoing edge indicates backtracking (1) to find solutions because searching for a solution in a prior subtree failed (e.g., see two source nodes in the right subtree each with two outgoing edges) or (2) to find additional solutions (e.g., second outgoing edge from the root node leads to the additional solution $X=a$).

Consider transposing the rules on lines 6 and 7 constituting the path predicate in the example database:

```
6 path(X, Y) :- edge(X, Z), path(Z, Y).
7 path(X, Y) :- edge(X, Y).
```

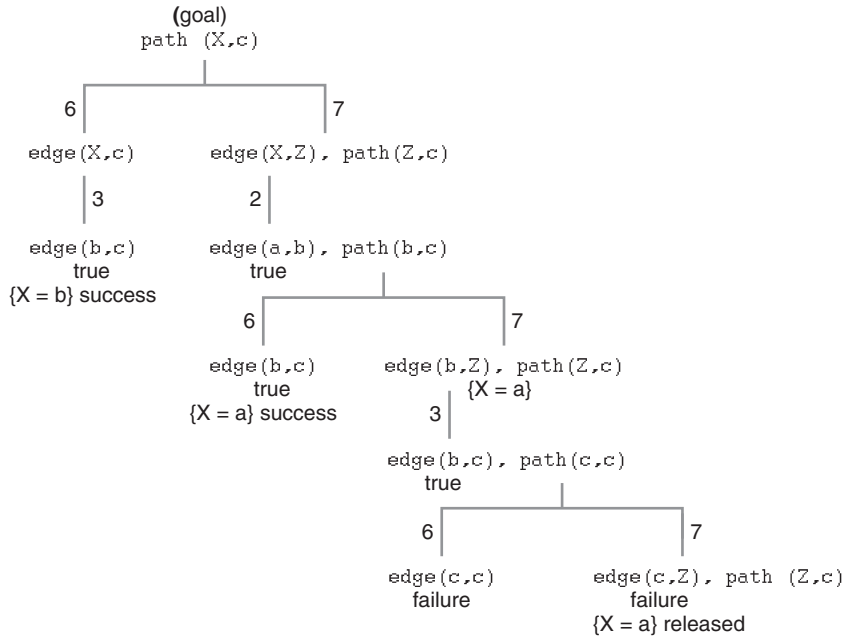


Figure 14.2 A search tree illustrating the resolution process used to satisfy the goal `path(X, c)`.

A top-down search of this modified database during resolution results in a unification of the goal `path(X, c)` with the head of the rule on line 6 and leads to two subgoals: `edge(X, Z), path(Z, c)`. A left-to-right pursuit of these two subgoals leads to additional unifications and subgoals, where the solution `X=a` is found before the solution `X=b`:

```

?- path(X, c) .
X = a ;
X = b .
?-

```

The entire search tree illustrating the resolution process with this modified database is illustrated in Figure 14.3. Notice the order of the terms in the body of the rule `path(X, Y) :- edge(X, Z), path(Z, Y)`. Left recursion is avoided in this rule since Prolog uses a depth-first search strategy. Consider a transposition of the terms in the body of the rule `path(X, Y) :- edge(X, Z), path(Z, Y)`:

```

6 path(X, Y) :- edge(X, Y) .
7 path(X, Y) :- path(Z, Y), edge(X, Z) .

```

The left-to-right pursuit of the subgoals leads to an infinite use of the rule `path(X, Y) :- path(Z, Y), edge(X, Z)` due to its left-recursive nature:

```

?- path(X, c) .
X = b ;

```

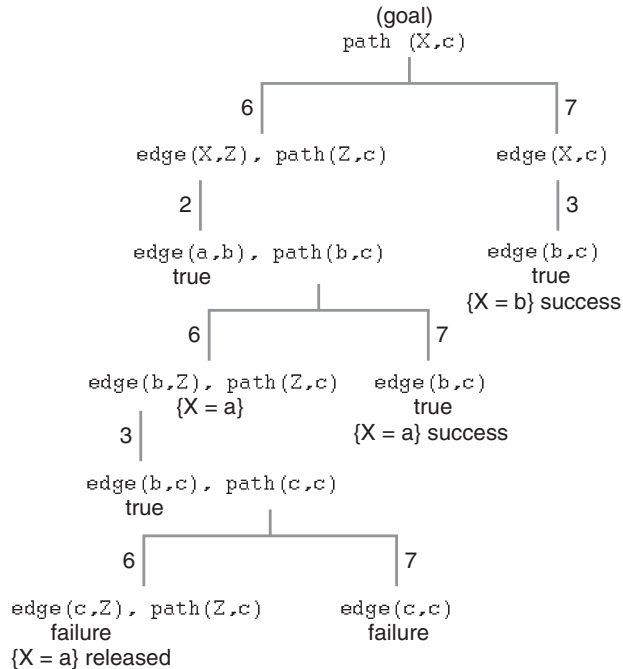


Figure 14.3 An alternative search tree illustrating the resolution process used to satisfy the goal `path(X, c)`.

```

X = a ;
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 1.0Gb, global: 28Kb, trail: 1Kb
ERROR:   Stack depth: 12,200,343, last-call: 0%, Choice points: 4
ERROR:   Probable infinite recursion (cycle):
ERROR:     [12,200,342] user:path(_7404, c)
ERROR:     [12,200,341] user:path(_7424, c)
?-

```

Since the database is also searched in a top-down fashion, if we reverse the two rules constituting the `path` predicate, the stack overflow occurs immediately and no solutions are returned:

```

6 path(X,Y) :- path(Z,Y),edge(X,Z) .
7 path(X,Y) :- edge(X,Y) .

?- path(X,c) .
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 1.0Gb, global: 23Kb, trail: 1Kb
ERROR:   Stack depth: 6,710,271, last-call: 0%, Choice points: 6,710,264
ERROR:   Probable infinite recursion (cycle):

```

The search tree for the goal `path(X, c)` illustrating the resolution process with this modified database is presented in Figure 14.4. Since Prolog terms are evaluated from left to right, `Z` will never be bound to a value. Thus, it is important to

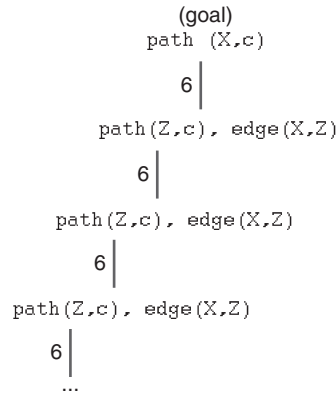


Figure 14.4 Search tree illustrating an infinite expansion of the `path` predicate in the resolution process used to satisfy the goal `path(X,c)`.

ensure that variables can be bound to values during resolution before they are used recursively.

Mutual recursion should also be avoided—to avert an infinite loop in the search, not a stack overflow:

```

/* The run-time stack will not be exhausted.
   Rather, there will be an infinite transfer of control. */

day_of_rain(X) :- day_of_umbrella_use(X).
day_of_umbrella_use(X) :- day_of_rain(X).
  
```

In summary, the order in which both the knowledge base in a Prolog program and the subgoals are searched and proved, respectively, during resolution is significant. While the order of the terms in the antecedent of a proposition in predicate calculus is insignificant (since conjunction is a commutative operator), Prolog pursues satisfaction of the subgoals in the body of a rule in a deterministic order. Prolog searches its database top-down and searches subgoals left-to-right during resolution and, therefore, constructs a search tree in a depth-first fashion (Figures 14.2–14.4). A Prolog programmer must be aware of the order in which the system searches both the database and the subgoals, which violates a defining principle of declarative programming—that is, the programmer need only be concerned with the logic and leave the control (i.e., inference methods used to satisfy a goal) up to the system. Resolution comes free with Prolog—the programmer need neither implement it nor be concerned with the details of its implementation. The goal of logic/declarative programming is to make programming entirely an activity of specification—programmers should not have to impart control upon the program. On this basis, Prolog falls short of the ideal. The language Datalog is a subset of Prolog. Unlike Prolog, the order of the clauses in a Datalog program is insignificant and has no effect on program control.

While a depth-first search strategy for resolution is efficient, it is *incomplete*; that is, DFS will not always result in solutions even if solutions exist. Thus,

Language	Sound	Complete	Turing-complete
Prolog	×	×	✓
Datalog	✓	✓	×

Table 14.13 A Comparison of Prolog and Datalog

Prolog	Haskell	Semantics
<code>[]</code>	<code>[]</code>	an <i>empty</i> list
<code>[X Y]</code>	<code>X:Y</code>	a list of <i>at least</i> one element
<code>[X, Y Z]</code>	<code>X:Y:Z</code>	a list of <i>at least</i> two elements
<code>[X, Y, Z W]</code>	<code>X:Y:Z:W</code>	a list of <i>at least</i> three elements
<code>.(X, Y)</code>	<code>X:Y</code>	
<code>[X]</code>	<code>X:nil</code>	a list of <i>exactly</i> one element
<code>[X, Y]</code>	<code>X:Y:nil</code>	a list of <i>exactly</i> two elements
<code>[X Y, Z]</code>	<code>N/A</code>	
<code>[X Y Z]</code>	<code>N/A</code>	

Table 14.14 Example List Patterns in Prolog Vis-à-Vis the Equivalent List Patterns in Haskell

Prolog, which uses DFS, is incomplete. In contrast, a breadth-first search strategy, while *complete* (i.e., BFS will always find solutions if any exist), is inefficient. However, Prolog and Datalog are both *sound*—neither will find incorrect solutions. Table 14.13 compares Prolog and Datalog.

14.7.2 Lists and Pattern Matching in Prolog

The built-in list data structures in Prolog and the associated pattern matching are nearly identical syntactically to those in ML/Haskell (Table 14.14). However, ML/Haskell, unlike Prolog, support currying and curried functions and a powerful and clean type and module system for creating abstract data types. As a result, ML and Haskell are used in AI for applications where Prolog (or Lisp) may have once been the only programming languages considered.

```

1  fruit(apple).
2  fruit(orange).
3  fruit('Pear').
4
5  likes('Olimpia',tangerines).
6  likes('Lucia',apples).
7  likes('Georgeanna',grapefruit).
8
9  composer('Johann Sebastian Bach').
10 composer('Rachmaninoff').
11 composer(beethoven).
12
13 sweet(_x) :- fruit(_x).
14
```

```

15 soundsgood(X) :- composer(X).
16 soundsgood(orange).
17
18 ilike([apples, oranges, pears]).
19 ilike([classical, [music, literature, theatre]]).
20 ilike([truth]).
21 ilike([[2020, mercedes, c300], [2021, bmw, m3]]).
22 ilike([[lisp, prolog], [apples, oranges, pears], ['ClaudeDebussy']]).
23 ilike(truth).
24 ilike(computerscience).

```

Notice the declarative nature of these predicates. Also, be aware that if we desire to include data in a Prolog program beginning with an uppercase letter, we must quote the entire string (lines 3, 5–10, and 22); otherwise, it will be treated as a variable. Similarly, if we desire to use a variable name beginning with a lowercase letter, we must preface the name with an underscore (`_`) (line 13). Consider the following the transcript of an interactive session with this database:

```

1  ?- fruit(Answer).
2
3  Answer = apple ;
4  Answer = orange ;
5  Answer = 'Pear'.
6
7  ?- fruit(answer).
8  false.
9
10 ?- fruit(X), fruit(Y).
11 X = Y, Y = apple ;
12 X = apple,
13 Y = orange ;
14 X = apple,
15 Y = 'Pear' ;
16 X = orange,
17 Y = apple ;
18 X = Y, Y = orange ;
19 X = orange,
20 Y = 'Pear' ;
21 X = 'Pear',
22 Y = apple ;
23 X = 'Pear',
24 Y = orange ;
25 X = Y, Y = 'Pear'.
26
27 ?- fruit(X), fruit(Y), X \= Y.
28 X = apple,
29 Y = orange ;
30 X = apple,
31 Y = 'Pear' ;
32 X = orange,
33 Y = apple ;
34 X = orange,
35 Y = 'Pear' ;
36 X = 'Pear',
37 Y = apple ;
38 X = 'Pear',
39 Y = orange ;
40 false.
41
42 ?- likes('Lucia', X),
43      | likes(X, apples).
44 false.
45
46 ?- composer(X).
47 X = 'Johann Sebastian Bach' ;
48 X = 'Rachmaninoff' ;
49 X = beethoven.
50
51 ?- ilike(X).
52
53
54 X = [apples, oranges, pears] ;
55 X = [classical,
56      [music, literature,
57       theatre]] ;
58 X = [truth] ;
59 X = [[2020, mercedes, c300],
60      [2021, bmw, m3]] ;
61 X = [[lisp, prolog],
62      [apples, oranges, pears],
63      ['ClaudeDebussy']] ;
64 X = truth ;
65 X = computerscience.
66
67 ?- ilike([X|Y]).
68 X = apples,
69 Y = [oranges, pears] ;
70 X = classical,
71 Y = [[music, literature,
72       theatre]] ;
73 X = truth,
74 Y = [] ;
75 X = [2020, mercedes, c300],
76 Y = [[2021, bmw, m3]] ;
77 X = [lisp, prolog],
78 Y = [[apples, oranges, pears],
79      ['ClaudeDebussy']].
80
81 ?- ilike([X,Y|Z]).
82 X = apples,
83 Y = oranges,
84 Z = [pears] ;

```

```

85 X = classical,
86 Y = [music, literature,
87      theatre],
88 Z = [] ;
89 X = [2020, mercedes, c300],
90 Y = [2021, bmw, m3],
91 Z = [] ;
92 X = [lisp, prolog],
93 Y = [apples, oranges, pears],
94 Z = ['ClaudeDebussy'].
95
96 ?- ilike([X,Y]).
97 X = classical,
98 Y = [music, literature,
99      theatre] ;
100 X = [2020, mercedes, c300],
101 Y = [2021, bmw, m3].
102
103 ?- ilike([X]).
104 X = truth.
105
106 ?- ilike([X,Y,Z]).
107 X = apples,
108 Y = oranges,
109 Z = pears ;
110 X = [lisp, prolog],
111 Y = [apples, oranges, pears],
112 Z = ['ClaudeDebussy'].
113
114 ?- halt.

```

Notice the use of *pattern matching* and *pattern-directed invocation* with lists in the queries on lines 67, 81, 96, and 103 (akin to their use in ML and Haskell in Sections B.8.3 and C.9.3, respectively, in the online ML and Haskell appendices). Moreover, notice the nature of some of the queries. For instance, the query on line 10 called a *cross-product* or *Cartesian product*. A *relation* is a subset of the Cartesian product of two or more sets. For instance, if $A = \{1, 2, 3\}$ and $B = \{a, b\}$, then a relation $R \subseteq A \times B = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$. The query on line 27 is also a Cartesian product, but one in which the pairs with duplicate components are pruned from the resulting relation.

14.7.3 List Predicates in Prolog

Consider the following list predicates using some of these list patterns:

```

1  isempty([]).
2
3  islist([]).
4  islist([_|_]).
5
6  cons(H,T,[H|T]).
7
8  /* member is built-in */
9  member1(E,[E|_]).
10 member1(E,[_|T]) :- member1(E,T).

```

Notice the declarative nature of these predicates as well as the use of *pattern-directed invocation* (akin to its use in ML and Haskell in Sections B.8.3 and C.9.3, respectively, in the online ML and Haskell appendices). The second fact (line 4) of the `islist` predicate indicates that a non-empty list consists of a head and a tail, but uses an underscore (`_`), with the same semantics as in ML/Haskell, to indicate that the contents of the head and tail are not relevant. The `cons` predicate accepts a head and a tail and puts them together in the third list argument. The `cons` predicate is an example of using an additional argument to simulate another return value. However, the fact `cons(H,T,[H,T])` is just a declaration—we need not think of it as a function. For instance, we can pursue the following goal to determine the components necessary to construct the list `[1,2,3]`:

```
?- cons(H,T,[1,2,3]).
H = 1,
T = [2, 3].

?-
```

Notice also that the `islist` and `cons` facts can be replaced with the rules `islist(_|T) :- islist(T).` and `cons(H,T,L) :- L = [H|T].`, respectively, without altering the semantics of the program. The `member1` predicate declares that an element of a list is either in the head position (line 9) or a member of the tail (line 10):

```
?- member1(E,[1,2,3]).
E = 1 ;
E = 2 ;
E = 3 ;
false.

?- member1(2,L).
L = [2|_10094] .

?- member1(2,L).
L = [2|_11572] ;
L = [_12230, 2|_12238] ;
L = [_12230, _12896, 2|_12904] ;
L = [_12230, _12896, _13562, 2|_13570] .

?-
```

14.7.4 Primitive Nature of `append`

The Prolog `append/3` predicate succeeds when its third list argument is the result of appending its first two list arguments. While `append` is built into Prolog, for purposes of instruction we define it as `append1`:

```
1 append1([],L,L).
2 append1(L,[],L).
3 append1([X|L1], L2, [X|L12]) :- append1(L1, L2, L12).

?- append1([a,b,c], [d,e,f], L).
L = [a, b, c, d, e, f].
```

Notice that the fact on line 2 in the definition of the `append1/2` predicate is superfluous since the rule on line 3 recurses through the first list only. The `append` predicate is a primitive construct that can be utilized in the definition of additional list manipulation predicates:

```
1 /* rendition of member1 predicate to determine membership of E in L */
2 member1(E,L) :- append1(_, [E|_], L).
3
4 /* predicate to determine if X is a sublist of Y */
5 sublist(X,Y) :- append1(_, X, W), append1(W, _, Y).
6
```

```

7  /* predicate to reverse a list */
8  reverse([], []).
9  reverse([H|T], RL) :- reverse(T, RT), append1(RT, [H], RL).

```

We redefine the `member1` predicate using `append1` (line 2). The revised predicate requires only one rule and declares that `E` is a element of `L` if any list can be appended to any list with `E` as the head resulting in list `L`:

```

?- member1(4, [2,4,6,8]).
true.

```

The `sublist` predicate (line 5) is defined similarly using `append1`. The `reverse` predicate declares that the reverse of an empty list is the empty list (line 8). The rule (line 9) declares that the reverse of a list `[H|T]` is the reverse of list `T`—the tail—appended to the list `[H]` containing only the head `H`. Again, notice the declarative style in which these predicates are defined. We use lists to define graphs and a series of graph predicates in Section 14.7.8. However, before doing so, we discuss arithmetic predicates and the nature of negation in Prolog since those graph predicates involve those two concepts.

14.7.5 Tracing the Resolution Process

Consider the following Prolog program:

```

/* edge(X,Y) declares there is a directed edge from vertex X to Y */

edge(a,b).
edge(b,c).
edge(c,a).

/* path(X,Y,START,PATH) is true when there is a directed path
   from vertex X to Y through the vertices in the list PATH.

   START is the starting list of visited vertices, initially [].

   The third and fourth arguments help maintain
   a running tally of the vertices visited. */
path(X,X,P,P).
path(X,Y,START,FINISH) :- edge(X,Z),
                           \+(member(Z,START)),
                           /* We can go from vertex X to Y through Z
                              only if Z was not already visited in T */
                           append([Z],START,NEWSTART),
                           path(Z,Y,NEWSTART,FINISH).

```

To illustrate the assistance that the `trace/0` predicate provides, consider determining the vertices along the path from vertex `a` to `b`:

```

?- trace.
true.

[trace] ?- path(a,c,[],PATH).
Call: (10) path(a, c, [], _7026) ? creep
Call: (11) edge(a, _7468) ? creep
Exit: (11) edge(a, b) ? creep

```

```

Call: (11) lists:member(b, []) ? creep
Fail: (11) lists:member(b, []) ? creep
Redo: (10) path(a, c, [], _7026) ? creep
Call: (11) lists:append([b], [], _8030) ? creep
Exit: (11) lists:append([b], [], [b]) ? creep
Call: (11) path(b, c, [b], _7026) ? creep
Call: (12) edge(b, _8166) ? creep
Exit: (12) edge(b, c) ? creep
Call: (12) lists:member(c, [b]) ? creep
Fail: (12) lists:member(c, [b]) ? creep
Redo: (11) path(b, c, [b], _7026) ? creep
Call: (12) lists:append([c], [b], _8394) ? creep
Exit: (12) lists:append([c], [b], [c, b]) ? creep
Call: (12) path(c, c, [c, b], _7026) ? creep
Exit: (12) path(c, c, [c, b], [c, b]) ? creep
Exit: (11) path(b, c, [b], [c, b]) ? creep
Exit: (10) path(a, c, [], [c, b]) ? creep
PATH = [c, b] .

[trace]  ?-

```

This trace is produced incrementally as the user presses the <enter> key after each line of the trace to proceed one step deeper into the proof process.

14.7.6 Arithmetic in Prolog

Since comparison operators (e.g., < and >) in other programming languages are predicates (i.e., they return true or false), such predicates are generally used in Prolog in the same manner as they are used in other languages (i.e., using infix notation). The assignment operator in Prolog—in the capacity that an assignment operator can exist in a declarative style of programming—is the `is` predicate in Prolog:

```

1  ?- X is 5-3.
2  X = 2.
3
4  ?- Y is X-1.
5  ERROR: Arguments are not sufficiently instantiated
6  ?-

```

The binding is held only during the satisfaction of the goal that produced the instantiation/binding (lines 1–2). It is lost after the goal is satisfied (lines 4–5). The following are the mathematical Horn clauses in Section 14.5.3 represented in Prolog syntax for Horn clauses:

```

factorial(0,1).
factorial(N,F) :- N > 0, M is N-1, factorial(M,G), F is N*G.

fibonacci(1,0).
fibonacci(2,1).
fibonacci(N,P) :- N > 2, M is N-1, fibonacci(M,G),
                  L is N-2, fibonacci(L,H), P is G+H.

```

The `factorial` predicate binds its second parameter `F` to the factorial of the integer represented by its first parameter `N`:

```

?- factorial(0,F).
F = 1 .

?- factorial(N,1).
N = 0 .

?- factorial(5,F).
F = 120 .

```

14.7.7 Negation as Failure in Prolog

The built-in `\+/1` (not) predicate in Prolog is not a *logical not* operator (i.e., $\neg$), so we must exercise care when using it. The goal `\+(G)` succeeds if goal *G* cannot be proved, not if goal *G* is false. Thus, `\+` is referred to as the *not provable* operator. Thus, the use of `\+/1` can produce counter-intuitive results:

```

1  ?- mother(mary).
2  true.
3
4  ?- mother(M).
5  M = mary.
6
7  ?- \+(mother(M)).
8  false.
9
10 ?- \+(\+(mother(M))).
11 true.
12
13 ?- \+(\+(mother(mary))).
14 true.

```

Assume only the fact `mother(mary)` exists in the database. The predicate `\+(mother(M))` is asserting that “there are no mothers.” The response to the query on line 8 (i.e., `false`) is indicating that “there is a mother,” and not indicating that “there are no mothers.” In attempting to satisfy the goal on line 10, Prolog starts with the innermost term and succeeds with `M = mary`. It then proceeds outward to the next term. Once a term becomes false, the instantiation is released. Thus, on line 11, we do not see a substitution for *x*, which proves the goal on line 10, but we are only given `true`. Consider the following goals:

```

1  ?- \+(M=mary).
2  false.
3
4  ?- M=mary, \+(M=elizabeth).
5  M = mary.
6
7  ?- \+(M=elizabeth), M=mary.
8  false.
9
10 ?- \+(\+(M=elizabeth)), M=mary.
11 M = mary.
12
13 ?- \+(M=elizabeth), \+(M=mary).
14 false.

```

Again, `false` is returned on line 2 without presenting a binding for `M`, which was released. Notice that the goals on lines 4 and 7 are the same—only the order of the subgoals is transposed. While the validity of the goal in logic is not dependent on the order of the subgoals, the order in which those subgoals are pursued is significant in Prolog. On line 5, we see that Prolog instantiated `M` to `mary` to prove the goal on line 4. However, the proof of the goal on line 7 fails at the first subgoal without binding `M` to `mary`.

14.7.8 Graphs

We can model graphs in Prolog using a list whose first element is a list of vertices and whose second element is a list of directed edges, where each edge is a list of two elements—the source and target of the edge. Using this list representation of a graph, a sample graph is `[[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]`. Using the `append/2` and `member/2` predicates (and others not defined here, such as `noduplicateedges/1` and `makeset/2`—see Programming Exercises 14.7.15 and 14.7.16, respectively), we can define the following graph predicates:

```
1 graph([Vertices,Edges]) :-
2     noduplicateedges(Edges),
3     flatten(Edges, X), makeset(X, Y), subset(Y, Vertices).
4
5 vertex([Vset,Eset], Vertex1) :- graph([Vset,Eset]), member(Vertex1, Vset).
6
7 edge([Vset,Eset], Edge) :- graph([Vset,Eset]), member(Edge, Eset).
```

The `graph` predicate (lines 1–3) tests whether a given list represents a valid graph by checking if there are no duplicate edges (line 2) and confirming that the defined edges do not use vertices that are not included in the vertex set (line 3). The `flatten/2` and `subset/2` predicates (line 3) are built into SWI-Prolog. The `vertex` predicate (line 5) accepts a graph and a vertex; it returns `true` if the graph is valid and the vertex is a member of that graph’s vertex set, and `false` otherwise. Similarly, the `edge` predicate (line 7) takes a graph and an edge; it returns `true` if the graph is valid and the edge is a member of that graph’s edge set, and `false` otherwise. The following are example goals:

```
?- graph([a,b,c],[[a,b],[b,c]]).
true .
?- graph([a,b,c],[[a,b],[b,c],[d,a]]).
false.
?- vertex([a,b,c],[[a,b],[b,c]], Vertex).
Vertex = a ;
Vertex = b ;
Vertex = c ;
false.
?- edge([a,b,c],[[a,b],[b,c]], [a,b]).
true .
?- edge([a,b,c],[[a,b],[b,c],[d,a]], [a,b]).
false.
```


These predicates serve as building blocks from which we can construct more graph predicates. For instance, we can check if one graph is a subgraph of another one:

```

8  /* checks if the first graph as [Vset1,Eset1] is a subgraph of
9     the second graph as [Vset2,Eset2] */
10 subgraph([Vset1,Eset1], [Vset2,Eset2]) :-
11     graph([Vset1,Eset1]), graph([Vset2,Eset2]), % inputs are graphs
12     subset(Vset1,Vset2), subset(Eset1,Eset2).

```

The following are subgraph goals :

```

?- subgraph([[a,b,c],[[a,b],[a,c]]], [[a,b,c],[[a,b],[a,c],[b,c]]) .
true .
?- subgraph([[a,b,c],[[a,b],[a,c],[b,c]]], [[a,b,c],[[a,b],[a,c]]) .
false.

```

We can also check whether a graph has a cycle, or a cycle containing a given vertex. A *cycle* is a chain where the start vertex and the end vertex are the same vertex. A *chain* is a path of directed edges through a graph from a source vertex to a target vertex. Using a Prolog list representation, a *chain* is a list of vertices such that there is an edge between each pair of adjacent vertices in the list. Thus, in that representation of a chain, a *cycle* is a chain such that there is an edge from the final vertex in the list to the first vertex in the list. Consider the following predicate to test a graph for the presence of cycles:

```

13 /* checks if Graph has a cycle from Vertex to Vertex */
14 cycle(Graph, Vertex) :- chain(Graph, Vertex, Vertex, _).
15
16 /* checks if graph G has a cycle
17    involving any vertex in the set [V1|Vset] */
18 cyclevertices(G, [V1|Vset]) :- cycle(G, V1); cyclevertices(G, Vset).
19
20 /* checks if graph as [Vset, Eset] has a cycle */
21 cycle([Vset, Eset]) :- cyclevertices([Vset,Eset], Vset).

```

Note that the `cycle/2` predicate uses a `chain/4` predicate (not defined here; see Programming Exercise 14.7.19) that checks for the presence of a path from a start vertex to an end vertex in a graph.

```

?- cycle([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]], a) .
false.

?- cycle([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]], d) .
true .

?- cycle([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]) .
true .

```

An *independent set* is a graph with no edges, or a set of vertices with no edges between them. A *complete graph* is a graph in which each vertex is adjacent to every other vertex. These two classes of graphs are complements of each other. To identify an independent set, we must check if the edge set is empty.

In contrast, a complete graph has no self-edges (i.e., an edge from and to the same vertex), but all other possible edges. A complete directed graph with n vertices has exactly $n \times (n - 1)$ edges. Thus, we can check if a graph is complete by verifying that it is a valid graph, that it has no self-edges, and that the number of edges is described by the prior arithmetic expression. The following are independent and complete predicates for these types of graphs—proper is a helper predicate:

```

22  /* checks if a graph with N vertices has N*(N-1) edges */
23  proper(E, N) :- D is E - N*(N-1), D == 0.
24
25  /* checks if a graph as [Vset, []] is an independent set */
26  independent([Vset, []]) :- graph([Vset], []).
27
28  /* checks if a graph as [Vset,Eset] is a complete graph */
29  complete([Vset,Eset]) :-
30      graph([Vset,Eset], \+(member([V,V], Eset)),
31      length(Vset, NV), length(Eset, NE), proper(NE, NV).
```

The list `length/2` predicate (line 32) is built into SWI-Prolog. The following are goals involving `independent` and `complete`:

```

?- independent([], []).
true.

?- independent([[a,b,c],[a,b],[b,c]]).
false.

?- independent([[a,b,c],[]]).
true.

?- complete([], []).
true.
?- complete([[a,b,c],[a,b],[a,c],[b,a],[b,c],[c,a],[c,b]]).
true.
```

14.7.9 Analogs Between Prolog and an RDBMS

Interaction with the Prolog interpreter is strikingly similar to interacting with a relational database management system (RDBMS) using SQL. Pursuing goals in Prolog is the analog of running queries against a database. Consider the following database of Prolog facts:

```

nineteenthcennovels('Sense and Sensibility','Jane Austen',1811).
nineteenthcennovels('Pride and Prejudice','Jane Austen',1813).
nineteenthcennovels('Notes from Underground','Fyodor Dostoyevsky',1864).
nineteenthcennovels('Crime and Punishment','Fyodor Dostoyevsky',1866).
nineteenthcennovels('The Brothers Karamazov','Fyodor Dostoyevsky',1879-80).

twentiethcennovels('1984','George Orwell',1949).
twentiethcennovels('Wise Blood','Flannery O\u2019Connor',1952).

read('Pride and Prejudice','Jane Austen',1813).
read('Crime and Punishment','Fyodor Dostoyevsky',1866).
```

```

read('1984','George Orwell',1949).

authors('Jane Austen','16 Dec 1775', 'Hampshire, England').
authors('Fyodor Dostoyevsky', '11 Nov 1821', 'Moscow, Russian Empire').

```

Each of the five predicates in this Prolog program (each containing multiple facts) is the analog of a table (or relation) in a database system. The following is a mapping from some common types of queries in SQL to their equivalent goals in Prolog.

Union

```

SELECT * FROM nineteenthcennovals
UNION
SELECT * FROM twentiethcennovals;

?- nineteenthcennovals(TITLE,AUTHOR,YEAR);
| twentiethcennovals(TITLE,AUTHOR,YEAR).
TITLE = 'Sense and Sensibility',
AUTHOR = 'Jane Austen',
YEAR = 1811 ;
TITLE = 'Pride and Prejudice',
AUTHOR = 'Jane Austen',
YEAR = 1813 ;
TITLE = 'Notes from Underground',
AUTHOR = 'Fyodor Dostoyevsky',
YEAR = 1864 ;
TITLE = 'Crime and Punishment',
AUTHOR = 'Fyodor Dostoyevsky',
YEAR = 1866 ;
TITLE = 'The Brothers Karamazov',
AUTHOR = 'Fyodor Dostoyevsky',
YEAR = 1879-80 ;
TITLE = '1984',
AUTHOR = 'George Orwell',
YEAR = 1949 ;
TITLE = 'Wise Blood',
AUTHOR = 'Flannery O'Connor',
YEAR = 1952.

?-

```

While a comma (,) is the conjunction or the and operator in Prolog, a semicolon (;) is the disjunction or the or operator in Prolog.

Intersection

```

SELECT * FROM twentiethcennovals
INTERSECT
SELECT * FROM read;

?- twentiethcennovals(TITLE,AUTHOR,YEAR), read(TITLE,AUTHOR,YEAR).
TITLE = '1984',
AUTHOR = 'George Orwell',
YEAR = 1949 ;
false.

?-

```

Difference

```
SELECT * FROM twentiethcennovels
EXCEPT
SELECT * FROM read;

?- twentiethcennovels(TITLE,AUTHOR,YEAR), \+(read(TITLE,AUTHOR,YEAR)).
TITLE = 'Wise Blood',
AUTHOR = 'Flannery O'Connor',
YEAR = 1952.

?-
```

Projection

```
SELECT title
FROM nineteenthcennovels;

?- nineteenthcennovels(TITLE,_,_).
TITLE = 'Sense and Sensibility' ;
TITLE = 'Pride and Prejudice' ;
TITLE = 'Notes from Underground' ;
TITLE = 'Crime and Punishment' ;
TITLE = 'The Brothers Karamazov'.

?-
```

Selection

```
SELECT *
FROM nineteenthcennovels
WHERE author = "Fyodor Dostoyevsky" and year >= 1865;

?- nineteenthcennovels(TITLE,'Fyodor Dostoyevsky',YEAR), YEAR >= 1865.
TITLE = 'Crime and Punishment',
YEAR = 1866 ;
false.

?-
```

Projection Following Selection

```
SELECT title
FROM nineteenthcennovels
WHERE author = "Fyodor Dostoyevsky";

?- nineteenthcennovels(TITLE,'Fyodor Dostoyevsky',_).
TITLE = 'Notes from Underground' ;
TITLE = 'Crime and Punishment' ;
TITLE = 'The Brothers Karamazov'.

?-
```

Natural Join

```
SELECT *
FROM nineteenthcennovels, authors
WHERE nineteenthcennovels.author = authors.name;

?- nineteenthcennovels(TITLE,AUTHOR,YEAR), authors(AUTHOR,DOB,BIRTHPLACE).
```

```

TITLE = 'Sense and Sensibility',
AUTHOR = 'Jane Austen',
YEAR = 1811,
DOB = '16 Dec 1775',
BIRTHPLACE = 'Hampshire, England' ;
TITLE = 'Pride and Prejudice',
AUTHOR = 'Jane Austen',
YEAR = 1813,
DOB = '16 Dec 1775',
BIRTHPLACE = 'Hampshire, England' ;
TITLE = 'Notes from Underground',
AUTHOR = 'Fyodor Dostoyevsky',
YEAR = 1864,
DOB = '11 Nov 1821',
BIRTHPLACE = 'Moscow, Russian Empire' ;
TITLE = 'Crime and Punishment',
AUTHOR = 'Fyodor Dostoyevsky',
YEAR = 1866,
DOB = '11 Nov 1821',
BIRTHPLACE = 'Moscow, Russian Empire' ;
TITLE = 'The Brothers Karamazov',
AUTHOR = 'Fyodor Dostoyevsky',
YEAR = 1879-80,
DOB = '11 Nov 1821',
BIRTHPLACE = 'Moscow, Russian Empire'.

?-

```

Theta-Join

```

SELECT *
FROM nineteenthcennovels, authors
WHERE nineteenthcennovels.author = authors.name and year <= 1850;

?- nineteenthcennovels(TITLE,AUTHOR,YEAR),
   | authors(AUTHOR,DOB,BIRTHPLACE), YEAR =< 1850.
TITLE = 'Sense and Sensibility',
AUTHOR = 'Jane Austen',
YEAR = 1811,
DOB = '16 Dec 1775',
BIRTHPLACE = 'Hampshire, England' ;
TITLE = 'Pride and Prejudice',
AUTHOR = 'Jane Austen',
YEAR = 1813,
DOB = '16 Dec 1775',
BIRTHPLACE = 'Hampshire, England' ;
TITLE = 'The Brothers Karamazov',
AUTHOR = 'Fyodor Dostoyevsky',
YEAR = 1879-80,
DOB = '11 Nov 1821',
BIRTHPLACE = 'Moscow, Russian Empire'.

?-

```

Adding the preceding queries in the form of rules creates what are called *views* in database terminology, where the head of the headed Horn clause is the name of the view:

```

% Union:
novels(TITLE,AUTHOR,YEAR) :-
    nineteenthcennovels(TITLE,AUTHOR,YEAR);

```

```

    twentiethcennovels(TITLE,AUTHOR, YEAR) .

% Intersection:
readtwentiethcennovels(TITLE,AUTHOR, YEAR) :-
    twentiethcennovels(TITLE,AUTHOR, YEAR), read(TITLE,AUTHOR, YEAR) .

% Difference:
unread(TITLE,AUTHOR, YEAR) :-
    twentiethcennovels(TITLE,AUTHOR, YEAR), \+(read(TITLE,AUTHOR, YEAR)) .

% Projection:
nineteenthcennoveltiles(TITLE) :- nineteenthcennovels(TITLE,_,_) .

% Selection:
latenineteenthcennovelsbyFD(TITLE, YEAR) :-
    nineteenthcennovels(TITLE,'Fyodor Dostoyevsky',YEAR), YEAR >= 1865.

% Projection following selection:
titlesnineteenthcennovelsbyFD :-
    nineteenthcennovels(TITLE,'Fyodor Dostoyevsky',_) .

% Natural join:
nineteenthcennovelsauthors :-
    nineteenthcennovels(TITLE,AUTHOR, YEAR), authors(AUTHOR,DOB,BIRTHPLACE) .

% Theta-join:
earlynineteenthcennovelsauthors :-
    nineteenthcennovels(TITLE,AUTHOR, YEAR),
    authors(AUTHOR,DOB,BIRTHPLACE), YEAR <= 1850.

```

Table 14.15 presents analogs between Relational Database Management Systems and Prolog. Datalog is a non-Turing-complete subset of Prolog for use with *deductive databases* or *rule-based databases*.

Conceptual Exercises for Sections 14.6–14.7

Exercise 14.7.1 Prolog is a *declarative* programming language. What does this mean?

Exercise 14.7.2 Give an example of a language supporting *declarative/logic programming* other than Prolog.

Exercise 14.7.3 Explain why the `\+ / 1` Prolog predicate is not a true logical NOT operator. Provide an example to support your explanation.

Exercise 14.7.4 Does Prolog use *short-circuit evaluation*? Provide a Prolog goal (and the response the interpreter provides in evaluating it) to unambiguously support your answer. Note that the result of the goal `?- 3 = 4, 3 = 3.` does not prove or disprove the use of short-circuit evaluation in Prolog.

Exercise 14.7.5 Since the depth-first search strategy is problematic for reasons demonstrated in Section 14.7.1, why does Prolog use depth-first search? Why is breadth-first search not used instead?

RDBMS	Prolog
relation	predicate
attribute	argument
tuple	ground fact
table	extensional definition of predicate (i.e., set of facts)
view	intensional definition of predicate (i.e., a rule)
variable query evaluation	fixed query evaluation (i.e., depth-first search)
forward chaining	backward chaining
table/set at a time	tuple at a time

Table 14.15 Analogs Between a Relational Database Management System (RDBMS) and Prolog

Exercise 14.7.6 In Section 14.7.1, we saw that left-recursion on the left-hand side of a rule causes a stack overflow. Why is this not the case in the `reverse` predicate in Section 14.7.4?

Exercise 14.7.7 Consider the following Prolog predicate `a :- b, c, d.`, where *b*, *c*, and *d* can represent any subgoals. Prolog will *try* to satisfy subgoals *b*, *c*, and *d*, *in that order*. However, might Prolog satisfy subgoal *c* before it satisfies subgoal *b*? Explain.

Exercise 14.7.8 Reconsider the `factorial` predicate presented in Section 14.7.6. Explain why the goal `factorial(N, 120)` results in an error.

Exercise 14.7.9 Consider the following Prolog goal and its result:

```
?- X=0, \+(X=1).
X = 0.
```

Explain why the result of the following Prolog goal does not bind *x* to 1:

```
?- \+(X=0), X=1.
false.
```

Exercise 14.7.10 Which approach to resolution is more complex: *backward chaining* or *forward chaining*? Explain with reasons.

Programming Exercises for Sections 14.6–14.7

Exercise 14.7.11 Reconsider the `append1/3` predicate in Section 14.7.4:

```
append1([], L, L).
append1(L, [], L).
append1([X|L1], L2, [X|L12]) :- append1(L1, L2, L12).
```

This predicate has a bug—it produces duplicate solutions (lines 4–5, 8–9, 12–13, 14–15, and 16–17):

```

1  ?- append1(X,Y, [dostoyevsky,orwell,oconnor]).
2  X = [],
3  Y = [dostoyevsky, orwell, oconnor] ;
4  X = [dostoyevsky, orwell, oconnor],
5  Y = [] ;
6  X = [dostoyevsky],
7  Y = [orwell, oconnor] ;
8  X = [dostoyevsky, orwell, oconnor],
9  Y = [] ;
10 X = [dostoyevsky, orwell],
11 Y = [oconnor] ;
12 X = [dostoyevsky, orwell, oconnor],
13 Y = [] ;
14 X = [dostoyevsky, orwell, oconnor],
15 Y = [] ;
16 X = [dostoyevsky, orwell, oconnor],
17 Y = [] ;
18 false.
19
20 ?-

```

This bug propagates when `append1` is used as a primitive construct to define other (list) predicates. Modify the definition of `append1` to eliminate this bug:

```

?- append1(X,Y, [dostoyevsky,orwell,oconnor]).
X = [],
Y = [dostoyevsky, orwell, oconnor] ;
X = [dostoyevsky],
Y = [orwell, oconnor] ;
X = [dostoyevsky, orwell],
Y = [oconnor] ;
X = [dostoyevsky, orwell, oconnor],
Y = [] ;
false.

?-

```

Exercise 14.7.12 Define a Prolog predicate `reverse(L,R)` that succeeds when the list `R` represents the list `L` with its elements reversed, and fails otherwise. Your predicate must not produce duplicate results. Use no auxiliary predicates, except for `append/3`.

Exercise 14.7.13 Define a Prolog predicate `sum` that binds its second argument `S` to the sum of the integers from 1 up to and including the integer represented by its first parameter `N`.

Examples:

```

?- sum(N,0).
N = 0 .
?- sum(0,S).
S = 0 .
?- sum(4,S).
S = 10 .
?- sum(4,8).
false.

```



```
?- sum(5, Y) .
Y = 15 .
?- sum(500, Y) .
Y = 125250 .
?- sum(-100, Y) .
false.
```

Exercise 14.7.14 Consider the following logical description for the *Euclidean algorithm* to compute the *greatest common divisor* (gcd) of two positive integers u and v :

The gcd of u and 0 is u .

The gcd of u and v , if v is not 0, is the same as the gcd of v and the remainder of dividing v into u .

Define a Prolog predicate `gcd(U, V, W)` that succeeds if W is the greatest common divisor of U and V , and fails otherwise.

Exercise 14.7.15 Reconsider the list representation of an edge in a graph described in Section 14.7.8. Define a Prolog predicate `noduplicateedges/1` that accepts a list of edges and that returns `true` if the list of edges is a set (i.e., has no duplicates) and `false` otherwise. Use no auxiliary predicates, except for `not/1` and `member/2`.

Examples:

```
?- noduplicateedges([[a,b],[b,c],[d,a]]).
true.
?- noduplicateedges([[a,b],[b,c],[d,a],[b,c]]).
false.
```

Exercise 14.7.16 Define a Prolog predicate `makeset/2` that accepts a list and removes any repeating elements—producing a set. The result is returned in the second list parameter. Use no auxiliary predicates, except for `not/1` and `member/2`.

Examples:

```
?- makeset([], []).
true.
?- makeset([a,b,c], SET).
SET = [a, b, c].
?- makeset([a,b,c,a], SET).
SET = [b, c, a].
?- makeset([a,b,c,a,b], SET).
SET = [c, a, b].
```

Exercise 14.7.17 Using only `append`, define a Prolog predicate `adjacent` that accepts only three arguments and that succeeds if its first two arguments are adjacent in its third list argument and fails otherwise.

Examples:

```
?- adjacent(1,2,[1,2,3]).
true.
?- adjacent(1,2,[3,1,2]).
true.
?- adjacent(1,2,[1,3,2]).
false.
?- adjacent(2,1,[1,2,3]).
true.
?- adjacent(2,3,[1,2,3]).
true.
?- adjacent(3,1,[1,2,3]).
false.
```

Exercise 14.7.18 Modify your solution to Programming Exercise 14.7.17 so that the list is circular.

Examples:

```
?- adjacent(1,4,[1,2,3,4]).
true.
?- adjacent(4,1,[1,2,3,4]).
true.
?- adjacent(2,4,[1,2,3,4]).
false.
```

Exercise 14.7.19 Reconsider the description of a *chain* in a graph described in Section 14.7.8. Define a Prolog predicate `chain/4` that returns `true` if the graph represented by its first parameter contains a chain (represented by its fourth parameter) from the source vertex and target vertex represented by its second and third parameters, respectively, and `false` otherwise.

Examples:

```
?- chain([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]], b, d, CHAIN).
CHAIN = [b, c, d] .
?- chain([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]], a, d, CHAIN).
CHAIN = [a, b, c, d] .
?- chain([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]], a, a, CHAIN).
false.
?- chain([[a,b,c,d],[[a,b],[b,c],[c,d],[d,b]]], d, d, CHAIN).
CHAIN = [d, b, c, d] .
```

Exercise 14.7.20 Define a Prolog predicate `sort` that accepts two arguments, sorts its first integer list argument, and returns the result in its second integer list argument.

Examples:

```
?- sort([1],S).
S = [1] .
?- sort([1,2],S).
S = [1, 2] .
```

```
?- sort([5,4,3,2,1],S).
S = [1, 2, 3, 4, 5] .
```

The Prolog *less than* predicate is <:

```
?- 3 < 4.
true.

?- 4 < 3.
false.

?- 
```

Exercise 14.7.21 Define a Prolog predicate `last` that accepts only two arguments and that succeeds if its first argument is the last element of its second list argument and fails otherwise.

Examples:

```
?- last(X,[1,2,3]).
X = 3
?- last(4,[1,2,3]).
false.
```

Exercise 14.7.22 Define a Prolog `nand/3` predicate. The following table models a *nand* gate:

p	q	$p \text{ NAND } q$
0	0	1
1	0	1
0	1	1
1	1	0

Exercise 14.7.23 A *multiplexer* is a device that connects one of many inputs to a single output through one or more selector lines, which collectively model the numeric position of the selected input as a binary number. Define a Prolog predicate that acts as a 4-input 2-bit multiplexer.

Examples:

```
?- mux("1", "2", "3", "4", 0, 1, Output).
Output = "2".
?- mux("1", "2", "3", "4", 1, 1, Output).
Output = "4".
```

Exercise 14.7.24 Define a Prolog predicate `validdate(Month,Day,Year)` that accepts only three arguments, which represent a month, day, and year, in that order. The predicate succeeds if these arguments represent a valid date in the Gregorian calendar, and fails otherwise. For example, `date(oct,15,1996)` is valid, but `date(jun,31,1921)` is not. You must account for both different

numbers of days in different months and February in leap years. A leap year is a year that is divisible by 400, or divisible by 4 but not also by 100. Therefore, 2000, 2012, 2016, and 2020 were leap years, while 1800 and 1900 were not. Your solution must not use more than three user-defined predicates or exceed 20 lines of code.

Examples:

```
?- validate(feb,29,2000).
true.
?- validate(feb,30,2000).
false.
?- validate(feb,29,2004).
true.
?- validate(feb,29,1900).
false.
?- validate(may,16,2007).
true.
?- validate(jun,31,2007).
false.
?- validate(apr,-10,3).
false.
?- validate(apr,32,3).
false.
?- validate(apr,30,-100).
false.
?- validate(apr,30,0).
true.
?- validate(jul,0,0).
false.
?- validate(jul,1,0).
true.
?- validate(fun,15,2020).
false.
```

14.8 Imparting More Control in Prolog: Cut

The `!` operator is the *cut* predicate in Prolog. The cut predicate gives the programmer a measure of control over the search process—an impurity—so it should be used with caution. (The use of the cut predicate in Prolog is, in spirit, not unlike the use of `goto` in C or `call/cc` in Scheme in its manipulation of normal program control.) Cut is a predicate that always evaluates to true:

```
?- !.
true.
```

However, the cut predicate has a side effect: It both freezes parts of solutions already found and prevents multiple/alternative solutions from being produced/considered. In this way, it prunes branches in the resolution search tree and reduces the number of branches in the search tree considered.

The cut predicate can appear in a Prolog program in the body of a rule or in a goal (as a subgoal). In either case, when the cut is encountered, it freezes (i.e., fixes) any prior instantiations of free variables bound during unification for the remainder of the search and prevents backtracking. As a consequence, alternative

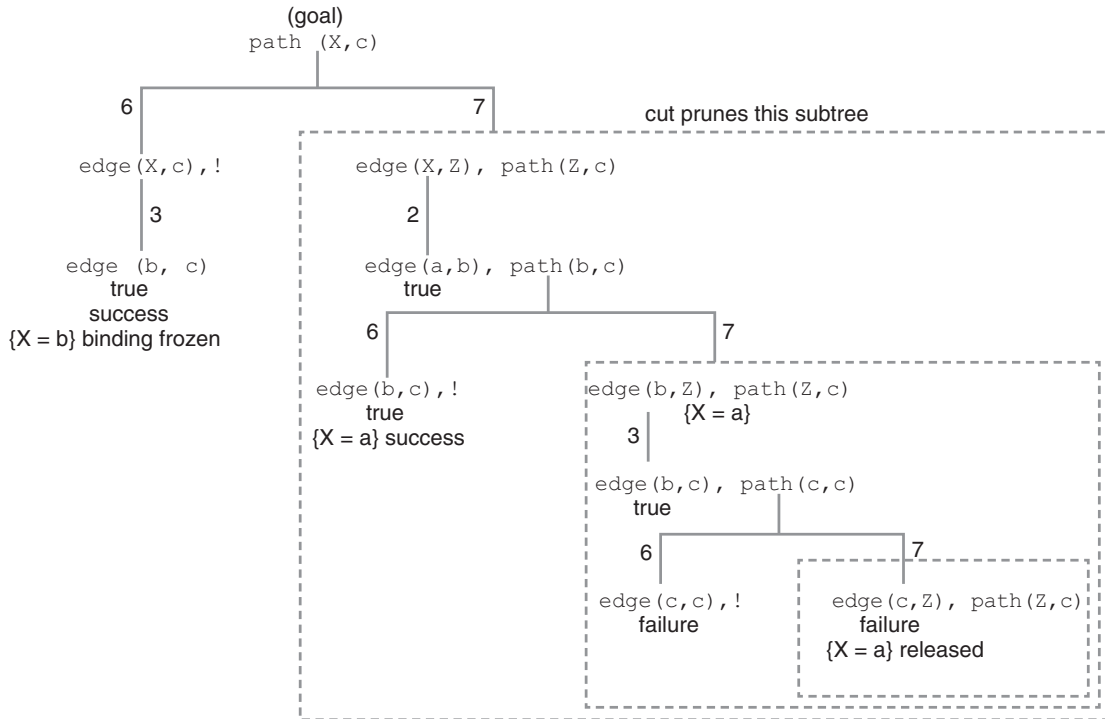


Figure 14.5 The branch (encompassed by a dotted box) of the resolution search tree for the `path(X, c)` goal that the cut operator removes in the first `path` predicate.

instantiations, which might lead to success, are not tried. Reconsider the `path` predicate from Section 14.7.1, but with a cut included (line 6):

```

1  /* edge(X,Y) declares there is a directed edge from vertex X to Y */
2  edge(a,b) .
3  edge(b,c) .
4
5  /* path(X,Y) declares there is a path from vertex X to Y */
6  path(X,Y) :- edge(X,Y), !.
7  path(X,Y) :- edge(X,Z), path(Z,Y) .

1  /* edge(X,Y) declares there is a directed edge from vertex X to Y */
2  edge(a,b) :- writeln('Finished evaluating edge(a,b) fact.').
3  edge(b,c) :- writeln('Finished evaluating edge(b,c) fact.').
4
5  /* path(X,Y) declares there is a path from vertex X to Y */
6  path(X,Y) :- writeln('Evaluate 1st path rule.'), edge(X,Y), !.
7  path(X,Y) :- writeln('Evaluate 2nd path rule.'), edge(X,Z), path(Z,Y) .

```

Output statements have been added to the body of the rules to assist in tracing the search. Consider the goal `path(X, c)`:

```

?- path(X,c).
Evaluate 1st path rule.
Finished evaluating edge(b,c) fact.
X = b.
?-

```

The search tree for this goal is shown in Figure 14.5. The edge labels in the figure denote the line number from the Prolog program involved in the match from sub-goal source to antecedent target. The left subtree corresponds to the rule on line 6, whose antecedent contains a cut. Here, the cut freezes the binding of *X* to *b*, so that the right subtree is not considered. Once a cut has been encountered (i.e., evaluated to true), during backtracking the search of the subtrees of the parent node of the node containing the cut stops, and the search resumes with the parent node of the parent, if present. As a result, the cut prunes from the search tree all siblings to the right of the node with the cut. Consider the following modification to the path predicate:

```

5  /* path(X,Y) declares there is a path from vertex X to Y */
6  path(X,Y) :- writeln('Evaluate 1st path rule.'), edge(X,Z), path(Z,Y), !.
7  path(X,Y) :- writeln('Evaluate 2nd path rule.'), edge(X,Y).

```

The two rules constituting the prior path predicate are transposed and the cut is shifted from the last predicate of the body of one of the rules to the last predicate of the body of the other rule. Reconsider the goal `path(X,c)`:

```

?- path(X,c).
Evaluate 1st path rule.
Finished evaluating edge(a,b) fact.
Evaluate 1st path rule.
Finished evaluating edge(b,c) fact.
Evaluate 1st path rule.
Evaluate 2nd path rule.
Evaluate 2nd path rule.
Finished evaluating edge(b,c) fact.
X = a.
?-

```

The search tree for this goal is presented in Figure 14.6. Notice that the output statements trace the depth-first search of the resolution tree. In this example, the failure in the left subtree occurs *before* the cut is evaluated, so the solution *X=a* is found. Once the cut is evaluated (after *X* is bound to *a*), the solution *X=a* is frozen and the right subtree is never considered. Now consider one last modification to the path predicate:

```

5  /* path(X,Y) declares there is a path from vertex X to Y */
6  path(X,Y) :- writeln('Evaluate 1st path rule.'), edge(X,Z), !, path(Z,Y).
7  path(X,Y) :- writeln('Evaluate 2nd path rule.'), edge(X,Y).

```

The cut predicate is shifted one term to the left on line 6. Reconsider the goal `path(X,c)`:

```

?- path(X,c).
Evaluate 1st path rule.

```

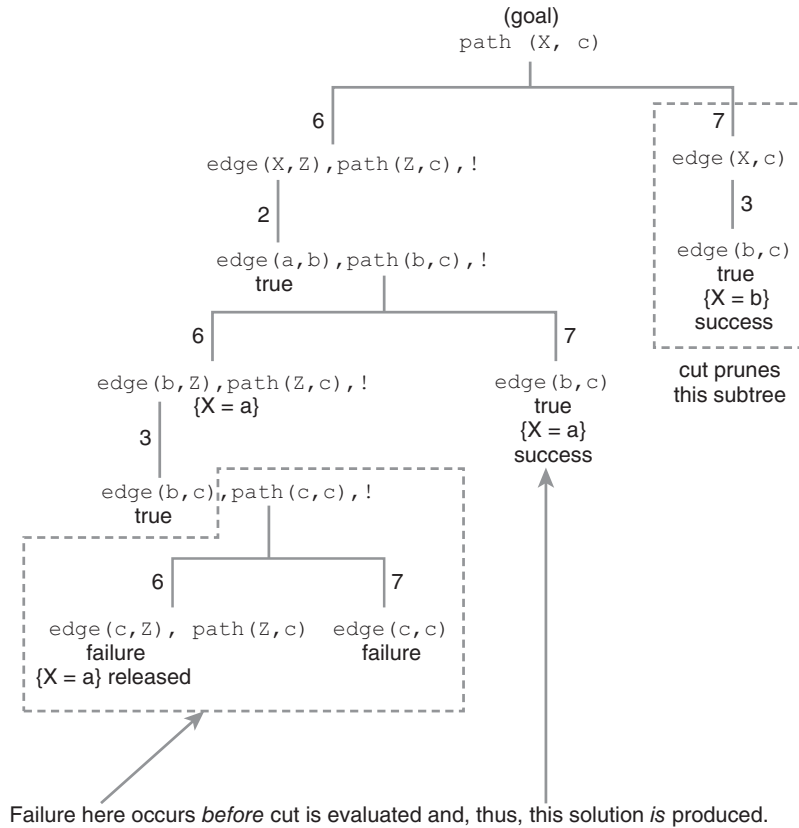


Figure 14.6 The branch (encompassed by a dotted box) of the resolution search tree for the `path(X, c)` goal that the cut operator removes in the second path predicate.

```

Finished evaluating edge(a,b) fact.
Evaluate 1st path rule.
Finished evaluating edge(b,c) fact.
Evaluate 1st path rule.
Evaluate 2nd path rule.
false.

```

The search tree for the goal `path(X, c)` is presented in Figure 14.7. Unlike in the prior example, here the failure in the left subtree occurs *after* the cut is evaluated, so even the solution `X=a` is not found. Now no solutions are returned.

In the three preceding examples, the cut predicate is used in the body of a rule. However, the cut predicate can also be used (as a subgoal) in a goal. Consider the following database:

```

1 author(dostoyevsky).
2 author(orwell).
3 author(oconnor).

```

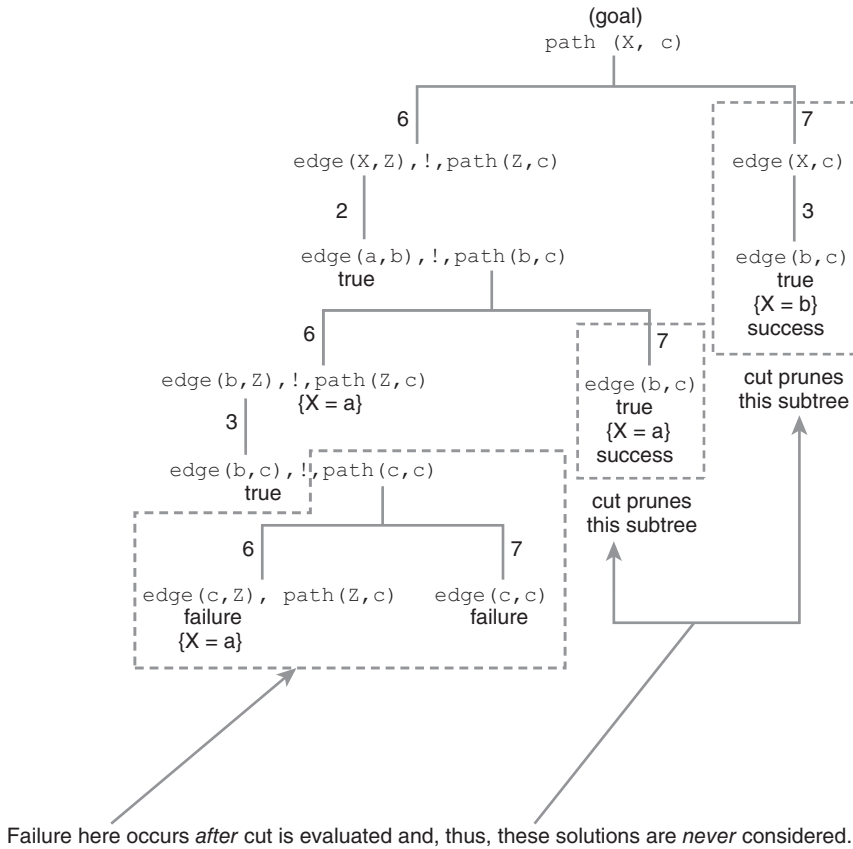


Figure 14.7 The branch (encompassed by a dotted box) of the resolution search tree for the `path(X, c)` goal that the cut operator removes in the third `path` predicate.

We use the cut predicate in the goal on line 23 in the following transcript to prevent consideration of alternative instantiations of `X` by freezing the first instantiation (i.e., `X=dostoyevsky`):

```

1  ?- author(AUTHOR).
2  AUTHOR = dostoyevsky ;
3  AUTHOR = orwell ;
4  AUTHOR = oconnor.
5
6  ?- author(X), author(Y).
7  X = Y, Y = dostoyevsky ;
8  X = dostoyevsky,
9  Y = orwell ;
10 X = dostoyevsky,
11 Y = oconnor ;
12 X = orwell,
13 Y = dostoyevsky ;
14 X = Y, Y = orwell ;
15 X = orwell,
16 Y = oconnor ;
17 X = oconnor,
18 Y = dostoyevsky ;
19 X = oconnor,
20 Y = orwell ;
21 X = Y, Y = oconnor.
22
23 ?- author(X), !, author(Y).
24 X = Y, Y = dostoyevsky ;
25 X = dostoyevsky,
26 Y = orwell ;
27 X = dostoyevsky,
28 Y = oconnor.
```


Notice how the cut in the goal on line 23 froze the instantiation of `X` to `dostoyevsky`, so that backtracking pursued only alternative instantiations of `Y` (lines 26 and 28) to prove the goal. Consider replacing the second fact (line 2) with the rule `author(orwell) :- !:`

```

1  ?- author(AUTHOR).
2  AUTHOR = dostoyevsky ;
3  AUTHOR = orwell.
4
5  ?- author(X), author(Y).
6  X = Y, Y = dostoyevsky ;
7  X = dostoyevsky,
8  Y = orwell ;
9  X = orwell,
10 Y = dostoyevsky ;
11 X = Y, Y = orwell.
12
13 ?- author(X), !, author(Y).
14 X = Y, Y = dostoyevsky ;
15 X = dostoyevsky,
16 Y = orwell.
```

The cut in the rule on line 2 affects the results of the goals on lines 1, 5, and 13. In particular, once a variable is bound to `orwell`, no additional instantiations are considered. The cut freezes the instantiations and prevents backtracking to the left of the cut predicate in a line of code, while alternative instantiations are considered to the right of the cut predicate:

instantiations frozen and, thus, backtracking prevented, to left of cut

$$\boxed{T_1, T_2, \dots, T_m,}$$

!,

$$\boxed{T_{m+1}, \dots, T_n-1, T_n.}$$

alternative instantiations and backtracking occur to right of cut

Lastly, reconsider the `member1/2` predicate in Section 14.7.3:

```

member1(E, [E|_]).
member1(E, [_|T]) :- member1(E, T).
```

This definition of `member1/2` returns `true` as many times as there are occurrences of the element in the input list:

```

?- member1(oconnor, [dostoyevsky, orwell, oconnor, austen, oconnor]).
true ;
true ;
false.

?-
```

Using a cut we can prevent multiple solutions from being produced such that `member1/2` returns `true` only once, even if the element occurs more than once in the input list:

```

% The cut here prevents member1 from finding all occurrences.
member1(E, [E|_]) :- !.
member1(E, [_|T]) :- member1(E, T).

?- member1(oconnor, [dostoyevsky, orwell, oconnor, austen, oconnor]).
true
-?
```

However, this modification prevents multiple solutions universally. As a consequence, we can no longer successfully query for all elements of the input list (as we did at the end of Section 14.7.3):

```
?- member1(AUTHOR, [dostoyevsky, orwell, oconnor, austen, oconnor]).
AUTHOR = dostoyevsky.

?-
```

Conceptual Exercises for Section 14.8

Exercise 14.8.1 Consider the following two Prolog programs, each of which is a variation of the `path` program from Section 14.8, where the `cut` predicate is in a different position. Predict the output and draw the search tree for the goal `path(X, c)` using each of the following two programs:

(a)

```
/* edge(X,Y) declares there is a directed edge from vertex X to Y */
edge(a,b).
edge(b,c).

/* path(X,Y) declares there is a path from vertex X to Y */
path(X,Y) :- !, edge(X,Z), path(Z,Y).
path(X,Y) :- edge(X,Y).
```

(b)

```
/* edge(X,Y) declares there is a directed edge from vertex X to Y */
edge(a,b).
edge(b,c).

/* path(X,Y) declares there is a path from vertex X to Y */
path(X,Y) :- edge(X,Z), path(Z,Y).
path(X,Y) :- !, edge(X,Y).
```

Exercise 14.8.2 Consider the following Prolog database:

```
1 author(dostoyevsky).
2 author(orwell).
3 author(oconnor).
```

For each of the following goals, draw the search tree and indicate which parts of it the cut prunes, as done in Figures 14.5–14.7:

(a) `!, author(X), author(Y).`

(b) `author(X), !, author(Y).`

(c) `author(X), author(Y), !.`

(d) `!, author(X), author(Y).` if rule 1 above is `author(dostoyevsky) :- !.`

(e) `author(X), !, author(Y) . if rule 2 above is author(orwell) :- ! .`

(f) `author(X), author(Y), ! . if rule 3 above is author(oconnor) :- ! .`

Exercise 14.8.3 John Robinson’s development of the concept of *unification* is a seminal contribution to automatic theorem proving and logic programming. During resolution, Prolog binds values to variables through unification. However, most implementations of Prolog, including SWI-Prolog, do not check if a candidate clause contains any instances of the variable being matched—a test called the *occurs-check*. For instance, the terms `X` and `philosopher(X)` can never be unified; there is no substitution for `X` that could ever make the two terms match. Therefore, an implementation of unification that does not perform the occurs-check is optimistic and, ultimately, incomplete. For what reasons might implementers decide not to perform the occurs-check?

Programming Exercises for Section 14.8

Exercise 14.8.4 When the `complete` predicate in Section 14.7.8 succeeds, it repeatedly returns `true`:

```
?- complete([[a,b,c],[[a,b],[a,c],[b,a],[b,c],[c,a],[c,b]]) .
true ;
true ;
...
```

Modify the `complete` predicate using a cut to rectify this problem.

Exercise 14.8.5 Consider the following Prolog implementation of the bubblesort algorithm:

```
/* List L bubblesorted is list SL */
bubblesort(L,SL) :- appendl(M,[A,B|N],L),
                  A > B,
                  appendl(M,[B,A|N],S),
                  bubblesort(S,SL) .

bubblesort(L,L) .
```

Now consider the following goal:

```
?- bubblesort([9,8,7,6,5,4,3,2,1],SL) .
SL = [1, 2, 3, 4, 5, 6, 7, 8, 9] ;
SL = [2, 1, 3, 4, 5, 6, 7, 8, 9] ;
SL = [2, 3, 1, 4, 5, 6, 7, 8, 9] ;
SL = [2, 3, 4, 1, 5, 6, 7, 8, 9]
...
...
```

As can be seen, after producing the sorted list (line 2), the predicate produced multiple spurious solutions. Modify the `bubblesort` predicate to ensure that it does not return any additional results after it produces the first result—which is always the correct one:

```
?- bubblesort([9,8,7,6,5,4,3,2,1],SL).
SL = [1, 2, 3, 4, 5, 6, 7, 8, 9].

?-
```

Exercise 14.8.6 Define a Prolog predicate `squarelistofints/2` that returns `true` if the list of integers represented by its second parameter are the squares of the list of integers represented by its first parameter, and `false` otherwise. If an element of the first list parameter is not an integer, insert it into the second list parameter in the same position. The built-in Prolog predicate `integer/1` succeeds if its parameter is an integer and fails otherwise.

Examples:

```
?- squarelistofints([1,2,3,4,5,6],SQUARES).
SQUARES = [1, 4, 9, 16, 25, 36].
?- squarelistofints([1,2,3.3,4,5,6],SQUARES).
SQUARES = [1, 4, 3.3, 16, 25, 36].
?- squarelistofints([1,2,"pas un entier",4,5,6],SQUARES).
SQUARES = [1, 4, "pas un entier", 16, 25, 36].
```

Exercise 14.8.7 Implement the Towers of Hanoi algorithm in Prolog. *Towers of Hanoi* is a mathematical puzzle using three pegs, where the objective is to shift a stack of discs of different sizes from one peg to another peg using the third peg as an intermediary. At the start, the discs are stacked along one peg such that the largest disc is at the bottom and the remaining discs are progressively smaller, with the smallest at the top. Only one disc may be moved at a time—the uppermost disc on any peg, and a disc may not be placed on a disc that is smaller than it. The following is a sketch of an implementation of the solution to the Towers of Hanoi puzzle in Prolog:

```
1  /* Move N disks from peg A to peg B using peg C as intermediary. */
2  towers(0,_,_,_) :-
3  towers(N,A,B,C) :-
4
5
6
7  move(A,B) :- write('Move a disc from peg '),
8               write(A),
9               write(' to peg '),
10              write(B),
11              writeln('.').
```

Complete this program. Specifically, define the bodies of the two rules constituting the `towers` predicate. *Hint*: The body of the second rule requires four terms (lines 3–6).

Example (with three discs):

```
?- towers(3,"A","B","C").
Move a disc from peg A to peg B.
Move a disc from peg A to peg C.
Move a disc from peg B to peg C.
```

```

Move a disc from peg A to peg B.
Move a disc from peg C to peg A.
Move a disc from peg C to peg B.
Move a disc from peg A to peg B.
true.

```

The solution to the Towers of Hanoi puzzle is an *exponential-time algorithm* that requires $2^n - 1$ moves, where n is the number of discs. Thus, if we ran the program with an input size of 100 discs on a computer that performs 1 billion operations per second, the program would run for approximately 4×10^{11} centuries!

Exercise 14.8.8 Define the `\=` predicate in Prolog using only the `!`, `fail`, and `=` predicates. Name the predicate `donotunify`.

Exercise 14.8.9 Define the `\==` predicate in Prolog using only the `!`, `fail`, and `==` predicates. Name the predicate `notequal`.

Exercise 14.8.10 Consider the following Prolog database:

```

parent(olimpia,lucia).
parent(olimpia,olga).
sibling(X,Y) :- parent(M,X), parent(M,Y).

```

Prolog responds to the goal `sibling(X,Y)` with

```

1  ?- sibling(X,Y).
2  X = Y, Y = lucia ;
3  X = lucia,
4  Y = olga ;
5  X = olga,
6  Y = lucia ;
7  X = Y, Y = olga.
8
9  ?-

```

Thus, Prolog thinks that `lucia` is a sibling of herself (line 1) and that `olga` is a sibling of herself (line 7). Modify the `sibling` rule so that Prolog does not produce pairs of siblings with the same elements.

Exercise 14.8.11 The following is the definition of the `member1/2` Prolog predicate presented in Section 14.7.3:

```

member1(E, [E|_]).
member1(E, [_|T]) :- member1(E,T).

```

The `member1(E,L)` predicate returns true if the element represented by `E` is a member of list `L` and fails otherwise.

(a) Give the response Prolog produces for the goal

```
member1(E, [lucia, leisel, linda]).
```

(b) Give the response Prolog produces for the goal

```
\+(\+(\member1(E, [lucia, leisel, linda]))).
```

(c) Define a Prolog predicate `notamember(E, L)` that returns true if `E` is not a member of list `L` and fails otherwise.

Exercise 14.8.12 Define a Prolog predicate `emptyintersection/2` that succeeds if the set intersection of two given list arguments, representing sets, is empty and fails otherwise. Do not use any built-in set predicates.

Exercise 14.8.13 The following is the triple predicate, which triples a list (i.e., given `[3]`, it produces `[3, 3, 3]`):

```
triple(L, LLL) :- append(L, L, LL), append(LL, L, LLL).
```

For instance, if `L=[1, 2, 3]`, `triple` produces `[1, 2, 3, 1, 2, 3, 1, 2, 3]` in `LLL`. Rewrite the `triple` predicate so that for `X=[1, 2, 3]`, `LLL` is set equal to `[1, 1, 1, 2, 2, 2, 3, 3, 3]`. The revised `triple` predicate must not produce duplicate results.

Exercise 14.8.14 Implement a “negation as failure” `not1/1` predicate in Prolog. *Hint:* The solution requires a cut.

14.9 Analysis of Prolog

14.9.1 Prolog Vis-à-Vis Predicate Calculus

The following are a set of interrelated impurities in Prolog with respect to predicate calculus:

- **The Capability to Impart Control:** To conduct pure declarative programming, the programmer should be neither permitted nor required to affect the control flow for program success. However, as a practical matter, sometimes a Prolog programmer must be aware of, if not affect, program control, as a consequence of a depth-first search strategy. Unlike declarative programming in Prolog, using a declarative style of programming in the Mercury programming language is considered more pure because Mercury does not support a cut operator or other control facilities intended to circumvent or direct the system’s built-in search strategy (Somogyi, Henderson, and Conway 1996). Also, Mercury programs are fast—they typically execute faster than the equivalent Prolog programs.
- **The Closed-World Assumption:** Another impure feature of Prolog is its *closed-world assumption*—it can reason only from the facts and rules given to it in a database. If Prolog cannot satisfy a given goal using the given database, it assumes the goal is false. Prolog cannot, however, *prove* a goal to be false.

Moreover, there is no mechanism in Prolog by which to assert propositions as false (e.g., $\neg P$). As a result, the goal $\neg (P)$ can succeed simply because Prolog cannot prove P to be true, and not because P is indeed false. For instance, the success of the goal $\neg (\text{member}(4, [1, 2]))$ does not prove

that 4 is not a member of the list `[1, 2]`; it just means that the system failed to prove that 4 is not a member of the list.

- **Limited Expressivity of Horn Clauses:** Horn clauses are not expressive enough to capture any arbitrary proposition in predicate calculus. For instance, a proposition in clausal form with a disjunction of more than one non-negated term cannot be expressed as a Horn clause. As an example, the penultimate proposition in clausal form presented in Section 14.5.1, represented here, contains a disjunction of two non-negated terms:

$$\begin{array}{lcl} \text{siblings}(\text{christina}, \text{maria}) & \vee & \text{cousins}(\text{christina}, \text{maria}) \subset \\ \text{grandfather}(\text{virgil}, \text{christina}) & \wedge & \text{grandfather}(\text{virgil}, \text{maria}) \end{array}$$

The Horn clauses that model this proposition are

$$\begin{array}{lcl} \text{siblings}(\text{christina}, \text{maria}) & \subset & \text{grandfather}(\text{virgil}, \text{christina}) \wedge \text{grandfather}(\text{virgil}, \text{maria}) \wedge \neg \text{cousins}(\text{christina}, \text{maria}) \\ \text{cousins}(\text{christina}, \text{maria}) & \subset & \text{grandfather}(\text{virgil}, \text{christina}) \wedge \text{grandfather}(\text{virgil}, \text{maria}) \wedge \neg \text{siblings}(\text{christina}, \text{maria}) \end{array}$$

These Horn clauses can be approximated as follows in Prolog:

```
siblings(christina,maria) :- grandfather(virgil,christina),
                             grandfather(virgil,maria),
                             \+(cousins(christina,maria)).

cousins(christina,maria) :- grandfather(virgil,christina),
                             grandfather(virgil,maria),
                             \+(siblings(christina,maria)).
```

Since there is a difference in the semantics of the `\+/1` (not) predicate in Prolog (i.e., inability to prove) vis-à-vis the negation operator in logic (i.e., falsehood), these rules are an inaccurate representation of the preceding Horn clauses.

It is also a challenge to represent a proposition involving an existentially quantified conjunction of two non-negated terms in clausal form:

$$\exists X.(\text{country}(X) \wedge \text{continent}(X))$$

To cast this proposition, from Section 14.3.1, in clausal form, we can (1) negate it, which declares that a value for X which renders the proposition true does not exist, and (2) represent the negated proposition as a goal:

$$\forall X.(\text{false} \subset \text{country}(X) \wedge \text{continent}(X))$$

- **Negation as Failure:** Another manifestation of both the limitation of Horn clauses in Prolog and the issue with the `\+/1` (not) predicate in the `siblings` and `cousins` predicates given previously is that the clause `\+(transmission(X,manual))` means

$$\neg \exists X.(\text{transmission}(X, \text{manual}))$$

(There are no cars with manual transmissions.)

First-Order Predicate Calculus	Logic Programming in Prolog
Any form of proposition is possible.	Restricted to Horn clauses.
Order in which subgoals are searched is insignificant.	Order in which subgoals are searched is significant (left-to-right).
Order in which terms are searched is insignificant.	Order in which clauses are searched is significant (top-down).
$\neg p(X)$ is false when $p(X)$ is true, and vice versa.	$\neg (p(X))$ is false when $p(X)$ is not provable.

Table 14.16 Summary of the Mismatch Between Predicate Calculus and Prolog

rather than

$\exists X. (\neg \text{transmission}(X, \text{manual}))$
(Not all cars have a manual transmission.)

As a result, the goal $\neg (\text{transmission}(X, \text{manual}))$ fails even if the fact $\text{transmission}(\text{accord}, \text{manual})$ is in the database.

- **Occurs-Check Problem:** See Conceptual Exercise 14.8.3.

In summary, there is a mismatch between predicate calculus and Prolog (Table 14.16). Some propositions in predicate calculus cannot be modeled in Prolog. Similarly, the ability to manipulate program control in Prolog (e.g., through the cut predicate or term ordering) is a concept foreign to predicate calculus. Datalog is a subset of Prolog that has no provisions for imparting program control through cuts or clause rearrangement. Unlike Prolog, Datalog is both *sound*—it finds no incorrect solutions—and *complete*—if a solution exists, it will find it. Table 14.13 compares Prolog and Datalog.

While Prolog primarily supports a logic/declarative style of programming, it also supports functional and imperative language concepts. The *pattern-directed invocation* in Prolog is nearly the same as that used in languages supporting functional programming, including ML and Haskell. Similarly, the provisions for supporting program control in Prolog are imperative in nature (e.g., cut). Conversely, UNIX scripting languages for command and control, such as the Korn shell, *sed*, and *awk*, are primarily imperative, but often involve the plentiful use of declaratively specified regular expressions for matching strings. Curry is a language supporting both functional and logic programming.

14.9.2 Reflection in Prolog

Reflection in computer programming refers to a program inspecting itself or altering its contents and behavior while it is running (i.e., computation about computation). The former is sometimes referred to as *introspection* or *read-only reflection* (e.g., a function inquiring how many argument it takes), while the latter is referred to as *intercession*. Table 14.17 presents a suite of reflective predicates built into Prolog. The following are examples of their use:


```

1  /* indicates that automobile
2     is a dynamic predicate */
3  :- dynamic automobile/1.
4
5  automobile(bmw) .
6  automobile(mercedes) .

1  ?- automobile(A) .
2  A = bmw ;
3  A = mercedes.
4
5  ?- asserta (automobile(honda)) .
6  true .
7
8  ?- assertz (automobile(toyota)) .
9  true .

10
11 ?- retract (automobile(bmw)) .
12 true .
13
14 ?- automobile(A) .
15 A = honda ;
16 A = mercedes ;
17 A = toyota.

```

In *Gödel, Escher, Bach: An Eternal Golden Braid*, Douglas R. Hofstadter stated: “A computer program can modify itself but it cannot violate its own instructions—it can at best change some parts of itself by *obeying* its own instructions” (Hofstadter 1979, p. 478).

14.9.3 Metacircular Prolog Interpreter and WAM

The built-in predicate `call/1` is the Prolog analog of the `eval` function in Scheme. The following is an implementation of the `call/1` predicate in Prolog (Harmelen and Bundy 1988):

```

call1(Leaf)           :- clause (Leaf, true) .
call1((Goal1, Goal2)) :- call1(Goal1), call1(Goal2) .
call1(Goal)           :- clause (Goal, Clause), call1(Clause) .

```

These three lines of code constitute the semantic part of the Prolog interpreter. Like Lisp, Prolog is a *homoiconic* language—all Prolog programs are valid Prolog terms. As a result, it is easy—again, as in Lisp—to write Prolog programs that analyze

Predicate	Semantics
<code>assert/1</code> :	Adds a fact to the end of the database.
<code>assertz/1</code> :	Adds a fact to the end of the database.
<code>asserta/1</code> :	Adds a fact to the beginning of the database.
<code>retract/1</code> :	Removes a fact from the database.
<code>var(<Term>)</code> :	Succeeds if <Term> is currently a free variable.
<code>novar(<Term>)</code> :	Succeeds if <Term> is currently not a free variable.
<code>ground(<Term>)</code> :	Succeeds if <Term> holds no free variable.
<code>clause/2</code> :	Matches the head and body of an existing clause in the database; can be used to implement a <i>metacircular interpreter</i> (i.e., an implementation of <code>call/1</code> ; see Section 14.9.3).

Table 14.17 A Suite of Built-in Reflective Predicates in Prolog

other Prolog programs. Thus, the Prolog interpreter shown here is not only a self-interpreter, but a *metacircular interpreter*.

The *Warren Abstract Machine* (WAM) is a theoretical computer that defines an execution model for Prolog programs; it includes an instruction set and memory model (Warren 1983). A feature of WAM code is tail-call optimization (discussed in Chapter 13) to improve memory usage. WAM code is a standard target for Prolog compilers and improves program efficiency in the interpretation that follows. A compiler, called WAMCC, from Prolog to C through the WAM has been constructed and evaluated (Codognet and Diaz 1995).⁹

14.10 The CLIPS Programming Language

CLIPS¹⁰ (C Language Integrated Production System) is a language for implementing expert systems using a logic/declarative style of programming. Originally called NASA's Artificial Intelligence Language (NAIL), CLIPS started as a tool for creating expert systems at NASA in the 1980s. An *expert system* is a computer program capable of modeling the knowledge of a human expert (Giarratano 2008). In artificial intelligence, a production system is a computer system that relies on facts and rules to guide its decision making.

While CLIPS and Prolog both support declarative programming, they use fundamentally different search strategies. Prolog works backward from the goal using resolution to find a series of facts and rules that can be used to satisfy the goal (i.e., backward chaining). CLIPS, in contrast, takes asserted facts and attempts to match them to rules to make inferences (i.e., forward chaining). Thus, unlike Prolog, there is no concept of a goal in CLIPS.

The *Match-Resolve-Act* cycle is the foundation of the CLIPS inference engine, which performs pattern matching between rules and facts through the use of the *Rete Algorithm*. Once the CLIPS inference engine has matched all applicable rules, conflict resolution occurs. Conflict resolution is the process of scheduling rules that were matched at the same time. Once the actions have been performed, the inference engine returns to the pattern matching stage to search for new rules that may be matched as a result of the previous actions. This process continues until a fixed point is reached.

14.10.1 Asserting Facts and Rules

In CLIPS expert systems, as in Prolog, knowledge is represented as facts and *rules*; thus, a CLIPS program consists of a set of facts and rules. For example, a fact may be “it is raining.” In CLIPS, this fact is written as `(assert (weather raining))`. The `assert` keyword defines facts, which are inserted in FIFO order into the `fact-list`. Facts can also be added to the `fact-list` with the `deffacts` command. An example rule is “if it is raining, then I carry an umbrella”:

9. The `wamcc` compiler is available at <https://github.com/thezerobot/wamcc>.

10. <http://www.clipsrules.net/>

```
(defrule ourrule
  (weather raining)
  =>
  (assert (carry umbrella)))
```

The following is the general syntax of a rule¹¹:

```
(defrule rule_name
  (pattern_1) ; IF Condition 1
  (pattern_2) ; And Condition 2
  .
  .
  (pattern_N) ; And Condition N
  => ; THEN
  (action_1) ; Perform Action 1
  (action_2) ; And Action 2
  .
  .
  (action_N)) ; And Action N
```

The CLIPS shell can be invoked in UNIX-based systems with the `clips` command. From within the CLIPS shell, the user can `assert` facts, `defrules`, and `(run)` the inference engine. When the user issues the `(run)` command, the inference engine pattern matches facts with rules. If all patterns are matched within the rule, then the actions associated with that rule are fired. To load facts and rules from an external file, use the `-f` option (e.g., `clips -f database.clp`). Table 14.18 summarizes the commands accessible from within the CLIPS shell and usable in CLIPS scripts. Next, we briefly discuss three language concepts that are helpful in CLIPS programming.

14.10.2 Variables

Variables in CLIPS are prefixed with a `?` (e.g., `?x`). Variables need not be declared explicitly, but they must be bound to a value before they are used. Consider the following program that computes a factorial:

```
(defrule factorial
  (factrun ?x)
  =>
  (assert (fact ?x 1)))

(defrule facthelper
  (fact ?x ?y)
  (test (> ?x 0))
  =>
  (assert (fact (- ?x 1) (* ?x ?y))))
```

When the facts for the rule `facthelper` are pattern matched, `?x` and `?y` are each bound to a value. Next, the bound value for `?x` is used to evaluate the validity of the fact `(test (> ?x 0))`. When variables are bound within a rule, that binding

11. Note that `;` begins a comment.

Command	Function
(run)	Run the inference engine.
(facts)	Retrieve the current fact-list.
(clear)	Restores CLIPS to startup state.
(retract n)	Retract fact n.
(retract *)	Retract all facts.
(watch facts)	Observe facts entering or exiting memory.
(exit)	Exits the CLIPS shell.

Table 14.18 Essential CLIPS Shell Commands

Reproduced from Watkin, Jack L., Adam C. Volk, and Saverio Perugini. 2019. "An Introduction to Declarative Programming in CLIPS and PROLOG." In *Proceedings of the International Conference on Scientific Computing (CSC)*, 105–111. Publication of the World Congress in Computer Science, Computer Engineering, and Applied Computing (CSCE).

exists only within that rule. For persistent global data, `defglobal` should be used as follows:

```
(defglobal ?*var* = " ")
```

Assignment to global variables is done with the `bind` operator.

14.10.3 Templates

Templates are used to associate related data (e.g., facts) in a single package—similar to `structs` in C. Templates are containers for multiple facts, where each fact is a `slot` in the template. Rules can be pattern matched to templates based on a subset of a template's slots. Following is a demonstration of the use of pattern matching to select specific data from a database of facts:

```
(deftemplate car
  (slot make
    (type SYMBOL)
    (allowed-symbols
      truck compact)
    (default compact))
  (multislot name
    (type SYMBOL)
    (default ?DERIVE)))
(deffacts cars
  (car (make truck)
    (name Tundra))
  (car (make compact)
    (name Accord))
  (car (make compact)
    (name Passat)))

(defrule compactcar
  (car (make compact)
    (name ?name))
  =>
  (printout t ?name crlf))
```

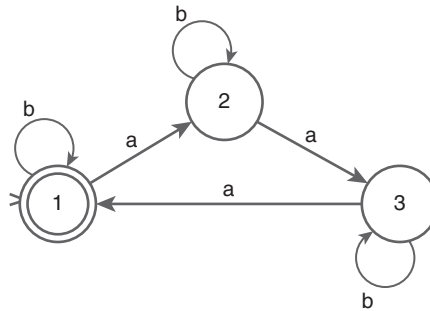
14.10.4 Conditional Facts in Rules

Pattern matching need not match an exact pattern. Logical operators—*or* (`|`), *and* (`&`), and *not* (`~`)—can be applied to pattern operands to support conditional matches. The following rule demonstrates the use of these operators:

```
(defrule walk
  (light ~red&~yellow) ; if the light
                        ; is not yellow and
                        ; is not red
  (cars none|stopped) ; no cars or stopped
=>
  (printout t "Walk" crlf))
```

Programming Exercises for Section 14.10

Exercise 14.10.1 Build a finite state machine using CLIPS that accepts a language L consisting of strings in which the number of a's in the string is a multiple of 3 over an alphabet $\{a,b\}$. Use the following state machine for L :



Reproduced from Arabnia, Hamid R., Leonidas Deligiannidis, Michale R. Grimala, Douglas D. Hodson, and Fernando G. Tinetti. 2019. *CSC'19: Proceedings of the 2019 International Conference on Scientific Computing*. Las Vegas: CSREA Press.

Examples:

```
CLIPS> (run)
Input string: aaabba
Rejected
CLIPS> (reset)
CLIPS> (run)
Input string: aabbbba
Accepted
```

Exercise 14.10.2 Rewrite the factorial program in Section 14.10.2 so that only the fact with the final result of the factorial rule is stored in the fact list. Note that `retract` can be used to remove facts from the fact list.

Examples:

```
CLIPS> (assert (factrun 5))
CLIPS> (run)
CLIPS> (facts)
f-0 (factrun 5)
f-1 (fact 0 120)
```

14.11 Applications of Logic Programming

Applications of logic/declarative programming include cryptarithmic problems, puzzles (e.g., tic-tac-toe), artificial intelligence, and design automation. In this section, we briefly introduce some other applications of Prolog and CLIPS.

14.11.1 Natural Language Processing

One application of Prolog is *natural language processing* (Eckroth 2018; Matthews 1998)—the search engine used by Prolog naturally functions as a recursive-descent parser. One could conceive facts as terminals and rules as non-terminals or production rules. Consider the following simple grammar:

```
<sentence> ::= <noun phrase> <verb phrase>
<noun phrase> ::= <determiner> <adj noun phrase>
<noun phrase> ::= <adj noun phrase>
<adj noun phrase> ::= <adj> <adj noun phrase>
<adj noun phrase> ::= <noun>
<verb phrase> ::= <verb> <noun phrase>
<verb phrase> ::= <verb>
```

Using this grammar, a Prolog program can be written to verify the syntactic validity of a sentence. The candidate sentence is represented as a list in which each element is a single word in the language (e.g., `sentence(["The", "dog", "runs", "fast"])`).

```
sentence(S) :- append(NP, VP, S), noun_phrase(NP), verb_phrase(VP).

noun_phrase(NP) :- append(ART, NP2, NP), det(ART), noun_phrase_adj(NP2).

noun_phrase(NP) :- noun_phrase_adj(NP).

noun_phrase_adj(NP) :- append(ADJ, NPADJ, NP),
                        adjective(ADJ), noun_phrase_adj(NPADJ).

noun_phrase_adj(NP) :- noun(NP).

verb_phrase(VP) :- append(V, NP, VP), verb(V), noun_phrase(NP).

verb_phrase(VP) :- verb(VP).
```

A drawback of using Prolog to implement a parser is that left-recursive grammars cannot be implemented for the same reasons discussed in Section 14.6.4.

14.11.2 Decision Trees

An application of CLIPS is decision trees. More generally, CLIPS can be applied to graphs that represent a human decision-making process. Facts can be thought of as the edges of these graphs, while rules can be thought of as the actions or states associated with each vertex of the graph. An example of this decision-making process is an expert system that emulates a physician in treating, diagnosing, and explaining diabetes (Garcia et al. 2001). The patient asserts facts about herself including eating habits, blood-sugar levels, and symptoms. The rules within this expert system match these facts and provide recommendations about managing diabetes in the same way that a physician might interact with a patient.

14.12 Thematic Takeaways

- In declarative programming, the programmer specifies *what* they want to compute, not *how* to compute it.
- In logic programming, the programmer specifies a knowledge base of known propositions—axioms declared to be true—from which the system infers new propositions using a deductive apparatus:

representing the relevant knowledge	←	predicate calculus
rule of inference	←	resolution

- Propositions in a logic program are purely syntactic, so they have no intrinsic semantics—they can mean whatever the programmer wants them to mean.
- In Prolog, the programmer specifies a knowledge base of facts and rules as a set of Horn clauses—a canonical representation for propositions—and the system uses resolution to determine the validity of goal propositions issued as queries, which are also represented as Horn clauses.
- Unlike Prolog, which uses *backward chaining*, CLIPS uses *forward chaining*—there is no concept of a goal in CLIPS.
- There is a mismatch between predicate calculus and Prolog. Some things can be modeled in one but not the other, and vice versa.
- While Prolog primarily supports a logic/declarative style of programming, it also supports functional and imperative language concepts.
- The ultimate goal of logic/declarative programming is to make programming entirely an activity of specification—programmers should not have to impart control upon the program. Prolog falls short of the ideal.

14.13 Chapter Summary

In contrast to an imperative style of programming, in which programmers specify *how* to compute a solution to a problem, in a logic/declarative style of programming, programmers specify *what* they want to compute, and

the system uses a built-in search strategy to compute a solution. Prolog is a classical programming language supporting a logic/declarative style of programming.

Logic/declarative programming is based on a formal system of symbolic logic called *first-order predicate calculus*. In logic programming, the programmer specifies a knowledge base of known propositions—axioms declared to be true—from which the system infers new propositions using a deductive apparatus. Propositions in a logic program are purely syntactic, so they have no intrinsic semantics—they can mean whatever the programmer wants them to mean. The primary rule of inference used in logic programming is *resolution*. Resolution in predicate calculus requires *unification* and *instantiation* to match terms. There are two ways resolution can be applied to the propositions in the knowledge base of a system supporting logic programming: *backward chaining*, where the inference engine works backward from a goal to find a path through the database sufficient to satisfy the goal (e.g., Prolog); and *forward chaining*, where the engine starts from the given facts and rules to deduce new propositions (e.g., CLIPS).

In Prolog, the programmer specifies a knowledge base of facts and rules as a set of Horn clauses—a canonical representation for propositions—and the system uses resolution to determine the validity of goal propositions issued as queries (i.e., backward chaining), which are also expressed as Horn clauses. While Prolog primarily supports a logic/declarative style of programming, it also supports functional (e.g., pattern-directed invocation) and imperative (e.g., cut) language concepts.

There is a mismatch between predicate calculus and Prolog. Some things can be modeled in one but not the other, and vice versa. In particular, Prolog equips the programmer with facilities to impart control over the search strategy used by the system (e.g., the cut operator). These control facilities violate a defining principle of declarative programming—that is, the programmer need only be concerned with the logic and leave the control (i.e., the inference methods used to satisfy goals) up to the system. Moreover, Prolog searches its database in a top-down manner and searches subgoals from left to right during resolution—this approach constructs a search tree in a depth-first fashion. Thus, the use of a left-recursive rule in a Prolog program is problematic due to the left-to-right pursuit of the subgoals.

CLIPS is programming language for building expert systems that supports a declarative style of programming. Unlike Prolog, which uses *backward chaining*, CLIPS uses *forward chaining* to deduce new propositions—there is no concept of a goal in CLIPS.

The goal of logic programming is to make programming entirely an activity of specification—programmers should not have to impart control upon the program. Thus, Prolog falls short of the ideal. Datalog and Mercury foster a purer form of declarative programming than Prolog, because, unlike Prolog, they do not support control facilities intended to circumvent or direct the system's built-in search strategy.

14.14 Notes and Further Reading

A detailed treatment of the steps necessary to convert a wff in predicate calculus into clausal form can be found in Rich, Knight, and Nair (2009, Section 5.4.1). The *unification algorithm* used during resolution, which rendered logic programming practical, was developed by John Alan Robinson (1965). For a detailed outline of the steps of the unification algorithm, we direct readers to Rich, Knight, and Nair (2009, Section 5.4.4). For a succinct introduction to Prolog, we refer readers to Pereira (1993). The Watson question-answering system from IBM was developed in part in Prolog. Some parts of this chapter, particularly Section 14.10, appear in Watkin, Volk, and Perugini (2019).

Chapter 15

Conclusion

Well, what do you know about that! These forty years now, I've been speaking in prose without knowing it!

— Monsieur Jourdain in Molière's *The Bourgeois Gentleman* (new verse adaptation by Timothy Mooney)

A programming language is for thinking about programs, not for expressing programs you've already thought of. It should be a pencil, not a pen.

— Paul Graham

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

— Sperber et al. (2010)

WE have come to the end of our journey through the study of programming languages. Programming languages are the conduits through which we describe, affect, and experience computation. We set out in this course of study to establish an understanding of programming language *concepts*. We did this in five important ways:

1. We explored the methods of both defining the *syntax* of programming languages and implementing the syntactic part of a language (Chapters 2–4).
2. We learned functional programming, which is different from the imperative and object-oriented programming with which readers may have been more familiar (Chapters 5–6 and 8).
3. We studied *type systems* (Chapter 7) and *data abstraction* techniques (Chapter 9).
4. We built *interpreters* for languages to operationalize language *semantics* for a variety of concepts (Chapters 10–12).

<i>Imperative</i>	programming describes computation through	<i>side effect</i> and <i>iteration</i> .
<i>Functional</i>	programming describes computation through	<i>function calls that return values</i> and <i>recursion</i> .
<i>Logic/declarative</i>	programming describes computation through	<i>the declaration of a knowledge base</i> and <i>built-in resolution</i> .
<i>Bottom-up</i>	programming describes computation through	<i>building up a language</i> and <i>then using it</i> .

Table 15.1 Reflection on Styles of Programming

5. We encountered and experienced these concepts through *other styles of programming*, particularly *programming with continuations* (Chapter 13) and *logic/declarative programming* (Chapter 14) and discovered that despite differences in semantics all languages support a set of core concepts.

This process has taught us how to *use*, *compare*, and *build* programming languages. It has also made us better programmers and well-rounded computer scientists.

15.1 Language Themes Revisited

We encourage readers to revisit the book objectives presented in Section 1.1. We also encourage readers to reconsider the recurring themes identified in Section 1.6 of Chapter 1. Table 15.1 summarizes the style of programming we encountered.

15.2 Relationship of Concepts

Figure 15.1 casts some of the language concepts we studied in relationship to each other. In particular, a solid directed arrow indicates that the target concept relies only on the presence of the source concept; a dotted directed arrow indicates that the target concept relies partially on the presence of the source concept. If a language supports all of the concepts emanating from the dotted incoming edges of some node, then it can support the concept represented by that node. For instance, the two dotted arrows into the recursion node express the result of the *fixed-point Y combinator*: Support for recursion can be built into any language that supports first-class *and* λ /anonymous functions. However, generators/iterators are supported by either the presence of lazy evaluation *or* first-class continuations.

Notice the central relationship of closures to other concepts in Figure 15.1. A first-class, heap-allocated closure is a fundamental construct/primitive for creating abstractions in programming languages. For instance, we built support for currying/uncurrying in Scheme using closures (Table 8.4) as well as ML, Haskell, and Python in Programming Exercises in Chapter 8. We also supported lazy evaluation (i.e., pass-by-need parameters) using heap-allocated lexical closures

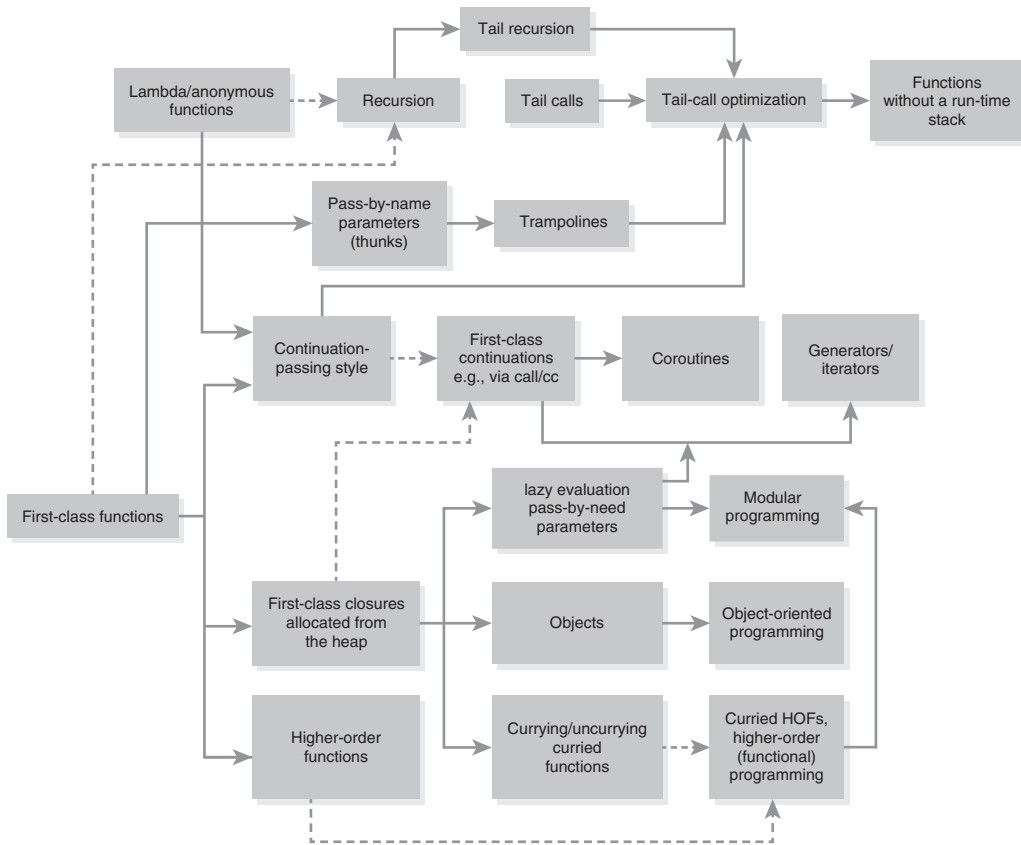


Figure 15.1 The relationships between some of the concepts we studied. A solid directed arrow indicates that the target concept relies only on the presence of the source concept. A dotted directed arrow indicates that the target concept relies partially on the presence of the source concept.

in Chapter 12 (e.g., in Python in Section 12.5.5 and in Scheme in Programming Exercise 12.5.19). Similarly, we studied how to build any control abstraction (e.g., iteration, conditionals, repetition, `gotos`, coroutines, and lazy iterators) using *first-class continuations* in Scheme in Chapter 13.

We also implemented recursion from first principles in Scheme using first-class, non-recursive λ /anonymous functions in Chapter 5. The following is the Python rendition of the construction of recursion (in the list `length1` function in Section 5.9.3):

```

print ((lambda f,l: f(f,l))
      ((lambda f,l: 0 if l == [] else (1 + f(f, l[1:]))),
       ["a", "b", "c", "d"]))

```

The abstraction baked into this expression is isolated in the *fixed-point Y combinator*, which we implemented in JavaScript in Programming Exercise 6.10.15.

15.3 More Advanced Concepts

We discussed how a *higher-order function* can capture a pattern of recursion. If the function returned by a HOF at run-time accesses the environment in which it was created, it is called a *lexical closure*—a package that encapsulates an *environment* and an *expression*. Lexical closures resemble objects from object-oriented programming. Higher-order functions, the lexical closures they can return, and the style of programming both support and lead to the concept of *macros*—an operator that writes a program. While a higher-order function can return at run-time a function that was written before run-time, a macro can write a program at run-time. This style of programming is called *metaprogramming*. The *homoiconic* nature of languages like Lisp (and Prolog), where programs are represented using a primitive data structure in the language itself, more easily facilitates metaprogramming than does a non-homoiconic language. Lisp programs are expressed as lists, which means that a Lisp program can generate Lisp code and subsequently interpret Lisp code at run-time—through the built-in `eval` function. The quirky syntax in Lisp that makes the language homoiconic allows the programmer to directly write programs as abstract-syntax trees (Section 9.5) that the front end of (traditional) languages generate (Figures 3.1–3.2 and 4.1–4.2). This AST, however, has first-class status in Lisp: the programmer has access to it and, thus, can write functions that write functions called macros that manipulate it (Graham 2004b, p. 177). Macros support the addition of new operators to language. Adding new operators to an existing language makes the existing language a new language. Thus, macros are a helpful ingredient in defining new languages or *bottom-up programming*: “the Scheme macro system permits programmers to add constructs to Scheme, thereby effectively providing a compiler from Scheme+ (the extended Scheme language) to Scheme itself” (Krishnamurthi 2003, p. 319).

15.4 Bottom-up Programming

Bottom-up programming is a type of metaprogramming that has been referred to as *language-oriented programming* (Felleisen et al. 2018). In bottom-up programming, “[i]nstead of subdividing a task down into smaller units [(i.e., top-down programming)], you build a ‘language’ of ideas up toward your task” (Graham 2004b, p. 242).

...Lisp is a programmable programming language. Not only can you program in Lisp (that makes it a programming language) but you can program the language itself. This is possible, in part, because Lisp programs are represented as Lisp data objects, and partly because there are places during the scanning, compiling and execution of Lisp programs where user-written programs are given control. (Foderaro 1991, p. 27)

Often the resulting language is called a *domain-specific* (e.g., SQL) or *embedded language*. It has been said that “[i]f you give someone Fortran, he has Fortran. If

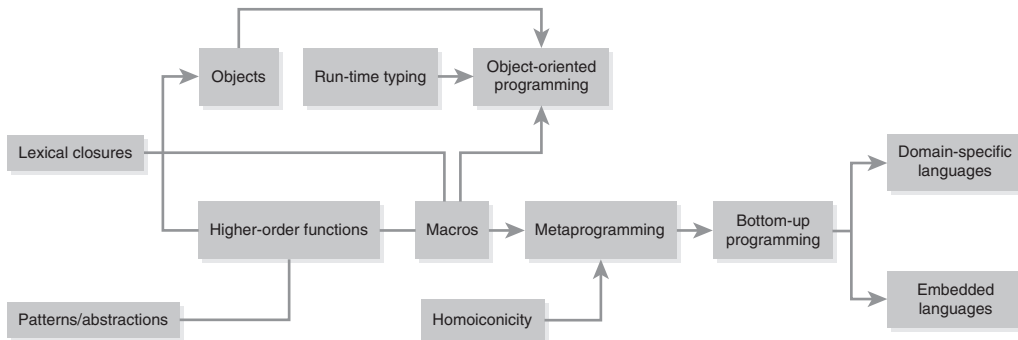


Figure 15.2 Interplay of advanced concepts of programming languages. A directed edge indicates a “leads to” relationship, while an undirected edge indicates a general relation.

you give someone Lisp, he has any language he pleases” (Friedman and Felleisen 1996b, Afterword, p. 207). For instance, support for object-oriented programming can be built from the abstractions already available to the programmer in Lisp (Graham 1993, p. ix). Lisp’s support for macros, closures, and dynamic typing lifts object-oriented programming to another level (Graham 1996, p. 2). Figure 15.2 depicts the relationships between these advanced concepts of programming languages. (Notice that macros are central in Figure 15.2, much as closures are central in Figure 15.1.) *Homoiconic* languages with *macros* (e.g., Lisp and Clojure) simplify metaprogramming and, thus, bottom-up programming (Figure 15.2). We encourage readers to explore macros and bottom-up programming further, especially in the works by Graham (1993, 1996) and Krishnamurthi (2003).

Lastly, let us reconsider some of the ideas introduced in Chapter 1. Over the past 20 years or so, certain language concepts introduced in foundational languages have made their way into more contemporary languages. Today, language concepts conceived in Lisp and Smalltalk—first-class functions and closures, dynamic binding, first-class continuations, and homoiconicity—are increasingly making their way into contemporary languages. Heap-allocated, first-class, lexical closures; first-class continuations; homoiconicity; and macros are concepts and constructs for building language abstractions to make programming easier.

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary (Sperber et al. 2010).

Ample scope for exploration and discovery in the terrain of programming languages remains:

Programming language research is short of its ultimate goal, namely, to provide software developers tools for formulating solutions in the languages of problem domains. (Felleisen et al. 2018, p. 70)

Conceptual Exercises for Chapter 15

Exercise 15.1 Aside from dynamic scoping, list two specific concepts that are examples of dynamic binding in programming languages. Describe what is being bound to what in each example.

Exercise 15.2 Identify a programming language with which you are unfamiliar. Armed with your understanding of language concepts, design options, and styles of programming as a result of formal study of language and language concepts, describe the language through its most defining characteristics. If you completed Conceptual Exercise 1.16 when you embarked on this course of study, revisit the language you analyzed in that exercise. In which ways do your two (i.e., before and after) descriptions of that language differ?

Exercise 15.3 Revisit the recurring book themes introduced in Section 1.6 and reflect on the instances of these themes you encountered through this course of study. Classify the following items using the themes outlined in Section 1.6.

- Comments cannot nest in C and C++.
- Scheme uses prefix notation for both operators and functions—there really is no difference between the two in Scheme. Contrast with C, which uses infix notation for operators and prefix notation for functions.
- The `while` loop in Camille.
- Static vis-à-vis dynamic scoping.
- Lazy evaluation enables the implementation of complex algorithms in a concise way (e.g., quicksort in three lines of code, *Sieve of Eratosthenes*).
- C uses pass-by-name for the `if` statement, but pass-by-value for user-defined functions.
- Deep, ad hoc, and shallow binding.
- All operators use lazy evaluation in Haskell.
- First version of Lisp used dynamic scoping, which is easier to implement than lexical scoping but turned out to be less natural to use.
- In Smalltalk, everything is an object and all computation is described as passing messages between objects.
- Conditional evaluation in Camille.
- Multiple parameter-passing mechanisms.

Exercise 15.4 Reflect on why some languages have been in use for more than 50 years (e.g., Fortran, C, Lisp, Prolog, Smalltalk), while others are either no longer supported or rarely, if ever, used (e.g., APL, PL/1, Pascal). Write a short essay discussing the factors affecting language survival.

Exercise 15.5 Write a short essay reflecting on how you met, throughout this course of study, the learning outcomes identified in Section 1.8. Perhaps draw some diagrams to aid your reflection.

15.5 Further Reading

If you enjoy languages and enjoyed this course of study, you may enjoy the following books:

Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. 1977. *A Pattern Language: Towns, Buildings, Construction*. New York, NY: Oxford University Press.

Carroll, Lewis. 1865. *Alice's Adventures in Wonderland*.

Carroll, Lewis. 1872. *Through the Looking-Glass, and What Alice Found There*.

Friedman, D. P., and M. Felleisen. 1996. *The Little Schemer*. 4th ed. Cambridge, MA: MIT Press.

Friedman, D. P., and M. Felleisen. 1996. *The Seasoned Schemer*. Cambridge, MA: MIT Press.

Friedman, D. P., W. E. Byrd, O. Kiselyov, and J. Hemann. 2005. *The Reasoned Schemer*. 2nd ed. Cambridge, MA: MIT Press.

Graham, P. 1993. *On Lisp*. Upper Saddle River, NJ: Prentice Hall. Available: <http://paulgraham.com/onlisp.html>.

Graham, P. 2004. *Hackers and Painters: Big Ideas from the Computer Age*. Beijing: O'Reilly.

Hofstadter, D. R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York, NY: Basic Books.

Kiczales, G., J. des Rivieres, and D. G. Bobro. 1991. *The Art of the Metaobject Protocol*. Cambridge, MA: MIT Press.

Korienek, G., T. Wrensch, and D. Dechow. 2002. *Squeak: A Quick Trip to ObjectLand*. Boston, MA: Addison-Wesley.

Tolkien, J. R. R. 1973. *The Hobbit*. New York, NY: Houghton Mifflin.

Tolkien, J. R. R. 1991. *The Lord of the Rings*. London, UK: Harper Collins.

Weinberg, G. M. 1988. *The Psychology of Computer Programming*. New York, NY: Van Nostrand Reinhold.

Appendix A

Python Primer

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one—and preferably only one—obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than **right** now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea—let's do more of those!

— Tim Peters, *The Zen of Python* (2004)¹

PYTHON is a programming language that blends features from imperative, functional, and object-oriented programming.

A.1 Appendix Objective

Establish an understanding of the syntax and semantics of Python through examples so that a reader with familiarity with imperative, and some functional,

1. >>> `import this`

programming, after having read this appendix can write intermediate programs in Python.

A.2 Introduction

Python is a statically scoped language, uses an eager evaluation strategy, incorporates functional features and a terse syntax from Haskell, and incorporates data abstraction from Dylan and C++. One of the most distinctive features of Python is its use of indentation to demarcate blocks of code. While Python was developed and implemented in the late 1980s in the Netherlands by Guido van Rossum, it was not until the early 2000s that the language's use and popularity increased. Python is now embraced as a general-purpose, interpreted programming language and is available for a variety of platforms.

This appendix is not intended to be a comprehensive Python tutorial or language reference. Its primary objective is to establish an understanding of Python programming in a reader already familiar with imperative and some functional programming as preparation for the use of Python, through which to study of concepts of programming languages and build language interpreters in this text. Because of the multiple styles of programming it supports (e.g., imperative, object-oriented, and functional), Python is a worthwhile vehicle through which to explore language concepts, including lexical closures, lambda functions, iterators, dynamic type systems, and automatic memory management. (Throughout this text, we explore closures (in Chapter 6), typing (in Chapter 7), currying and higher-order functions (in Chapter 8), type systems (in Chapter 9), and lazy evaluation (in Chapter 12) through Python. We also build language interpreters in Python in Chapters 10–12.) We leave the use of Python for exploring language concepts for the main text of this book.

This appendix is designed to be straightforward and intuitive for anyone familiar with imperative and functional programming in another language, such as Java, C++, or Scheme. We often compare Python expressions to their analogs in Scheme. We use the Python 3.8 implementation of Python. Note that `>>>` is the prompt for input in the Python interpreter used in this text.

A.3 Data Types

Python does not have primitive types since all data in Python is represented as an object. Integers, booleans, floats, lists, tuples, sets, and dicts are all instances of classes:

```
>>> help(int)
Help on class int in module builtins:

class int(object)
|   int([x]) -> integer
|   int(x, base=10) -> integer
|
```

```
| Convert a number or string to an integer, or return 0 if  
| no arguments are given. If x is a number, return x.__int__().  
| For floating point numbers, this truncates towards zero.  
...
```

To convert a value of one type to a value of another type, the constructor method for the target type class can be called:

```
>>> # invoking int constructor to instantiate  
>>> # an int object out of the passed string  
>>> int("123")  
123  
  
>>> type(int("123"))  
<class 'int'>  
  
>>> str(123)  
'123'  
  
>>> type(str(123))  
<class 'str'>  
  
>>> str(int("123"))  
'123'  
  
>>> type(str(int("123")))  
<class 'str'>
```

Python has the following types: numeric (`int`, `float`, `complex`), sequences (`str`, `unicode`, `list`, `tuple`, `set`, `bytearray`, `buffer`, `xrange`), mappings (`dict`), files, classes, instances and exceptions, and `bool`:

```
>>> bool  
<type 'bool'>  
  
>>> type(True)  
<type 'bool'>  
  
>>> str  
<type 'str'>  
  
>>> type('a')  
<type 'str'>  
  
>>> type("hello world")  
<type 'str'>  
  
>>> int  
<type 'int'>  
  
>>> type(3)  
<type 'int'>  
  
>>> float  
<type 'float'>  
  
>>> type(3.3)  
<type 'float'>  
  
>>> list
```

```

<type 'list'>

>>> type([2,3,4])
<type 'list'>

>>> type([2,2.1,"hello"])
<type 'list'>

>>> tuple
<type 'tuple'>

>>> type((2,3,4))
<type 'tuple'>

>>> type((2, 2.1, "hello"))
<type 'tuple'>

>>> set
<class 'set'>

>>> type({1,2,3,3,4})
<class 'set'>

>>> dict
<type 'dict'>

>>> type({'ID': 1, 'Name': 'Mary', 'Rate': 7.75,
...      'Promoted?': True})
<type 'dict'>

```

For a list of all of the Python built-in types, enter the following:

```

>>> import builtins
>>> help(builtins)
Help on built-in module builtins:

NAME
    builtins - Built-in functions, exceptions, and other objects.

DESCRIPTION
    Noteworthy: None is the `nil' object;
    Ellipsis represents `...' in slices.

CLASSES
    object
        BaseException
        Exception
        ArithmeticError
    ...

```

Python does not use explicit type declarations for variables, but rather uses type inference as variables are (initially) assigned a value. Memory for variables is allocated when variables are initially assigned a value and is automatically garbage collected when the variable goes out of scope.

In Python, ' and " have the same semantics. When quoting a string containing single quotes, use double quotes, and vice versa:

```

>>> 'use single quotes when a "string" contains double quotes'
'use single quotes when a "string" contains double quotes'

```

```
>>> "use double quotes when a 'string' contains single quotes"
"use double quotes when a 'string' contains single quotes"
```

Alternatively, as in C, use `\` to escape the special meaning a `"` within double quotes:

```
>>> "use backslash to escape a \"double quote\" in double quotes"
'use backslash to escape a "double quote" in double quotes'
```

A.4 Essential Operators and Expressions

Python is intended for programmers who want to get work done quickly. Thus, it was designed to have a terse syntax, which even permeates the writability of Python programs. For instance, in what follows notice that a Python programmer rarely needs to use a `;` (semicolon).

- **Character conversions.** The `ord` and `chr` functions are used for character conversions:

```
>>> ord('a')
97
>>> chr(97)
'a'
>>> chr(ord('a'))
'a'
```

- **Numeric conversions.**

```
>>> int(3.4) # type conversion
3
>>> float(3)
3.0
```

- **String concatenation.** The `+` is the infix binary append operator that is used for concatenating two strings.

```
>>> "hello" + " " + "world"
'hello world'
```

- **Arithmetic.** The infix binary operators `+`, `-`, and `*` have the usual semantics. Python has two division operators: `//` and `/`. The `//` operator is a floor division operator for integer and float operands:

```
>>> 10 // 3
3
>>> -10 // 3
-4
>>> 10.0 // 3.333
3.0
>>> -10.0 // 3.333
-4.0
>>> 4 // 2
2
>>> 1 // -2
-1
```

Thus, integer division with `//` in Python floors, unlike integer division in C which truncates. Unlike `//`, the `/` division operator always returns a float:

```
>>> 10 / 3
3.3333333333333335
>>> -10 / 3
-3.3333333333333335
>>> 10.0 / 3.333
3.0003000300030003
>>> -10.0 / 3.333
-3.0003000300030003
>>> 4 / 2
2.0
>>> 1 / -2
-0.5
```

- **Comparison.** The infix binary operators `==` (equal to), `<`, `>`, `<=`, `>=`, and `!=` (not equal to) compare integers, floats, characters, strings, and values of other types:

```
>>> 4 == 2
False
>>> 4 > 2
True
>>> 4 != 2
True
>>> 'b' > 'a'
True
>>> ['b'] > ['a']
True
```

- **Boolean operators.** The infix operators `or`, `and`, and `not` are used with the usual semantics. The operators `or` and `and` use *short-circuit evaluation* (or *lazy evaluation* as discussed in Chapter 12):

```
>>> True or False
True
>>> False and False
False
>>> not False
True
```

- **Conditionals.** Use `if` and `if-else` statements:

```
>>> if 1 != 2:
...     "Python has a one-armed if statement"
...
'Python has a one-armed if statement'
>>>

>>> if 1 != 2:
...     "true branch"
... else:
...     "false branch"
...
'true branch'
```

- **Code indentation.** Indentation, rather than curly braces, is used in Python to delimit blocks of code. **Code indentation is significant in Python.** Two programs that are identical lexically when ignoring indentation are *not* the same in Python. One may be syntactically correct while the other may not. For instance:

```
>>> if 1 != 2:
...   "Python has a one-armed if statement"
      File "<stdin>", line 2
        "Python has a one-armed if statement"
            ^
IndentationError: expected an indented block

>>> if 1 != 2:
...     "true branch"
...     else:
      File "<stdin>", line 3
        else:
            ^
IndentationError: unindent does not match any outer
indentation level

>>> if 1 != 2: "true branch" else: "false branch"
      File "<stdin>", line 1
        if 1 != 2: "true branch" else: "false branch"
            ^
SyntaxError: invalid syntax
```

The indentation conventions enforced by Python are for the benefit of the programmer—to avoid buggy code. As Bruce Eckel says:

[B]ecause blocks are denoted by indentation in Python, indentation is uniform in Python programs. And indentation is meaningful to us as readers. So because we have consistent code formatting, I can read somebody else's code and I'm not constantly tripping over, "Oh, I see. They're putting their curly braces here or there." I don't have to think about that. (Venners 2003)

- **Comments.**

- Single-line comments:

```
>>> i=1 # single-line comment until the end of the line
```

- Multi-line comments. While Python does not have a special syntax for multi-line comments, a multi-line comment can be simulated using a multi-line string because Python ignores a string if it is not being used in an expression or statement. The syntax for multi-line strings in Python uses triple quotes—either single or double:

```
>>> hello = """Hello, this is a
... multi-line string."""
>>>
>>> hello
'Hello, this is a\nmulti-line string.'
```


In a Python source code file, a mutli-line string can be used as a comment if the first and last triple quotes are not on the same lines as other code:

```
print("This is code.")
"""
This string will be ignored by the Python interpreter
because it is not being used in an expression or statement.
Thus, it functions as a multi-line comment.
"""
print("More code.")
"Regular strings can also function as comments,"
# but since Python has special syntax for a
# single-line comment, they typically are not used that way.
```

- Docstrings are also used to comment, annotate, and document functions and classes:

```
>>> def a_function():
...     """
...     This is where docstrings reside for functions.
...     A docstring can be a single- or multi-line string.
...     Docstrings are used by the Python help system.
...     """
...     print("Function body")
...
>>> # Invokes Python's help system
>>> # where the docstring is used.
>>> help(a_function)
Help on function a_function in module __main__:

a_function()
    This is where docstrings reside for functions.
    A docstring can be a single- or multi-line string.
    Docstrings are used by the Python help system.

>>> # Docstrings can also be accessed
>>> # from a Python program.
>>> a_function.__doc__
"\n  This is where docstrings reside for functions.\n
A docstring can be a single- or multi-line string.\n
Docstrings are used by Python's help system.\n  "
```

- The list/split and join functions are Python's analogs of the explode and implode functions in ML, respectively:

```
>>> list("apple")
['a', 'p', 'p', 'l', 'e']

>>> ''.join(['a', 'p', 'p', 'l', 'e'])
'apple'

>>> ''.join(list("apple"))
'apple'

>>> "parse me into a list of strings".split(' ')
['parse', 'me', 'into', 'a', 'list', 'of', 'strings']
```

```
>>> list("parse me ...")
['p', 'a', 'r', 's', 'e', ' ', ' ', 'm', 'e', ' ', ' ', '.', '.', '.']

>>> ' '.join("parse me into a list of strings".split(' '))
'parse me into a list of strings'
```

- To run a Python program:
 - Enter `python`² at the command prompt and then enter expressions interactively to evaluate them:

```
$ python
>>> 2 + 3
5
>>>
```

Using this method of execution, the programmer can create bindings and define new functions at the prompt of the interpreter:

```
1 >>> answer = 2 + 3
2 >>> answer
3 5
4 >>> def f(x):
5 ...     return x + 1
6 ...
7 >>> f(1)
8 2
9 >>> ^D
10 $
```

Enter the EOF character [which is `<ctrl-d>` on UNIX systems (line 9) and `<ctrl-z>` on Windows systems] or `quit()` to exit the interpreter.

- Enter `python <filename>.py` from the command prompt using file I/O, which causes the program in `<filename>.py` to be evaluated line by line by the interpreter:³

```
1 $ cat first.py
2
3 answer = 2 + 3
4
5 answer
6
7 def f(x):
8     return x + 1
9
10 f(1)
11
12 $ python first.py
13 $
```

2. The name of the executable file for the Python interpreter may vary across systems (e.g., `python3.8`).

3. The interpreter automatically exits once EOF is reached and evaluation is complete.

Using this method of execution, the return value of the expressions (lines 5 and 10 in the preceding example) is not shown unless explicitly printed (lines 5 and 10 in the next example):

```

1  $ cat first.py
2
3  answer = 2 + 3
4
5  print(answer)
6
7  def f(x):
8      return x + 1
9
10 print(f(1))
11 $
12 $ python first.py
13 5
14 2
15 $

```

- Enter `python` at the command prompt and then load a program by entering `import <filename>` (without the `.py` filename extension) into the interpreter (line 18 in the next example):

```

1  $ cat first.py
2
3  answer = 2 + 3
4
5  print(answer)
6
7  def f(x):
8      return x + 1
9
10 print(f(1))
11
12 $ python
13 Python 3.8.3 (default, May 15 2020, 14:33:52)
14 [Clang 10.0.1 (clang-1001.0.46.4)] on darwin
15 Type "help", "copyright", "credits" or "license"
16 for more information.
17 >>>
18 >>> import first
19 5
20 2
21 >>>

```

If the program is modified, enter the following lines into the interpreter to reload it:

```

>>> from importlib import reload

>>> reload(first) # answer = 2+3 modified to answer = 2+4
6
2
<module 'first' from 'src/first.py'>

```

- Redirect standard input into the interpreter from the keyboard to a file by entering `python < <filename>.py` at the command prompt:⁴

```
$ cat first.py

answer = 2 + 3

print(answer)

def f(x):
    return x + 1

print(f(1))

$ python < first.py
5
2
$
```

A.5 Lists

As in Scheme, but unlike in ML and Haskell, lists in Python are *heterogeneous*, meaning all elements of the list need not be of the same type. For example, the list `[2, 2.1, "hello"]` in Python is heterogeneous while the list `[2, 3, 4]` in Haskell is homogeneous. Like ML and Haskell, Python is type safe. However, Python is dynamically typed, unlike ML and Haskell. The semantics of `[]` is the empty list. Tuples (Section A.6) are more appropriate to store unordered items of different types. Lists in Python are indexed using zero-based indexing. The `+` is the append operator that accepts two lists and appends them to each other.

Examples:

```
>>> [1,2,3]
[1, 2, 3]

>>> [1.1,2,False,"hello"]
[1.1,2,False,"hello"]

>>> []
[]

>>> # return the first element or head of the list
>>> [1.1,2,False,"hello"][0]
1.1

>>> # return the tail of the list
>>> [1.1,2,False,"hello"][1:]
[2, False, 'hello']

>>> # return the last element of the list
>>> [1.1,2,False,"hello"][len([1.1,2,False,"hello"])-1]
'hello'
```

4. Again, the interpreter automatically exits once EOF is reached and evaluation is complete.

```

>>> # return the last element of the list easier
>>> [1.1,2,False,"hello"][-1]
'hello'

>>> "hello world"[0]
'h'

>>> "hello world"[1]
'e'

>>> [2,2.1,"2"][2].isdigit()
True
>>> "hello world"[2].isdigit()
False

>>> [1,2,3][2]
3

>>> [1,2,3]+[4,5,6]
[1, 2, 3, 4, 5, 6]

>>> [i for i in range(5)] # a list comprehension
[0, 1, 2, 3, 4]

>>> [i*i for i in range(5)]
[0, 1, 4, 9, 16]

```

Pattern matching. Python supports a form of *pattern matching* with lists:

```

>>> head, *tail = [1,2,3]
>>> head
1
>>> tail
[2, 3]
>>> first, second, *rest = [1,2,3,4,5]
>>> first
1
>>> second
2
>>> rest
[3, 4, 5]
>>> head, *tail = [1]
>>> head
1
>>> tail
[]
>>> lst = [1,2,3,4,5]
>>> x, xs = lst[0], lst[1:]
>>> x
1
>>> xs
[2, 3, 4, 5]

```

Lists in Python vis-à-vis lists in Lisp. There is not a direct analog of the `cons` operator in Python. The append list operator `+` can be used to simulate `cons`, but its time complexity is $O(n)$. For instance,

<code>(cons x y)</code>	(in Scheme)	<code>≡</code>
<code>x:y</code>	(in ML)	<code>≡</code>
<code>[x] + y</code>	(in Python)	<code>≡</code>
<code>y.append(x)</code>	(in Python).	

Examples:

```
>>> [1] + [2] + [3] + [] # Python analog of 1:2:[3] in ML
[1, 2, 3]

>>> [1] + []             # Python analog of 1:[] in ML
[1]

>>> [1] + [2] + []       # Python analog of 1:2:[] in ML
[1, 2]
```

A.6 Tuples

A *tuple* is a sequence of elements of potentially mixed types. A tuple typically contains unordered, *heterogeneous* elements akin to a `struct` in C with the exception that a tuple is indexed by numbers (like a list) rather than by field names (like a `struct`). Formally, a tuple is an element e of a Cartesian product of a given number of sets: $e \in (S_1 \times S_2 \times \cdots \times S_n)$. A two-element tuple is called a *pair* [e.g., $e \in (A \times B)$]. A three-element tuple is called a *triple* [e.g., $e \in (A \times B \times C)$].

The difference between lists and tuples in Python, which has implications for their usage, can be captured as follows. Tuples are a data structure whose fields are unordered and have different meanings, such that they typically have different types. Lists, by contrast, are ordered sequences of elements, typically of the same type. For instance, a tuple is an appropriate data structure for storing an employee record containing id, name, rate, and a designation of promotion or not. In turn, a company can be represented by a list of these employee tuples ordered by employment date:

```
>>> [(1, "Mary", 7.75, True), (2, "Linus", 5.75, False),
...  (2, "Lucia", 10.25, True)]
[(1, 'Mary', 7.75, True), (2, 'Linus', 5.75, False),
 (2, 'Lucia', 10.25, True)]
```

It would not be possible or practical to represent this company database as a tuple of lists. Also, note that Python lists are mutable while Python tuples are immutable. Thus, tuples in Python are like lists in Scheme. For example, we could add and remove employees from the company list, but we could not change the rate of an employee. Elements of a tuple are accessed in the same way as elements of a list:

```
>>> (1, "Mary", 7.75, True)[0]
1

>>> (1, "Mary", 7.75, True)[1]
'Mary'
```

Tuples and lists can also be unpacked into multiple bindings:

```
>>> one, two, three = (1, 2, 3)
>>> two
2
>>> three
3
```

Although this situation is rare, the need might arise for a tuple with only one element. Suppose we tried to create a tuple this way:

```
>>> (1)
1
>>> ("Mary")
'Mary'
```

The expression `(1)` does not evaluate to a tuple; instead, it evaluates to the integer `1`. Otherwise, this syntax would introduce ambiguity with parentheses in mathematical expressions. However, Python does have a syntax for making a tuple with only one element—insert a comma between the element and the closing parenthesis:

```
>>> (1,)
(1,)
>>> ("Mary",)
('Mary',)
```

If a function appears to return multiple values, it actually returns a single tuple containing those values.

A.7 User-Defined Functions

A.7.1 Simple User-Defined Functions

The following are some simple user-defined functions:

```
1 >>> def square(x):
2 ...     return x*x;
3 ...
4 >>> def add(x,y):
5 ...     return x+y
6 ...
7 >>> square (4)
8 16
9 >>> square (4.4)
10 19.360000000000003
11 >>> square (True)
12 1
13 >>> add (3,4)
14 7
15 >>> add (3.1,4.2)
16 7.300000000000001
17 >>> add (True,False)
18 1
```

When defining functions at the *read-eval-print* loop as shown here, a blank line is required to denote the end of a function definition (lines 3 and 6).

A.7.2 Positional Vis-à-Vis Keyword Arguments

In defining a function in Python, the programmer must decide how they want the caller to assign argument values to parameters: by position (as in C), by keyword, or by a mixture of both. Readers with imperative programming experience are typically familiar with positional argument values, which are not prefaced with keywords, and where order matters. Let us consider keyword arguments and mixtures of both positional and keyword arguments.

- **Keyword arguments:** There are two types of keyword arguments: named and unnamed.
- **named keyword arguments:** The advantage of keyword arguments is that they need not conform to a strict order (as prescribed by a functional signature using positional arguments), and can have default values:

```
>>> def pizza(size="medium", topping="none", crust="thin"):
...     print("Size: " + size, end=', ')
...     print("Topping: " + topping, end=', ')
...     print("Crust: " + crust + ".")
...
>>> pizza(topping="onions", crust="thick", size="large")
Size: large, Topping: onions, Crust: thick.

>>> pizza(crust="thick", size="large", topping="onions")
Size: large, Topping: onions, Crust: thick.

>>> pizza(crust="thick", size="large")
Size: large, Topping: none, Crust: thick.
```

Note that order matters if you omit the keyword in the call:

```
>>> pizza("large", crust="thick", topping="onions")
Size: large, Topping: onions, Crust: thick.

>>> pizza("thick", "large", "onions")
Size: thick, Topping: large, Crust: onions.
```

- **unnamed keyword arguments:** Unnamed keyword arguments are supplied to the function in the same way as named keyword arguments (i.e., as key–value pairs), but are available in the body of the function as a dictionary:

```
>>> def pizza(**kwargs):
...     print(str(kwargs))
...
...     if 'size' in kwargs:
...         print("Size: " + kwargs['size'], end=', ')
...
...     if 'topping' in kwargs:
```



```

...     print("Topping: " + kwargs['topping'], end=', ')
...
...     if 'crust' in kwargs:
...         print("Crust: " + kwargs['crust'] + ".")
...
>>> pizza(topping="onions", crust="thick", size="large")
{'topping': 'onions', 'crust': 'thick', 'size': 'large'}
Size: large, Topping: onions, Crust: thick.

>>> pizza(crust="thick", size="large")
{'crust': 'thick', 'size': 'large'}
Size: large, Crust: thick.

>>> pizza(crust="thick", size="large", topping="onions")
{'topping': 'onions', 'crust': 'thick', 'size': 'large'}
Size: large, Topping: onions, Crust: thick.

```

Unnamed keyword arguments in Python are similar to variable argument lists in C:

```

void f(int nargs, ...) {
/* the declaration ... can only appear at
   the end of an argument list */

    int i, tmp;

    va_list ap;                /* argument pointer */

    va_start(ap, nargs);       /* initializes ap to point to
                                the first unnamed argument;
                                va_start must be called once
                                before ap can be used */

    for (i=0; i < nargs; i++)
        tmp = va_arg(ap, int); /* returns one argument and
                                steps ap to the next
                                argument */

                                /* the second argument to
                                va_arg must be a type
                                name so that va_args
                                knows how big a step
                                to take */

    va_end(ap);                /* clean-up; must be called
                                before function returns */
}

```

- **Mixture of positional and named keyword arguments:**

```

>>> def pizza(size, topping, crust="thin"):
...     print("Size: " + size, end=', ')
...     print("Topping: " + topping, end=', ')
...     print("Crust: " + crust + ".")
...
>>> pizza("large", "onions")
Size: large, Topping: onions, Crust: thin.

>>> pizza("large", "onions", crust="thick")
Size: large, Topping: onions, Crust: thick.

```

```
>>> pizza(crust="thick", "large", "onions")
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Note that the keyword arguments must be listed after all of the positional arguments in the argument list.

- **Mixture of positional and unnamed keyword arguments:**⁵

```
>>> import sys
>>> def pizza(size, topping, **kwargs):
...     print("Size: " + size, end=', ')
...     print("Topping: " + topping, end=', ')
...     print("Other options:", end=' ')
...
...     printed=False
...
...     if kwargs:
...         for key, value in kwargs.items():
...             if printed:
...                 print(", ", end='')
...                 sys.stdout.write(key)
...                 sys.stdout.write(' :')
...                 sys.stdout.write(value)
...                 printed=True
...             print(".")
...     else:
...         sys.stdout.write('None.\n')
...
>>> pizza("large", "onions")
Size: large, Topping: onions, Other options: None.

>>> pizza("large", "onions", crust="thick")
Size: large, Topping: onions, Other options: crust: thick.

>>> pizza("large", "onions", crust="thick", pickup="no",
...     coupon="yes")
Size: large, Topping: onions, Other options: coupon: yes,
pickup: no, crust: thick.
```

Other Related Notes

- If the arguments to a function are not available individually, they can be passed to a function in a list whose identifier is prefaced with a `*` (line 7):

```
1 >>> def add(x,y):
2     ...     return x+y
3     ...
4 >>> add(3,7)
5 10
6 >>> args = [3,7]
7 >>> add(*args)
8 10
```

5. We use `sys.stdout.write` here rather than `print` to suppress a space from being automatically written between arguments to `print`.

- Python supports *function annotations*, which, while optional, allow the programmer to associate arbitrary Python expressions with parameters and/or return value at compile time.
- Python does not support traditional *function overloading*. When a programmer defines a function a second time, albeit with a new argument list, the second definition fully replaces the first definition rather than providing an alternative, overloaded definition.

A.7.3 Lambda Functions

Lambda functions (i.e., anonymous or literal functions) are introduced with `lambda`. They are often used, as in other languages, in concert with higher-order functions including `map`, which is built into Python as in Scheme:

```
>>> square = lambda x: x*x
>>>
>>> add = lambda x,y: x+y
>>>
>>> inc = lambda n: n+1
>>>
>>> square (4)
16
>>> square (4.4)
19.360000000000003
>>> square (True)
1
>>> add (3,4)
7
>>> add (3.1,4.2)
7.300000000000001
>>> add (True,False)
1
>>> inc (5)
6
>>> inc (5.1)
6.1
>>>
>>> map(inc, [1,2,3])
<map object at 0x10b5a77f0>
>>> list(map(inc, [1,2,3]))
[2, 3, 4]
>>> [i for i in map(inc, [1,2,3])]
[2, 3, 4]
>>>
>>> map(lambda n: n+1, [1,2,3])
<map object at 0x10b5a77f0>
>>> list(map(lambda n: n+1, [1,2,3]))
[2, 3, 4]
>>> [i for i in map(lambda n : n+1, [1,2,3])]
[2, 3, 4]

>>> type(lambda x: x*x)
<type 'function'>
```

These Python functions are the analogs of the following Scheme functions:

```
> (define square (lambda (x) (* x x)))
> (define add (lambda (x y) (+ x y)))
> (define inc (lambda (n) (+ n 1)))

> (square 4)
16

> (square 4.4)
19.360000000000003

> (add 3 4)
7

> (add 3.1 4.2)
7.300000000000001

> (inc 5)
6

> (inc 5.1)
6.1

> (map inc '(1 2 3))
'(2 3 4)

> (map (lambda (n) (+ n 1)) '(1 2 3))
'(2 3 4)

> (map inc '(1 2 3))
(2 3 4)

> (map (lambda (x) (+ n 1)) '(1 2 3))
(2 3 4)
```

Anonymous functions are often used as arguments to higher-order functions (e.g., `map`) and are, hence, helpful. Python also supports the higher-order functions `filter` and `reduce`.

A.7.4 Lexical Closures

Python supports both *first-class functions* and *first-class closures*:

```
>>> def f(x):
...     return lambda y: x+y
...
>>> add5 = f(5)
>>> add6 = f(6)
>>>
>>> add5
<function f.<locals>.<lambda> at 0x10bd3b700>
>>> add6
<function f.<locals>.<lambda> at 0x10bd3b790>
>>>
>>> add5(2)
```

```
7
>>> add6(2)
8
```

For more information, see Section 6.10.2.

A.7.5 More User-Defined Functions

gcd

```
>>> def gcd(u,v):
...     if v == 0:
...         return u
...     else:
...         return gcd(v, (u % v))
...
>>> gcd (16,32)
16
```

factorial

```
>>> def factorial(n):
...     if n == 0:
...         return 1
...     else:
...         return n * factorial(n-1)
...
>>> factorial (5)
120
```

fibonacci

```
>>> def fibonacci(n):
...     if (n == 0) or (n == 1):
...         return 1
...     else:
...         return fibonacci(n-1) + fibonacci(n-2)
...
>>> fibonacci (0)
1
>>> fibonacci (1)
1
>>> fibonacci (2)
2
>>> fibonacci (3)
3
>>> fibonacci (4)
5
>>> fibonacci (5)
8
```

reverse

```

>>> def reverse (lst):
...     if lst == []:
...         return []
...     else:
...         return (reverse (lst[1:]) + [lst[0]])
...
>>> reverse ([])
[]
>>> reverse ([1])
[1]
>>> reverse ([1,2,3,4,5])
[5, 4, 3, 2, 1]
>>> reverse ([1,2,3,4,5,6,7,8,9,10])
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> reverse ([10,9,8,7,6,5,4,3,2,1])
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>
>>> reverse (["my","dear","grandmother","Lucia"])
['Lucia', 'grandmother', 'dear', 'my']
>>> reverse (["Lucia", "grandmother", "dear", "my"])
['my', 'dear', 'grandmother', 'Lucia']

```

Note that `reverse` can reverse a list containing values of any type.

member

Consider the following definition of a list `member` function in Python:

```

>>> def member(e, lst):
...     if lst == []:
...         return False
...     else:
...         return (lst[0] == e) or member(e, lst[1:])
...
>>> member (1, [1,2,3,4])
True
>>> member (2, [1,2,3,4])
True
>>> member (3, [1,2,3,4])
True
>>> member (4, [1,2,3,4])
True
>>> member (0, [1,2,3,4])
False
>>> member (5, [1,2,3,4])
False
>>>
>>> # "in" is the Python member operator
>>> 1 in [1,2,3,4]
True
>>> 2 in [1,2,3,4]
True
>>> 3 in [1,2,3,4]
True
>>> 4 in [1,2,3,4]
True
>>> 0 in [1,2,3,4]
False

```

```
False
>>> 5 in [1,2,3,4]
False
```

A.7.6 Local Binding and Nested Functions

A local variable in Python can be used to introduce local binding for the purposes of avoiding recomputation of common subexpressions and creating nested functions for both protection and factoring out (so as to avoid passing and copying) arguments that remain constant between recursive function calls.

Local Binding

```
>>> def insertineach(item, lst):
...     if lst == []:
...         return []
...     else:
...         return ([[item] + lst[0]] + insertineach(item, lst[1:]))
...
>>> def powerset(lst):
...     if lst == []:
...         return [[]]
...     else:
...         temp = powerset(lst[1:])
...         return (insertineach (lst[0], temp) + temp)
...
>>> insertineach (1, [])
[]
>>> insertineach (1, [[2,3], [4,5], [6,7]])
[[1, 2, 3], [1, 4, 5], [1, 6, 7]]
>>>
>>> powerset ([])
[[]]
>>> powerset ([1])
[[1], []]
>>> powerset ([1,2])
[[1, 2], [1], [2], []]
>>> powerset ([1,2,3])
[[1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []]
>>> powerset (["a","b","c"])
[['a', 'b', 'c'], ['a', 'b'], ['a', 'c'], ['a'], ['b', 'c'], ['b'],
['c'], []]
```

These functions are the Python analogs of the following Scheme functions:

```
(define (insertineach item l)
  (cond
    ((null? l) '())
    (else (cons (cons item (car l))
                  (insertineach item (cdr l))))))

(define (powerset l)
  (cond
    ((null? l) '())
    (else
     (let ((y (powerset (cdr l))))
       (append (insertineach (car l) y) y)))))
```

Nested Functions

Since the function `insertineach` is intended to be only visible within, accessible within, and called by the `powerset` function, we can nest it within the `powerset` function:

```
>>> def powerset(lst):
...     def insertineach(item,lst):
...         if lst == []:
...             return []
...         else:
...             return ([[item] + lst[0]] + insertineach(item,lst[1:]))
...     if lst == []:
...         return [[]]
...     else:
...         temp = powerset(lst[1:])
...         return (insertineach (lst[0], temp) + temp)
...
>>> powerset ([])
[[]]
>>> powerset ([1])
[[1], []]
>>> powerset ([1,2])
[[1, 2], [1], [2], []]
>>> powerset ([1,2,3])
[[1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []]
>>> powerset (["a","b","c"])
[['a', 'b', 'c'], ['a', 'b'], ['a', 'c'], ['a'], ['b', 'c'], ['b'],
 ['c'], []]
```

The following is an example of using a nested function within the definition of a reverse function:

```
>>> # nesting rev within reverse to hide and protect it
>>> def reverse(lst):
...     def rev (lst,m):
...         if lst == []:
...             return m
...         else:
...             return rev (lst[1:], [lst[0]]+m)
...     return rev (lst,[])
...
>>> reverse ([])
[]
>>> reverse ([1])
[1]
>>> reverse ([1,2,3,4,5])
[5, 4, 3, 2, 1]
>>> reverse ([1,2,3,4,5,6,7,8,9,10])
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> reverse ([10,9,8,7,6,5,4,3,2,1])
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> reverse (["my","dear","grandmother","Lucia"])
['Lucia', 'grandmother', 'dear', 'my']
>>> reverse (["Lucia", "grandmother", "dear", "my"])
['my', 'dear', 'grandmother', 'Lucia']
```


A.7.7 Mutual Recursion

Unlike ML, but like Scheme and Haskell, Python allows a function to call a function that is defined below it:

```
>>> def f(x,y):
...     return square(x+y)
...
>>> def square(x):
...     return x*x
...
>>> f(3,4)
49
```

This makes the definition of *mutually recursive* functions straightforward. For instance, consider the functions `iseven` and `isodd`, which rely on each other to determine if an integer is even or odd, respectively:

```
>>> def isodd(n):
...     if n == 1:
...         return True
...     elif n == 0:
...         return False
...     else:
...         return iseven(n-1)
...
>>> def iseven(n):
...     if n == 0:
...         return True
...     else:
...         return isodd(n-1)
...
>>> isodd(9)
True
>>> isodd(100)
False
>>> iseven(100)
True
```

Note that more than two mutually recursive functions can be defined.

A.7.8 Putting It All Together: Mergesort

Consider the following definitions of a mergesort function.

Unnested, Unhidden, Flat Version

```
def split (lat):
    if lat == []:
        return ([], [])
    elif len(lat) == 1:
        return ([], lat)
    else:
        (left, right) = split (lat[2:])
        return ([lat[0]] + left, [lat[1]] + right)
```

```

def merge(left, right):
    if left == []:
        return right
    elif right == []:
        return left
    elif left[0] < right[0]:
        return [left[0]] + merge(left[1:], right)
    else:
        return [right[0]] + merge(right[1:], left)

def mergesort (lat):
    if lat == []:
        return []
    elif len(lat) == 1:
        return lat
    else:
        # split it
        (left, right) = split(lat)

        # mergesort each side
        leftsorted = mergesort(left)
        rightsorted = mergesort(right)

        return merge(leftsorted, rightsorted)

print(mergesort ([9,8,7,6,5,4,3,2,1]))
$
$ python mergesort.py
[1, 2, 3, 4, 5, 6, 7, 8, 9]

```

Nested, Hidden Version

```

def mergesort (lat):

    def split (lat):

        if lat == []:
            return ([], [])
        elif len(lat) == 1:
            return ([], lat)
        else:
            (left, right) = split (lat[2:])
            return ([lat[0]] + left, [lat[1]] + right)

    def merge(left, right):
        if left == []:
            return right
        elif right == []:
            return left
        elif left[0] < right[0]:
            return [left[0]] + merge(left[1:], right)
        else:
            return [right[0]] + merge(right[1:], left)

    if lat == []:
        return []
    elif len(lat) == 1:
        return lat
    else:

```

```

    # split it
    (left, right) = split(lat)

    # mergesort each side
    leftsorted = mergesort(left)
    rightsorted = mergesort(right)

    return merge(leftsorted, rightsorted)

print(mergesort ([9,8,7,6,5,4,3,2,1]))
$
$ python mergesort.py
[1, 2, 3, 4, 5, 6, 7, 8, 9]

```

Nested, Hidden Version Accepting a Comparison Operator as a Parameter

```

$ cat mergesort.py
import operator

def mergesort (compop,lat):

    def split (lat):

        if lat == []:
            return ([], [])
        elif len(lat) == 1:
            return ([], lat)
        else:
            (left, right) = split (lat[2:])
            return ([lat[0]] + left, [lat[1]] + right)

    def merge(compop,left, right):
        if left == []:
            return right
        elif right == []:
            return left
        elif compop (left[0], right[0]):
            return [left[0]] + merge(compop, left[1:], right)
        else:
            return [right[0]] + merge(compop, right[1:], left)

    if lat == []:
        return []
    elif len(lat) == 1:
        return lat
    else:
        # split it
        (left, right) = split(lat)

        # mergesort each side
        leftsorted = mergesort (compop, left)
        rightsorted = mergesort (compop, right)

        return merge(compop, leftsorted, rightsorted)

print(mergesort (operator.lt, [9,8,7,6,5,4,3,2,1]))
print(mergesort (operator.gt, [1,2,3,4,5,6,7,8,9]))
$
$ python mergesort.py

```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Final Version

The following is the final version of `mergesort` using nested, protected functions and accepting a comparison operator as a parameter that is factored out to avoid passing it between successive recursive calls. We also use a keyword argument for the comparison operator:

```
import operator

def mergesort (lat, compop=operator.lt):

    def mergesort1 (lat):

        def split (lat):

            if lat == []:
                return ([], [])
            elif len(lat) == 1:
                return ([], lat)
            else:
                (left, right) = split (lat[2:])
                return ([lat[0]] + left, [lat[1]] + right)

        def merge(left, right):
            if left == []:
                return right
            elif right == []:
                return left
            elif compop (left[0], right[0]):
                return [left[0]] + merge(left[1:], right)
            else:
                return [right[0]] + merge(right[1:], left)

        if lat == []:
            return []
        elif len(lat) == 1:
            return lat
        else:
            # split it
            (left, right) = split(lat)

            # mergesort each side
            leftsorted = mergesort1(left)
            rightsorted = mergesort1(right)

            return merge(leftsorted, rightsorted)

    return mergesort1(lat)

print(mergesort ([9,8,7,6,5,4,3,2,1]))
print(mergesort ([1,2,3,4,5,6,7,8,9], operator.gt))
$
$ python mergesort.py
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Notice also that we factored the argument `compop` out of the function `merge` in this version, since it is visible from an outer scope.

A.8 Object-Oriented Programming in Python

Recall that we demonstrated (in Section 6.10.2) how to create a first-class counter closure in Python that encapsulates code and state and, therefore, resembles an object. Here we demonstrate how to use the object-oriented facilities in Python to develop a counter object. In both cases, we are binding an object (here, a function or method) to a specific context (or environment). In Python nomenclature, the closure approach is sometimes called *nested scopes*. However, in both approaches the end result is the same—a *callable object* (here, a function) that remembers its context.

```
>>> class new_counter:
...     def __init__(self, initial):
...         self.current = initial
...     def __call__(self):
...         self.current = self.current + 1
...         return self.current
...
>>> counter1 = new_counter(1)
>>> counter2 = new_counter(100)
>>>
>>> counter1
<__main__.new_counter object at 0x10c12b250>
>>> counter2
<__main__.new_counter object at 0x10c0f37f0>
>>>
>>> counter1()
2
>>> counter1()
3
>>> counter2()
101
>>> counter2()
102
>>> counter1()
4
>>> counter1()
5
>>> counter2()
103
```

While the object-oriented approach is perhaps more familiar to those readers from a traditional object-oriented programming background, it executes more slowly due to the object overhead. However, the following approach permits multiple callable objects to share their signature through inheritance:

```
>>> class new_counter:
...     def __init__(self, initial):
...         self.current = initial
...     def __call__(self):
...         self.current = self.current + 1
...         return self.current
```

```

...
>>>
>>> class custom_counter(new_counter):
...     # __init__ is inherited from parent class new_counter
...     def __call__(self, step):
...         self.current = self.current + step
...         return self.current
...
>>> counter1 = custom_counter(1)
>>> counter2 = custom_counter(100)
>>>
>>> counter1(1)
2
>>> counter1(2)
4
>>> counter2(3)
103
>>> counter2(4)
107
>>> counter1(5)
9
>>> counter1(6)
15
>>> counter2(7)
114

```

Notice that the callable object returned is bound to the environment in which it was created. In traditional object-oriented programming, an object encapsulates (or binds) *multiple* functions (called methods) and (to) an (the same) environment. Thus, we can augment the class `new_counter` with additional methods:

```

>>> class new_counter:
...     current = 0
...     def initialize(self, initial):
...         self.current = initial
...     def increment(self):
...         self.current = self.current+1
...     def decrement(self):
...         self.current = self.current-1
...     def get(self):
...         return self.current
...     def write(self):
...         print (self.current)
...
>>> counter1 = new_counter()
>>> counter2 = new_counter()
>>> counter1.initialize(1)
>>> counter2.initialize(100)
>>> counter1.increment()
>>> counter2.increment()
>>> counter1.increment()
>>> counter2.increment()
>>> counter1.increment()
>>> counter2.increment()
>>> counter1.write()
4
>>> counter2.write()
103
>>> counter1.decrement()
>>> counter2.decrement()
>>> counter1.decrement()

```

```
>>> counter2.decrement()
>>> counter1.decrement()
>>> counter2.decrement()
>>> counter1.write()
1
>>> counter2.write()
100
```

A.9 Exception Handling

When an error occurs in a syntactically valid Python program, that error is referred to as an *exception*. Exceptions are immediately fatal when they are unhandled. Exceptions may be raised and caught as a way to affect the control flow of a Python program. Consider the following interaction with Python:

```
>>> divisor = 1
>>> integer = integer / divisor
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'integer' is not defined
```

In executing the syntactically valid second line of code, the interpreter raises a `NameError` because `integer` is not defined before the value of `integer` is used. Because this exception is not handled by programmer, it is fatal. However, the exception may be caught and handled:

```
>>> divisor = 1
>>> try:
...     integer = integer / divisor
... except:
...     print("An exception occurred. Proceeding anyway.")
...     integer = 0
...
An exception occurred. Proceeding anyway.
>>> print(integer)
0
```

This example catches all exceptions that may occur within the `try` block of the exception. The `except` block will execute only if an exception occurs in the `try` block.

Python also permits programmers to catch specific exceptions and define a unique `except` block for each exception:

```
>>> divisor = 1
>>> try:
...     integer = integer / divisor
... except NameError:
...     print("Caught a name error.")
... except ZeroDivisionError:
...     print("Caught a divide by 0 error.")
... except:
...     print("Caught something else.")
```

```
...     print(e)
...
Caught a name error.
```

If a `try` block raises a `NameError` or a `ZeroDivisionError` error, the interpreter executes the corresponding `except` block (and no other `except` block). If any other type of exception occurs, the final `except` block executes.

The `finally` clause may be used to specify a block of code that must run regardless of the exception raised—even if that exception is not caught. If a return value is encountered in the `try` or `except` block, the `finally` block executes before the return occurs:

```
>>> def divide_numbers(integer, divisor):
...     try:
...         integer = integer / divisor
...         return integer
...     except ZeroDivisionError as e:
...         print("Caught a name error.")
...         print("Printing exception: %s" % e)
...         return None
...     finally:
...         print("Hitting the finally block before returning.")
...
>>> print(divide_numbers(39, 2))
Hitting the finally block before returning.
19.5
>>> print(divide_numbers(39, 0))
Caught a name error.
Printing exception: division by zero
Hitting the finally block before returning.
None
```

Lastly, programmers may raise their own exceptions to force an exception to occur:

```
>>> try:
...     raise NameError
... except NameError:
...     print("Caught my own exception!")
...
Caught my own exception!
```

Programming Exercises for Appendix A

Exercise A.1 Define a recursive Python function `remove` that accepts only a list and an integer `i` as arguments and returns another list that is the same as the input list, but with the `i`th element of the input list removed. If the length of the input list is less than `i`, return the same list. Assume that `i = 1` refers to the first element of the list.

Examples:

```
>>> remove(1, [9,10,11,12])
[10,11,12]
```



```
>>> remove(2, [9,10,11,12])
[9,11,12]
>>> remove(3, [9,10,11,12])
[9,10,12]
>>> remove(4, [9,10,11,12])
[9,10,11]
>>> remove(5, [9,10,11,12])
[9,10,11,12]
```

Exercise A.2 Define a recursive Python function called `makeset` without using a `set`. The `makeset` function accepts only a list as input and returns the list with any repeating elements removed. The order in which the elements appear in the returned list does not matter, as long as there are no duplicate elements. Do not use any user-defined auxiliary functions, except `member`.

Examples:

```
>>> makeset([1,3,4,1,3,9])
[4,1,3,9]
>>> makeset([1,3,4,9])
[1, 3, 4, 9]
>>> makeset(["apple", "orange", "apple"])
['orange', 'apple']
```

Exercise A.3 Solve Programming Exercise A.2, but this time use a `set` in your definition. The function must still accept and return a `list`. *Hint:* This can be done in one line of code.

Exercise A.4 Define a recursive Python function `cycle` that accepts only a list and an integer `i` as arguments and cycles the list `i` times. Do not use any user-defined auxiliary functions.

Examples:

```
>>> cycle(0, [1,4,5,2])
[1, 4, 5, 2]
>>> cycle(1, [1,4,5,2])
[4, 5, 2, 1]
>>> cycle(2, [1,4,5,2])
[5, 2, 1, 4]
>>> cycle(4, [1,4,5,2])
[1, 4, 5, 2]
>>> cycle(6, [1,4,5,2])
[5, 2, 1, 4]
>>> cycle(10, [])
[]
>>> cycle(10, [1])
[1]
>>> cycle(9, [1,4])
[4, 1]
```

Exercise A.5 Define a recursive Python function `transpose` that accepts a list as its only argument and returns that list with adjacent elements transposed. Specifically, `transpose` accepts an input list of the form $[e_1, e_2, e_3, e_4, e_5, e_6, \dots, e_n]$

and returns a list of the form $[e_2, e_1, e_4, e_3, e_6, e_5, \dots, e_n, e_{n-1}]$ as output. If n is odd, e_n will continue to be the last element of the list. Do not use any user-defined auxiliary functions.

Examples:

```
>>> transpose([1, 2, 3, 4])
[2, 1, 4, 3]

>>> transpose([1, 2, 3, 4, 5, 6])
[2, 1, 4, 3, 6, 5]

>>> transpose([1, 2, 3])
[2, 1, 3]
```

Exercise A.6 Define a recursive Python function `oddevensum` that accepts only a list of integers as an argument and returns a pair consisting of the sum of the odd and even positions of the list, in that order. Do not use any user-defined auxiliary functions.

Examples:

```
>>> oddevensum([])
(0, 0)
>>> oddevensum([6])
(6, 0)
>>> oddevensum([6, 3])
(6, 3)
>>> oddevensum([6, 3, 8])
(14, 3)
>>> oddevensum([1, 2, 3, 4])
(4, 6)
>>> oddevensum([1, 2, 3, 4, 5, 6])
(9, 12)
>>> oddevensum([1, 2, 3])
(4, 2)
```

Exercise A.7 Define a recursive Python function `member` that accepts only an element and a list of values of the type of that element as input and returns `True` if the item is in the list and `False` otherwise. Do not use `in` within the definition of your function. *Hint:* This can be done in one line of code.

Exercise A.8 Define a recursive Python function `permutations` that accepts only a list representing a set as an argument and returns a list of all permutations of that list as a list of lists. You will need to define some nested auxiliary functions. Pass a λ -function to `map` where applicable in the bodies of the functions to simplify their definitions.

Examples:

```
>>> permutations([])
[]

>>> permutations([1])
```

```

[[1]]

>>> permutations([1,2])
[[1,2],[2,1]]

>>> permutations([1,2,3])
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

>>> permutations([1,2,3,4])
[[1,2,3,4],[1,2,4,3],[1,3,2,4],[1,3,4,2],[1,4,2,3],
 [1,4,3,2],[2,1,3,4],[2,1,4,3],[2,3,1,4],[2,3,4,1],
 [2,4,1,3],[2,4,3,1],[3,1,2,4],[3,1,4,2],[3,2,1,4],
 [3,2,4,1],[3,4,1,2],[3,4,2,1],[4,1,2,3],[4,1,3,2],
 [4,2,1,3],[4,2,3,1],[4,3,1,2],[4,3,2,1]]

>>> permutations(["oranges", "and", "tangerines"])
[["oranges","and","tangerines"], ["oranges","tangerines","and"],
 ["and","oranges","tangerines"], ["and","tangerines","oranges"],
 ["tangerines","oranges","and"], ["tangerines","and","oranges"]]

```

Hint: This solution requires less than 25 lines of code.

Exercise A.9 Reimplement the `mergesort` function in Section A.7.8 using an imperative style of programming. Specifically, eliminate the nested `split` function and define the nested `merge` function non-recursively. Implement the following four progressive versions as demonstrated in Section A.7.8:

- (a) Unnested, Unhidden, Flat Version
- (b) Nested, Hidden Version
- (c) Nested, Hidden Version Accepting a Comparison Operator as a Parameter
- (d) Final Version

A.10 Thematic Takeaway

Because of the multiple styles of programming it supports (e.g., imperative, object-oriented, and functional), Python is a worthwhile vehicle through which to explore language concepts, including lexical closures, lambda functions, list comprehensions, dynamic type systems, and automatic memory management.

A.11 Appendix Summary

This appendix provides an introduction to Python so that readers can explore concepts of programming languages through Python in this text, but especially in Chapters 10–12. Python is dynamically typed, and blocks of source code in Python are demarcated through indentation. Python supports heterogeneous lists, and the

+ operator appends two lists. Python supports *anonymous*/ λ *functions* and both *positional* and *named keyword arguments* to functions.

A.12 Notes and Further Reading

For Peter Norvig's comparison of Python and Lisp along a variety of language concepts and features, we refer readers to <https://norvig.com/python-lisp.html>.

Appendix B

Introduction to ML

I ...picked up the utility of giving students a fast overview, stressing the most commonly used constructs [and idioms] rather than the complete syntax. ...In writing this [short] guide to ML programming, I have thus departed from the approach found in many books on the language. I tried to remember how things struck me at first, the analogies I drew with conventional languages, and the concepts I found most useful in getting started.

— Jeffrey D. Ullman, *Elements of ML Programming* (1997)

ML is a statically typed and type-safe programming language that primarily supports functional programming, but has some imperative features.

B.1 Appendix Objective

Establish an understanding of the syntax and semantics of ML through examples so that a reader familiar with the essential elements of functional programming after having read this appendix can write intermediate programs in ML.

B.2 Introduction

ML (historically, MetaLanguage) is, like Scheme, a language supporting primarily functional programming with some imperative features. It was developed by A. J. Robin Milner and others in the early 1970s at the University of Edinburgh. ML is a general-purpose programming language in that it incorporates functional features from Lisp, rule-based programming (i.e., pattern matching) from Prolog, and data abstraction from Smalltalk and C++. ML is an ideal vehicle through which to explore the language concepts of type safety, type inference, and currying. The objective here, however, is elementary programming in ML. ML also, like Scheme, is statically scoped. We leave the use of ML to explore these language concepts to the main text.

This appendix is an example-oriented avenue to get started with ML programming and is intended to get a programmer already familiar with the essential tenets of functional programming (Chapter 5) writing intermediate programs in ML; it is not intended as an exhaustive tutorial or comprehensive reference. The primary objective of this appendix is to establish an understanding of ML programming in readers already familiar with the essential elements of functional programming in preparation for the study of typing and type inference (in Chapter 7), currying and higher-order functions (in Chapter 8), and type systems (in Chapter 9)—concepts that are both naturally and conveniently explored through ML. This appendix should be straightforward for anyone familiar with functional programming in another language, particularly Scheme. We sometimes compare ML expressions to their analogs in Scheme.

We use the Standard ML dialect of ML, and the Standard ML of New Jersey implementation of ML in this text. The original version of ML theoretically expressed by Milner in 1978 used a slightly different syntax than Standard ML and lacked pattern matching. Note that `-` is the prompt for input in the Standard ML of New Jersey interpreter used in this text. A goal of the functional style of programming is to bring programming closer to mathematics. In this appendix, ML and its syntax as well as the responses of the ML interpreter make the connection between functional programming and mathematics salient.

B.3 Primitive Types

ML has the following primitive types: integer (`int`), real (`real`), boolean (`bool`), character (`char`), and string (`string`):

```
1  - 3;
2  val it = 3 : int
3  - 3.33;
4  val it = 3.3 : real
5  - true;
6  val it = true : bool
7  - #"a";
8  val it = #"a" : char
9  - "hello world";
10 val it = "hello world" : string
```

Notice that ML uses *type inference*. The `:` colon symbol associates a value with a type and is read as “is of type.” For instance, the expression `3 : int` indicates that 3 is of type `int`. This explains the responses of the interpreter on lines 2, 4, 6, 8, and 10 when an expression is entered on the preceding line.

B.4 Essential Operators and Expressions

- **Character conversions.** The `ord` and `chr` functions are used for character conversions:

```

- ord("#a");
val it = 97 : int
- chr(97);
val it = "#a" : char
- chr(ord("#a"));
val it = "#a" : char

```

- **String concatenation.** The `^` append operator is used for string concatenation:

```

- "hello" ^ " " ^ "world"
val it = "hello world" : string

```

- **Arithmetic.** The infix binary¹ operators `+`, `-`, and `*` only accept two values of type `int` or two values of type `real`; the prefix unary minus operator `~` accepts a value of type `int` or `real`; the infix binary division operator `/` only accepts two values of type `real`; the infix binary division operator `div` only accepts two values of type `int`; and the infix binary modulus operator `mod` only accepts two values of type `int`.

```

- 4.2 / 2.1;
val it = 2.0 : real

- 4 div 2;
val it = 2 : int

- ~1;
val it = ~1 : int

```

- **Comparison.** The infix binary operators `=` (equal to), `<`, `>`, `<=`, `>=`, and `<>` (not equal to) compare `ints`, `reals`, `chars`, or `strings` with one exception: `reals` may not be compared using `=` or `<>`. Instead, use the prefix functions `Real.==` and `Real.!=`. For now, we can think of `Real` as an object (in an object-oriented program), `==` as a message, and the expression `Real.==` as sending the message `==` to the object `Real`, which in turn executes the method definition of the message. `Real` is called a *structure* in ML (Section B.10). Structures are used again in Section B.12.

```

- 4 = 2;
val it = false : bool
- 4 > 2;
val it = true : bool
- 4 <> 2;
val it = true : bool
- Real.==(2.1, 4.1);
val it = false : bool
- Real.!=(4.1, 2.1);
val it = true : bool

```

1. Technically, all operators in ML are unary operators, in that each accepts a single argument that is a pair. However, generally, though not always, there is no problem interpreting a unary operator that only accepts a single pair as a binary operator.

- **Boolean operators.** The infix operators `orelse`, `andalso` (not to be confused with `and`), and `not` are the *or*, *and*, and *not* boolean operators with their usual semantics. The operators `orelse` and `andalso` use *short-circuit evaluation* (or *lazy evaluation*, as discussed in Chapter 12):

```
- true orelse false;
val it = true : bool
- false andalso false;
val it = false : bool
- not false;
val it = true : bool
```

- **Conditionals.** Use `if-then-else` expressions:

```
- if 1 <> 2 then "true branch" else "false branch";
val it = "true branch" : string
```

There is no `if` expression without an `else` because all expressions must return a value.

- **Comments.**

```
(* this is a comment single-line comment and *)
(* this is
   a
      multi-line
comment *)
```

- The `explode` and `implode` functions:

```
- explode;
val it = fn : string -> char list
- explode("apple");
val it = [#"a",#"p",#"p",#"l",#"e"] : char list
- implode;
val it = fn : char list -> string
- implode([#"a", #"p", #"p", #"l", #"e"]);
val it = "apple" : string
- implode(explode("apple"));
val it = "apple" : string
```

B.5 Running an ML Program

(Assuming a UNIX environment.)

- Enter `sml` at the command prompt and enter expressions interactively to evaluate them:

```
$ sml
Standard ML of New Jersey (64-bit) v110.98
- 2 + 3;
- val it = 5 : int
- ^D
$
```

Using this method of execution, the programmer can define new functions at the prompt of the interpreter:

```
- fun f(x) = x + 1;
val f = fn : int -> int
- f(1);
val it = 2 : int
```

Use the EOF character (which is `<ctrl-d>` on UNIX systems and `<ctrl-z>` on Windows systems) to exit the interpreter.

- Enter `sml <filename>.sml` from the command prompt using file I/O, which causes the program in `<filename>.sml` to be evaluated:

```
0 $ cat first.sml
1
2 2 + 3;
3
4 fun inc(x) = x + 1;
5
6 $ sml first.sml
7 Standard ML of New Jersey (64-bit) v110.98
8 [opening first.sml]
9 val it = 5 : int
10 val f = fn : int -> int
11 -
12 - f(1);
13 val it = 2 : int
14 -
```

After the program is evaluated, the read-eval-print loop is available to the programmer as shown on line 14.

- Enter `sml` at the command prompt and load a program by entering `use "<filename>.sml";` into the read-eval-print prompt (line 9):

```
0 $ cat first.sml
1
2 2 + 3;
3
4 fun inc(x) = x + 1;
5
6 $ sml
7 Standard ML of New Jersey (64-bit) v110.98
8 -
9 - use "first.sml";
10 [opening first.sml]
11 val it = 5 : int
12 val it = () : unit
13 -
14 - inc(1);
15 val it = 2 : int
16 -
```

- Redirect standard input into the interpreter from the keyboard to a file by entering `sml <<filename>.sml` at the command prompt:²

2. The interpreter automatically exits once EOF is reached and evaluation is complete.

```
$ sml < a.sml
Standard ML of New Jersey (64-bit) v110.98
- val it = 5 : int
val f = fn : int -> int
-
$
```

B.6 Lists

The following are the some important points about lists in ML.

- Unlike in Scheme, lists in ML are *homogeneous*, meaning all elements of the list must be of the same type. For instance, the list `[1, 2, 3]` in ML is homogeneous, while the list `(1 "apple")` in Scheme is heterogeneous.
- In a type-safe language like ML the values in a tuple (Section B.7) generally have different types, but the number of elements in the tuple must be fixed. Conversely, the values of a list must all have the same type, but the number of elements in the list is not fixed.
- The semantics of the lexemes `nil` and `[]` are the empty list.
- The cons operator, which accepts an element (the head) and a list (the tail), is `::` (e.g., `1 :: 2 :: [3]`) and associates right-to-left.
- The expression `x :: xs` represents a list of *at least one element*.
- The expression `xs` is pronounced *exes*.
- The expression `x :: nil` represents a list of *exactly one element* and is the same as `[x]`.
- The expression `x :: y :: xs` represents a list of *at least two elements*.
- The expression `x :: y :: nil` represents a list of *exactly two elements*.
- The built-in functions `hd` (for head) and `tl` (for tail) are the ML analogs of the Scheme functions `car` and `cdr`, respectively.
- The built-in function `length` returns the number of elements in its only list argument.
- The append operator (`@`) accepts two lists and appends them to each other. For example, `[1, 2] @ [3, 4, 5]` returns `[1, 2, 3, 4, 5]`. The append operator in ML is also inefficient, just as it is in Scheme.

Examples:

```
- [1, 2, 3];
val it = [1, 2, 3] : int list
- nil;
val it = [] : 'a list
- [];
val it = [] : 'a list
- 1 :: 2 :: [3];
val it = [1, 2, 3] : int list
- 1 :: nil;
val it = [1] : int list
- 1 :: [];
val it = [1] : int list
```

```

- 1::2::nil;
val it = [1,2] : int list
- hd(1::2::[3]);
val it = 1 : int
- tl(1::2::[3]);
val it = [2,3] : int list
- hd([1,2,3]);
val it = 1 : int
- tl([1,2,3]);
val it = [2,3] : int list
- [1,2,3]@[4,5,6];
val it = [1,2,3,4,5,6] : int list

```

B.7 Tuples

A *tuple* is a sequence of elements of potentially mixed types. Formally, a tuple is an element e of a Cartesian product of a given number of sets: $e \in (S_1 \times S_2 \times \cdots \times S_n)$. A two-element tuple is called a *pair* [e.g., $e \in (A \times B)$]. A three-element tuple is called a *triple* [e.g., $e \in (A \times B \times C)$]. A tuple typically contains unordered, *heterogeneous* elements akin to a `struct` in C with the exception that a tuple is indexed by numbers (like a list) rather than by field names (like a `struct`). While tuples can be heterogeneous, in a list of tuples, each tuple in the list must be of the same type. Elements of a tuple are accessible by prefacing the tuple with $\#n$, where n is the number of the element, starting with 1:

```

1 - (1, "Mary", 3.76)
2 val it = (1,"Mary",3.76) : int * string * real
3 - #2((1,"Mary",3.76))
4 "Mary"

```

The response from the interpreter when `(1, "Mary", 3.76)` (line 1) is entered is `(1,"Mary",3.76) : int * string * real` (line 2). This response indicates that the tuple `(1, "Mary", 3.76)` consists of an instance of type `int`, an instance of type `string`, and an instance of type `real`. The response from the interpreter when a tuple is entered (e.g., `int * string * real`) demonstrates that a tuple is an element of a Cartesian product of a given number of sets. Here, the `*`, which is not intended to mean multiplication, is the analog of the Cartesian-product operator $\times$, and the data types are the sets involved in the Cartesian product. In other words, `int * string * real` is a type defined by the Cartesian product of the set of all `ints`, the set of all `strings`, and the set of all `reals`. An element of the Cartesian product of the set of all `ints`, the set of all `strings`, and the set of all `reals` has the type `int * string * real`:

$$(1, \text{"Mary"}, 3.76) \in (\text{int} \times \text{string} \times \text{real})$$

The argument list of a function in ML, described in Section B.8, is a tuple. Therefore, ML uses tuples to specify the domain of a function.

B.8 User-Defined Functions

A key language concept in ML is that all functions have types.

B.8.1 Simple User-Defined Functions

Named functions are introduced with `fun`:

```
- fun square(x) = x*x;
val square = fn : int -> int
- fun add(x,y) = x+y;
val add = fn : int * int -> int
```

Here, the type of `square` is a function `int -> int` or, in other words, a function that maps an `int` to an `int`. Similarly, the type of `add` is a function `int * int -> int` or, in other words, a function that maps a tuple of type `int * int` to an `int`. Notice that the interpreter prints the domain of a function that accepts more than one parameter as a Cartesian product using the notation described in Section B.7. These functions are the ML analogs of the following Scheme functions:

```
(define (square x) (* x x))
(define (add x y) (+ x y))
```

Notice that the ML syntax involves fewer lexemes than Scheme (e.g., `define` is not included). Without excessive parentheses, ML is also more readable than Scheme.

B.8.2 Lambda Functions

Lambda functions (i.e., anonymous or literal functions) are introduced with `fn`. They are often used, as in other languages, in concert with higher-order functions including `map`, which is built into ML as in Scheme:

```
- (fn (n) => n+1) (5)
val it = 6 : int
- map (fn (n) => n+1) [1,2,3];
val it = [2,3,4] : int list
```

These expressions are the ML analogs of the following Scheme expressions:

```
> ((lambda (n) (+ n 1)) 5)
6
> (map (lambda (n) (+ n 1)) '(1 2 3))
(2 3 4)
```

Moreover, the functions

```
- val add = fn (x,y) => x+y;
val add = fn : int * int -> int
- val square = fn (x) => x*x;
val square = fn : int -> int
```

are the ML analogs of the following Scheme functions:

```
(define add (lambda (x y) (+ x y)))
(define square (lambda (x) (* x x)))
```

Anonymous functions are often used as arguments to higher-order functions.

B.8.3 Pattern-Directed Invocation

A key feature of ML is its support for the definition and invocation of functions using a pattern-matching mechanism called *pattern-directed invocation*. In pattern-directed invocation, the programmer writes multiple definitions of the same function. When that function is called, the determination of the particular definition of the function to be executed is made based on pattern matching the arguments passed to the function with the patterns used as parameters in the signature of the function. For instance, consider the following definitions of a greatest common divisor function:

```
1 - (* first version without pattern-directed invocation *)
2 - fun gcd(u,v) = if v = 0 then u else gcd(v, (u mod v));
3 val gcd = fn : int * int -> int
4
5 - (* second version with pattern-directed invocation *)
6 - fun gcd(u,0) = u
7   = |   gcd(u,v) = gcd(v, (u mod v));
8 val gcd = fn : int * int -> int
```

The first version (defined on line 2) does not use pattern-directed invocation; that is, there is only one definition of the function. The second version (defined on lines 6–7) uses pattern-directed invocation. If the literal 0 is passed as the second argument to the function `gcd`, then the first definition of `gcd` is used (line 6); otherwise, the second definition (line 7) is used.

Pattern-directed invocation is not identical to operator/function overloading. Overloading involves determining which definition of a function to invoke based on the number and types of arguments it is passed at run-time. With pattern-directed invocation, no matter how many definitions of the function exist, all have the same type signature (i.e., number and type of parameters).

Native support for pattern-directed invocation is one of the most convenient features of user-defined functions in ML because it obviates the need for an `if-then-else` expression to differentiate between the various inputs to a function. Conditional expressions are necessary in languages without built-in pattern-directed invocation (e.g., Scheme). The following are additional examples of pattern-directed invocation:

```
- fun factorial(0) = 1
  = |   factorial(n) = n * factorial(n-1);
val factorial = fn : int -> int

- fun fibonacci(0) = 1
  = |   fibonacci(1) = 1
```

```
= |   fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);
val fibonacci = fn : int -> int
```

Argument Decomposition Within Argument List: **reverse**

Readers with an imperative programming background may be familiar with composing an argument to a function within a function call. For instance, in C:

```
1 int f (int a, int b) {
2     return (a+b);
3 }
4
5 int main() {
6     return f(2+3, 4);
7 }
```

Here, the expression `2+3` is the first argument to the function `f` that is called on line 6. Since C uses an *eager evaluation* parameter-passing strategy, the expression `2+3` is evaluated as 5 and then 5 is passed to `f`. However, in the body of `f`, there is no way to conveniently decompose 5 back to `2+3`.

Pattern-directed invocation allows ML to support the decomposition of an argument *from within the signature itself* by using a pattern in a parameter. For instance, consider these three versions of a `reverse` function:

```
1 $ cat reverse.sml
2 (* without pattern-directed invocation we need
3    an if-then-else and calls to hd and tl *)
4 fun reverse(lst) =
5     if null(lst) then nil
6     else reverse(tl(lst)) @ [hd(lst)];
7
8 (* with pattern-directed invocation and
9    calls to hd and tl *)
10 fun reverse(nil) = nil
11 |   reverse(lst) = reverse(tl(lst)) @ [hd(lst)];
12
13 (* with pattern-directed invocation,
14    calls to hd and tl are unnecessary *)
15 fun reverse(nil) = nil
16 |   reverse(x::xs) = reverse(xs) @ [x];
17 $
18 $ sml reverse.sml
19 Standard ML of New Jersey (64-bit) v110.98
20 [opening reverse.sml]
21 [autoloading]
22 [library $MLNJ-BASIS/basis.cm is stable]
23 [autoloading done]
24 val reverse = fn : 'a list -> 'a list
25 val reverse = fn : 'a list -> 'a list
26 val reverse = fn : 'a list -> 'a list
```

While the pattern-directed invocation in the second version (lines 10–11) obviates the need for the `if-then-else` expression (lines 5–6), the functions `hd` and `tl` (lines 6 and 11) are required to decompose `lst` into its head and tail. Calls to the functions `hd` and `tl` are obviated by using the pattern `x::xs` (line 16) in

the parameter to `reverse`. When the third version of `reverse` is called with a non-empty list, the second definition of it is executed (line 16), the head of the list passed as the argument is bound to `x`, and the tail of the list passed as the argument is bound to `xs`.

The `cases` form in the EOPL extension to Racket Scheme, which may be used to decompose the constituent parts of a variant record as described in Chapter 9 (Friedman, Wand, and Haynes 2001), is the Racket Scheme analog of the use of patterns in parameters to decompose arguments to a function. Pattern-directed invocation, including the use of patterns for decomposing arguments, and the pattern-action style of programming, is common in the programming language Prolog.

A Handle to Both Decomposed and Undecomposed Form of an Argument: `as`

Sometimes we desire access to both the decomposed argument and the undecomposed argument to a function without calling functions to decompose or recompose it. The use of `as` between a decomposed parameter and an undecomposed parameter maintains both throughout the definition of the function (line 3):

```

1 - fun konsMinHeadtoOther ([], _) = []
2 = | konsMinHeadtoOther (_, []) = []
3 = | konsMinHeadtoOther ((L1 as x::xs), (L2 as y::ys)) =
4   = if x < y then x::L2 else y::L1;
5 val konsMinHeadtoOther = fn : int list * int list -> int list
6 -
7 - konsMinHeadtoOther ([1,2,3,4], [5,6,7,8]);
8 val it = [1,5,6,7,8] : int list
9 -
10 - konsMinHeadtoOther ([9,2,3,4], [5,6,7,8]);
11 val it = [5,9,2,3,4] : int list

```

Anonymous Parameters

The underscore (`_`) pattern on lines 1 and 2 of the definition of the `konsMinHeadtoOther` function represents an anonymous parameter—a parameter whose name is unnecessary to the definition of the function. As an additional example, consider the following definition of a list `member` function:

```

- fun member(_, nil) = false
= | member(e, x::xs) = (x = e) orelse member(e, xs);
stdIn:2.27 Warning: calling polyEqual
val member = fn : 'a * 'a list -> bool

```

Type Variables

While some functions, such as `square` and `add`, require arguments of a particular type, others, such as `reverse` and `member`, accept arguments of any type or arguments whose types are partially restricted. For instance, the type of the

function `reverse` is `'a list -> 'a list`. Here, the `'a` means “any type.” Therefore, the function `reverse` accepts a list of any type `'a` and returns a list of the same type. The `'a` is called a *type variable*. In programming languages, the ability of a single function to accept arguments of different types is called *polymorphism* because *poly* means “many” and *morph* means “form.” Such a function is called *polymorphic*. A *polymorphic type* is a type expression containing type variables. The type of polymorphism discussed here is called *parametric polymorphism*, where a function or data type can be defined generically so that it can handle values identically without depending on their type. (The type variable `'a` means “any type that can be compared for equality.”)

Neither pattern-directed invocation nor operator/function overloading (sometimes called *ad hoc polymorphism*) is identical to (parametric) polymorphism. Overloading involves using the same operator/function name to refer to different definitions of a function, each of which is identifiable by the different number or types of arguments to which it is applied. Parametric polymorphism, in contrast, involves only one operator/function name referring to only one definition of the function that can accept arguments of multiple types. Thus, *ad hoc* polymorphism typically only supports a limited number of such distinct types, since a separate implementation must be provided for each type.

B.8.4 Local Binding and Nested Functions: `let` Expressions

A `let-in-end` expression in ML is used to introduce local binding for the purposes of avoiding recomputation of common subexpressions and creating nested functions for both protection and factoring out constant parameters so as to avoid passing (and copying) arguments that remain constant between successive recursive function calls.

Local Binding

Lines 8–12 of the following example demonstrate local binding in ML:

```

0  $ cat powerset.sml
1  fun insertineach(_, nil) = nil
2  |   insertineach(item, x::xs) =
3      (item::x)::insertineach(item, xs);
4
5  (* use of "let" prevents recomputation of powerset(xs) *)
6  fun powerset(nil) = [nil]
7  |   powerset(x::xs) =
8      let
9          val y = powerset(xs)
10         in
11             insertineach(x, y)@y
12         end;
13
14  $ sml powerset.sml
15  Standard ML of New Jersey (64-bit) v110.98
16  [opening powerset.sml]
17  val insertineach = fn : 'a * 'a list list -> 'a list list
18  val powerset = fn : 'a list -> 'a list list

```

These functions are the ML analogs of the following Scheme functions:

```
(define (insertineach item l)
  (cond
    ((null? l) '())
    (else (cons (cons item (car l))
                 (insertineach item (cdr l))))))
(define (powerset l)
  (cond
    ((null? l) '(()))
    (else
     (let ((y (powerset (cdr l))))
       (append (insertineach (car l) y) y))))))
```

Nested Functions

Since the function `insertineach` is intended to be only visible, accessible, and called by the `powerset` function, we can also use a `let ...in ...end` expression to nest it within the `powerset` function (lines 3–11 in the next example):

```
0 $ cat powerset.sml
1 fun powerset(nil) = [nil]
2 |   powerset(x::xs) =
3   let
4     fun insertineach(_, nil) = nil
5     |   insertineach(item, x::xs) =
6       (item::x)::insertineach(item, xs);
7
8     val y = powerset(xs)
9   in
10    insertineach(x, y)@y
11  end;
12 $
13 $ sml powerset.sml
14 Standard ML of New Jersey (64-bit) v110.98
15 [opening powerset.sml]
16 val powerset = fn : 'a list -> 'a list list
17
18 - powerset([1]);
19 val it = [[1],[[]]] : int list list
20
21 - powerset([1,2]);
22 val it = [[1,2],[1],[2],[[]]] : int list list
23
24 - powerset([1,2,3]);
25 val it = [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[[]]] : int list list
```

The following example uses a `let-in-end` expression to define a nested function that implements the *difference lists technique* to avoid appending in a definition of a `reverse` function:

```
$ cat reverse.sml
fun reverse(nil) = nil
|   reverse(l) =
  let
    fun reversel(nil, l) = l
```

```

        |   reversel(x::xs, ys) = reversel(xs, x::ys)
    in
        reversel(l, nil)
    end;
$
$ sml reverse.sml
Standard ML of New Jersey (64-bit) v110.98
[opening reverse.sml]
val reverse = fn : 'a list -> 'a list

```

Note that the polymorphic type of `reverse`, `[a] -> [a]`, indicates that `reverse` can reverse a list of any type.

B.8.5 Mutual Recursion

Unlike in Scheme, in ML a function must first be defined before it can be used in other functions:

```

- fun f(x) = g(x);
stdIn:1.12 Error: unbound variable or constructor: g

```

This makes the definition of *mutually recursive* functions (i.e., functions that call each other) problematic without direct language support. Mutually recursive functions in ML must be defined with the `and` reserved word between each definition. For instance, consider the functions `isodd` and `iseven`, which rely on each other to determine if an integer is odd or even, respectively:

```

- fun isodd(0) = false
  |   isodd(1) = true
  |   isodd(n) = iseven(n-1)
=
= and
=   iseven(0) = true
  |   iseven(n) = isodd(n-1);
val isodd = fn : int -> bool
val iseven = fn : int -> bool

- isodd(9);
val it = true : bool
- isodd(100);
val it = false : bool
- iseven(100);
val it = true : bool
- iseven(1000000000);
val it = true : bool

```

Note that more than two mutually recursive functions can be defined. Each but the last must be followed by an `and`, and the last is followed with a semicolon (`;`). ML performs tail-call optimization.

B.8.6 Putting It All Together: Mergesort

Consider the following definitions of a recursive `mergesort` function.

Unnested, Unhidden, Flat Version

```

$ cat mergesort.sml
fun split(nil) = (nil, nil)
|   split([x]) = (nil, [x])
|   split(x::y::excess) =
    let
      val (l, r) = split(excess)
    in
      (x::l, y::r)
    end;

fun merge(l, nil) = l
|   merge(nil, l) = l
|   merge(left as l::ls, right as r::rs) =
    if l < r then l::merge(ls, right)
    else r::merge(left, rs);

fun mergesort(nil) = nil
|   mergesort([x]) = [x]
|   mergesort(lat) =
    let
      (* split it *)
      val (left, right) = split(lat);

      (* mergesort each side *)
      val leftsorted = mergesort(left);
      val rightsorted = mergesort(right);
    in
      (* merge *)
      merge(leftsorted, rightsorted)
    end;

$
$ sml mergesort.sml
Standard ML of New Jersey (64-bit) v110.98
[opening mergesort.sml]
val split = fn : 'a list -> 'a list * 'a list
val merge = fn : int list * int list -> int list
val mergesort = fn : int list -> int list

```

Nested, Hidden Version

```

$ cat mergesort.sml
fun mergesort(nil) = nil
|   mergesort([x]) = [x]
|   mergesort(lat) =
    let
      fun split(nil) = (nil, nil)
      |   split([x]) = (nil, [x])
      |   split(x::y::excess) =
          let
            val (l, r) = split(excess)
          in
            (x::l, y::r)
          end;

      fun merge(l, nil) = l
      |   merge(nil, l) = l
    in
      merge(leftsorted, rightsorted)
    end;

```

```

|   merge(left as l::ls, right as r::rs) =
      if l < r then l::merge(ls, right)
      else r::merge(left, rs);

(* split it *)
val (left, right) = split(lat);

(* mergesort each side *)
val leftsorted = mergesort(left);
val rightsorted = mergesort(right);
in
  (* merge *)
  merge(leftsorted, rightsorted)
end;
$
$ sml mergesort.sml
Standard ML of New Jersey (64-bit) v110.98
[opening mergesort.sml]
val mergesort = fn : int list -> int list

```

Nested, Hidden Version Accepting a Comparison Operator as a Parameter

```

$ cat mergesort.sml
fun mergesort(_, nil) = nil
|   mergesort(_, [x]) = [x]
|   mergesort(compop, lat) =
    let
      fun split(nil) = (nil, nil)
      |   split([x]) = (nil, [x])
      |   split(x::y::excess) =
          let
            val (l, r) = split(excess)
          in
            (x::l, y::r)
          end;

      fun merge(_, l, nil) = l
      |   merge(_, nil, l) = l
      |   merge(compop, left as l::ls, right as r::rs) =
          if compop(l, r) then l::merge(compop, ls, right)
          else r::merge(compop, left, rs);

      (* split it *)
      val (left, right) = split(lat);

      (* mergesort each side *)
      val leftsorted = mergesort(compop, left);
      val rightsorted = mergesort(compop, right);
    in
      (* merge *)
      merge(compop, leftsorted, rightsorted)
    end;
$
$ sml mergesort.sml
Standard ML of New Jersey (64-bit) v110.98
[opening mergesort.sml]
val mergesort = fn : ('a * 'a -> bool) * 'a list -> 'a list

```

When passing an operator as an argument to a function, the operator passed must be a prefix operator. Since the operators `<` and `>` are infix operators, we cannot pass them to this version of `mergesort` without first converting each to a prefix operator. We can convert an infix operator to a prefix operator by wrapping it in a user-defined function (lines 1 and 4) or by using the built-in function `op`, which converts an infix operator to a prefix operator (lines 7, 10, and 13):

```

1 - mergesort((fn (x,y) => (x<y)), [9,8,7,6,5,4,3,2,1]);
2   val it = [1,2,3,4,5,6,7,8,9] : int list
3
4 - mergesort((fn (x,y) => (x>y)), [1,2,3,4,5,6,7,8,9]);
5   val it = [9,8,7,6,5,4,3,2,1] : int list
6
7 - (op <) (7, 2);
8   val it = false : bool
9
10 - mergesort((op <), [9,8,7,6,5,4,3,2,1]);
11   val it = [1,2,3,4,5,6,7,8,9] : int list
12
13 - mergesort((op >), [1,2,3,4,5,6,7,8,9]);
14   val it = [9,8,7,6,5,4,3,2,1] : int list

```

Since the closing lexeme for a comment in ML is `*`, we must add a whitespace character after the `*` when converting the infix multiplication operator to a prefix operator:

```

- (op *) (4,5);
stdIn:1.5 Error: unmatched close comment
stdIn:1.8-1.11 Error: syntax error: deleting LPAREN INT COMMA

- (op * ) (4,5);
val it = 20 : int

```

Final Version

The following code is the final version of `mergesort` using nested, protected functions and accepting a comparison operator as a parameter, which is factored out to avoid passing it between successive recursive calls:

```

$ cat mergesort.sml
fun mergesort(_, nil) = nil
|   mergesort(_, [x]) = [x]
|   mergesort(compop, lat) =
    let
        fun mergesort1(nil) = nil
        |   mergesort1([x]) = [x]
        |   mergesort1(lat1) =
            let
                fun split(nil) = (nil, nil)
                |   split([x]) = (nil, [x])
                |   split(x::y::excess) =
                    let
                        val (l, r) = split(excess)
                    in

```

```

        (x::l, y::r)
      end;

      fun merge(l, nil) = l
      |   merge(nil, l) = l
      |   merge(left as l::ls, right as r::rs) =
          if compop(l, r) then l::merge(ls, right)
          else r::merge(left, rs);

      (* split it *)
      val (left, right) = split(lat1);

      (* mergesort each side *)
      val leftsorted = mergesort1(left);
      val rightsorted = mergesort1(right);
    in
      (* merge *)
      merge(leftsorted, rightsorted)
    end;
  in
    mergesort1(lat)
  end;
$
$ sml mergesort.sml
Standard ML of New Jersey (64-bit) v110.98
[opening mergesort.sml]
val mergesort = fn : ('a * 'a -> bool) * 'a list -> 'a list

```

Notice also that we factored the argument `compop` out of the function `merge` in this version since it is visible from an outer scope.

B.9 Declaring Types

The reader may have noticed in the previous examples that ML infers the types of values (e.g., lists, tuples, and functions) that have not been explicitly declared by the programmer to be of a particular type with the `:` operator.

B.9.1 Inferred or Deduced

The following transcript demonstrates type inference.

```

- [1,2,3];
val it = [1,2,3] : int list
- (1, "Mary", 3.76)
val it = (1,"Mary",3.76) : int * string * real
- fun square(x) = x*x;
val square = fn : int -> int

```

B.9.2 Explicitly Declared

The following transcript demonstrates the use of explicitly declared types.

```

- [1,2,3] : int list;
val it = [1,2,3] : int list
- (1, "Mary", 3.76) : int * string * real;

```

```

val it = (1,"Mary",3.76) : int * string * real
- val square : int -> int = fn (x) => (x*x);
val square = fn : int -> int
- square(2);
val it = 4 : int
- square(2.0);
stdIn:7.1-7.12 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand:         real
  in expression:
    square 2.0
- val square : real -> real = fn (x) => (x*x);
val square = fn : real -> real
- square(2.0);
val it = 4.0 : real
- square(2);
stdIn:11.1-11.10 Error: operator and operand don't agree [literal]
  operator domain: real
  operand:         int
  in expression:
    square 2
- val rec reverse : int list -> int list = fn
-   (nil) => nil |
-   (x::xs) => reverse(xs) @ [x];
val reverse = fn : int list -> int list
- reverse([1,2,3]);
val it = [3,2,1] : int list
- reverse(["apple", "and", "orange"]);
stdIn:1.1-2.18 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int list
  operand:         string list
  in expression:
    reverse ("apple" :: "and" :: "orange" :: nil)

```

B.10 Structures

The ML module system consists of structures, signatures, and functors. A *structure* in ML is a collection of related data types and functions akin to a class from object-oriented programming. (Structures and functors in ML resemble classes and templates in C++, respectively.) Multiple predefined ML structures are available: TextIO, Char, String, List, Math. A function within a structure can be invoked with its fully qualified name (line 1) or, once the structure in which it resides has been opened (line 8), with its unqualified name (line 29):

```

1  - Int.toString(3);
2  [autoloading]
3  [library $SMLNJ-BASIS/basis.cm is stable]
4  [library $SMLNJ-BASIS/(basis.cm):basis-common.cm is stable]
5  [autoloading done]
6  val it = "3" : string
7  -
8  -- open Int;
9  opening Int
10  type int = ?int
11  val precision : Int.int option
12  val minInt : int option

```



```

13   val maxInt : int option
14   val toLarge : int -> IntInf.int
15   val fromLarge : IntInf.int -> int
16   val toInt : int -> Int.int
17   val fromInt : Int.int -> int
18   val ~ : int -> int
19   val + : int * int -> int
20   val - : int * int -> int
21   val * : int * int -> int
22   val div : int * int -> int
23   val mod : int * int -> int
24   val quot : int * int -> int
25   ...
26   ...
27   ...
28   -
29   - toString(4);
30   val it = "4" : string

```

To prevent a function from one structure overriding a different function with the same name from another structure in the single program, use fully qualified names [e.g., `Int.toString(3)`].

B.11 Exceptions

The following code is an example of an exception.

```

- exception NegativeInt;
- fun power(e,0) = if (e < 0) then raise NegativeInt else 0
= | power(e,1) = if (e > 0) then raise NegativeInt else 1
= | power(0,b) = 1
= | power(1,b) = b
= | power(e,b) = if (e > 0) then raise NegativeInt else b*power(e-1, b);
exception NegativeInt
val power = fn : int * int -> int
-
- power(3,~2);

uncaught exception NegativeInt
  raised at: stdIn:6.40-6.54

```

B.12 Input and Output

I/O is among the impure features of ML since I/O in ML involves side effect.

B.12.1 Input

The `option` data type has two values: `NONE` and `SOME`. Use `isSome(x)` to determine the value of a variable of type `option`. Use `valOf(x)` to extract the value of a variable of type `option`. A string option list is not the same as a string list.

Standard Input

The standard input stream generally does not need to be opened and closed.

```
- TextIO.inputLine(TextIO.stdIn);
get this line of text
val it = SOME "get this line of text\n" : string option
```

File Input

The following example demonstrates file input in ML.

```
$ cat input.txt
the quick brown fox ran slowly.
totally kewl
$
$ sml
Standard ML of New Jersey (64-bit) v110.98
-
- open TextIO;
< ... snipped ... >

- val ourinstream = openIn("input.txt");
val ourinstream = - : instream

- val line = inputLine(ourinstream);
val line = SOME "the quick brown fox ran slowly.\n" : string option

- isSome(line);
val it = true : bool

- val line = inputLine(ourinstream);
val line = SOME "totally kewl\n" : string option

- isSome(line);
val it = true : bool

- val line = inputLine(ourinstream);
val line = NONE : string option

- isSome(line);
val it = false : bool

- closeIn(ourinstream);
val it = () : unit
```

B.12.2 Parsing an Input File

The following program reads a file and returns a list of strings, where each string is a line from the file:

```
$ cat input.txt
This is certainly a
  a file containing
multiple lines of text.
  Each line is terminated with a
```

```

newline character. This file
    will    be    read

by an ML

    program.

$
$ cat input.sml
fun makeStringList(NONE) = nil
|   makeStringList(SOME str) = (String.tokens (Char.isSpace)) (str);

fun readInput(infile) =
    if TextIO.endOfStream(infile) then nil
    else TextIO.inputLine(infile)::readInput(infile);

val infile = TextIO.openIn("input.txt");

map makeStringList readInput(infile);

TextIO.closeIn(infile);
$
$ sml input.sml
Standard ML of New Jersey (64-bit) v110.98
[opening input.sml]
[autoloading]
[library $MLNJ-BASIS/basis.cm is stable]
[autoloading done]
val makeStringList = fn : string option -> string list
[autoloading]
[autoloading done]
val readInput = fn : TextIO.instream -> string option list
val infile = - : TextIO.instream
val it =
  [[{"This", "is", "certainly", "a"}, {"a", "file", "containing"},
   {"multiple", "lines", "of", "text."},
   {"Each", "line", "is", "terminated", "with", "a"},
   {"newline", "character.", "This", "file"}, {"will", "be", "read"}, [],
   {"by", "an", "ML"}, [], {"program."}] : string list list
val it = () : unit

```

B.12.3 Output

Standard Output

The `print` command prints strings to standard output:

```

- print "hello world";
hello worldval it = () : unit

- print "hello world\n";
hello world
val it = () : unit

```

Use the functions `Int.toString`, `Real.toString`, etc. to convert values of other data types into strings.

File Output

The following transcript demonstrates file output in ML.

```
- TextIO.output(TextIO.openOut("output.txt"), "hello world");
val it = () : unit

$ cat output.txt
hello world
```

Programming Exercises for Appendix B

Exercise B.1 Define a recursive ML function `remove` that accepts only a list and an integer `i` as arguments and returns another list that is the same as the input list, but with the `i`th element of the input list removed. If the length of the input list is less than `i`, return the same list. Assume that `i = 1` refers to the first element of the list.

Examples:

```
- remove;
val it = fn : int * 'a list -> 'a list
- remove(1, [9,10,11,12]);
val it = [10,11,12] : int list
- remove(2, [9,10,11,12]);
val it = [9,11,12] : int list
- remove(3, [9,10,11,12]);
val it = [9,10,12] : int list
- remove(4, [9,10,11,12]);
val it = [9,10,11] : int list
- remove(5, [9,10,11,12]);
val it = [9,10,11,12] : int list
```

Exercise B.2 Define a recursive ML function called `makeset` that accepts only a list of integers as input and returns the list with any repeating elements removed. The order in which the elements appear in the returned list does not matter, as long as there are no duplicate elements. Do not use any user-defined auxiliary functions, except `member`.

Examples:

```
- makeset;
val it = fn : 'a list -> 'a list
- makeset([1,3,4,1,3,9]);
val it = [4,1,3,9] : int list
- makeset([1,3,4,9]);
val it = [1,3,4,9] : int list
- makeset(["apple","orange","apple"]);
val it = ["orange","apple"] : string list
```

Exercise B.3 Define a recursive ML function `cycle` that accepts only a list and an integer `i` as arguments and cycles the list `i` times. Do not use any user-defined auxiliary functions.

Examples:

```
- cycle;
val it = fn : int * 'a list -> 'a list
- cycle(0, [1,4,5,2]);
val it = [1,4,5,2] : int list
- cycle(1, [1,4,5,2]);
val it = [4,5,2,1] : int list
- cycle(2, [1,4,5,2]);
val it = [5,2,1,4] : int list
- cycle(4, [1,4,5,2]);
val it = [1,4,5,2] : int list
- cycle(6, [1,4,5,2]);
val it = [5,2,1,4] : int list
- cycle(10, [1]);
val it = [1] : int list
- cycle(9, [1,4]);
val it = [4,1] : int list
```

Exercise B.4 Define an ML function `transpose` that accepts a list as its only argument and returns that list with adjacent elements transposed. Specifically, `transpose` accepts an input list of the form $[e_1, e_2, e_3, e_4, e_5, e_6 \dots, e_n]$ and returns a list of the form $[e_2, e_1, e_4, e_3, e_6, e_5, \dots, e_n, e_{n-1}]$ as output. If n is odd, e_n will continue to be the last element of the list. Do not use any user-defined auxiliary functions and do not use `@` (i.e., `append`).

Examples:

```
- transpose;
val it = fn : 'a list -> 'a list
- transpose ([1,2,3,4]);
val it = [2,1,4,3] : int list
- transpose ([1,2,3,4,5,6]);
val it = [2,1,4,3,6,5] : int list
- transpose ([1,2,3]);
val it = [2,1,3] : int list
```

Exercise B.5 Define a recursive ML function `oddevensum` that accepts only a list of integers as an argument and returns a pair consisting of the sum of the odd and even positions of the list. Do not use any user-defined auxiliary functions.

Examples:

```
- oddevensum;
val it = fn : int list -> int * int
- oddevensum([]);
val it = (0,0) : int * int
- oddevensum([6]);
val it = (6,0) : int * int
- oddevensum([6,3]);
val it = (6,3) : int * int
- oddevensum([6,3,8]);
val it = (14,3) : int * int
- oddevensum([1,2,3,4]);
val it = (4,6) : int * int
- oddevensum([1,2,3,4,5,6]);
```

```

val it = (9,12) : int * int
- oddevensum([1,2,3]);
val it = (4,2) : int * int

```

Exercise B.6 Define a recursive ML function `permutations` that accepts only a list representing a set as an argument and returns a list of all permutations of that list as a list of lists. Try to define only one auxiliary function and pass a λ -function to `map` within the body of that function and within the body of the `permutations` function to simplify their definitions. *Hint:* Use the ML function `List.concat`.

Examples:

```

- permutations;
val it = fn : 'a list -> 'a list list
- permutations([1]);
val it = [[1]] : int list list
- permutations([1,2]);
val it = [[1,2],[2,1]] : int list list
- permutations([1,2,3]);
val it = [[1,2,3],[1,3,2],[2,1,3],
          [2,3,1],[3,1,2],[3,2,1]] : int list list
- permutations([1,2,3,4]);
val it = [[1,2,3,4],[1,2,4,3],[1,3,2,4],[1,3,4,2],
          [1,4,2,3],[1,4,3,2],[2,1,3,4],[2,1,4,3],
          [2,3,1,4],[2,3,4,1],[2,4,1,3],[2,4,3,1],
          [3,1,2,4],[3,1,4,2],[3,2,1,4],[3,2,4,1],
          [3,4,1,2],[3,4,2,1],[4,1,2,3],[4,1,3,2],
          [4,2,1,3],[4,2,3,1],[4,3,1,2],[4,3,2,1]] : int list list
- permutations(["oranges", "and", "tangerines"]);
val it = [["oranges","and","tangerines"],
          ["oranges","tangerines","and"],
          ["and","oranges","tangerines"],
          ["and","tangerines","oranges"],
          ["tangerines","oranges","and"],
          ["tangerines","and","oranges"]] : string list list

```

Hint: This solution requires approximately 15 lines of code.

B.13 Thematic Takeaways

- While a goal of the functional style of programming is to bring programming closer to mathematics, ML and its syntax, as well as the responses of the ML interpreter (particularly for tuples and functions), make the connection between functional programming and mathematics salient.
- Native support for pattern-directed invocation is one of the most convenient features of user-defined functions in ML because it obviates the need for an `if-then-else` expression to differentiate between the various inputs to a function.
- Use of pattern-directed invocation (i.e., pattern matching) introduces declarative programming into ML.
- Pattern-directed invocation is not operator/function overloading.

- Operator/function overloading (sometimes called *ad hoc polymorphism*) is not *parametric polymorphism*.

B.14 Appendix Summary

ML is a statically typed and type-safe programming language that primarily supports functional programming, but has some imperative features. ML uses homogeneous lists with list operators `::` (i.e., cons) and `@` (i.e., append). The language supports *anonymous/λ functions* (i.e., unnamed or literal functions). A key language concept in ML is that all functions have types. Another key language concept in ML is *pattern-directed invocation*—a pattern-action rule-oriented style of programming, involving *pattern matching*, for defining and invoking functions. This appendix provides an introduction to ML so that readers can explore type concepts of programming languages through ML in Chapters 7–9. Table 9.7 compares the main concepts in Standard ML and Haskell.

B.15 Notes and Further Reading

There are two major dialects of ML: Standard ML (which is used in this text) and Caml (Categorical Abstract Machine Language). The primary implementation of Caml is OCaml (i.e., Object Caml), which extends Caml with object-oriented features. The language F#, which is part of the Microsoft .NET platform, is also a variant of ML and is largely compatible with OCaml. ML also influenced the development of Haskell. F#, like ML and Haskell, is statically typed and type safe and uses type inference. For more information on programming in ML, we refer readers to Ullman (1997). For a history of Standard ML, we refer readers to MacQueen, Harper, and Reppy (2020).

Appendix C

Introduction to Haskell

Haskell is one of the leading languages for teaching functional programming, enabling students to write simpler and cleaner code, and to learn how to structure and reason about programs.

— Graham Hutton, *Programming in Haskell* (2007)

HASKELL is a statically typed and type-safe programming language that primarily supports functional programming.

C.1 Appendix Objective

Establish an understanding of the syntax and semantics of Haskell, through examples, so that a reader with familiarity with imperative, and some functional, programming, after having read this appendix, can write intermediate programs in Haskell.

C.2 Introduction

Haskell is named after Haskell B. Curry, the pioneer of the *Y combinator* in λ -calculus—the mathematical theory of functions on which functional programming is based. Haskell is a useful general-purpose programming language in that it incorporates functional features from Lisp, rule-based programming (i.e., pattern matching) from Prolog, a terse syntax, and data abstraction from Smalltalk and C++. Haskell is a (nearly) pure functional language with some declarative features including pattern-directed invocation, guards, list comprehensions, and mathematical notation. It is an ideal vehicle through which to explore lazy evaluation, type safety, type inference, and currying. The objective here, however, is elementary programming in Haskell. We leave the use of the language to explore concepts to the main text.

This appendix is an example-oriented avenue to get started with Haskell programming and is intended to get a programmer who is already familiar with

the essential tenets of functional programming (Chapter 5) writing intermediate programs in Haskell; it is not intended as an exhaustive tutorial or comprehensive reference.

The primary objective of this appendix is to establish an understanding of Haskell programming in readers already familiar with the essential elements of functional programming in preparation for the study of typing and type inference (in Chapter 7), currying and higher-order functions (in Chapter 8), type systems (in Chapter 9), and lazy evaluation (in Chapter 12)—concepts that are both naturally and conveniently explored through Haskell. This appendix should be straightforward for anyone familiar with functional programming in another language, particularly Scheme. We sometimes compare Haskell expressions to their analogs in Scheme.

We use the *Glasgow Haskell Compiler* (GHC) implementation of Haskell developed at the University of Glasgow in this text. GHC is the state-of-the-art implementation of Haskell and compiles Haskell programs to native code on a variety of architectures as well as to C as an intermediate language. In this text, we use GHCi to interpret the Haskell expressions and programs we present. GHCi is the interactive environment of GHC—it provides a *read-eval-print* loop through which Haskell expressions can be interactively entered and evaluated, and through which entire programs can be interpreted. GHCi is started by entering `ghci` at the command prompt. Note that `Prelude>` is the prompt for input in the GHCi Haskell interpreter used in this text. A goal of the functional style of programming is to bring programming closer to mathematics. In this appendix, Haskell and especially its syntax as well as the responses of the Haskell interpreter make the connection between functional programming and mathematics salient.

C.3 Primitive Types

Haskell has the following primitive types: fixed precision integer (`Int`), arbitrary precision integer (`Integer`), single precision real (`Float`), boolean (`Bool`), and character (`Char`). The type of a string is `[Char]` (i.e., a list of characters); the type `String` is an alias for `[Char]`. The interpreter command `:type <expression>` (also `:t <expression>`) returns the type of `<expression>`:

```
1 Prelude> :type True
2 True :: Bool
3 Prelude> :type 'a'
4 'a' :: Char
5 Prelude> :type "hello world"
6 "hello world" :: String
7 Prelude> :type 3
8 3 :: Num a => a
9 Prelude> :type 3.3
10 3.3 :: Fractional a => a
```

Notice from lines 1–10 that Haskell uses *type inference*. The `::` double-colon symbol associates a value with a type and is read as “is of type.” For instance, the expression `a :: Char` indicates that `'a'` is of type `Char`. This explains the

responses of the interpreter on lines 2, 4, 6, 8, and 10 when an expression is entered prefaced with a `:type`. The responses from the interpreter for the expressions `3` (line 8) and `3.3` (line 10) require some explanation. In response to the expression `:type 3` (line 7), the interpreter prints `3 :: Num a => a` (line 8). Here, the `a` means “any type” and is called a *type variable*. Identifiers for type variables must begin with a lowercase letter (traditionally `a`, `b`, and so on are used). Before we can explain the meaning of the entire expression `3 :: Num a => a` (line 8), we must first discuss type classes.

C.4 Type Variables, Type Classes, and Qualified Types

To promote flexibility, Haskell has a hierarchy of type classes. A type class in Haskell is a set of types, unlike the concept of a class from object-oriented programming. Specifically, a *type class* in Haskell is a set of types, all of which define certain functions. The definition of a type class declares the names and types of the functions that all members of that class must define. Thus, a type class is like an interface from object-oriented programming, particularly an interface in Java. The concept of a class from object-oriented programming, which is the analog of a type (not a type class) in Haskell, can implement several interfaces, which means it must provide definitions for the functions specified (i.e., prototyped) in each interface. Haskell types are made instances of type classes in a similar way. When a Haskell type is declared to be an instance of a type class, that type promises to provide definitions of the functions in the definition of that class (i.e., signature). In summary, a class in object-oriented programming and a type in Haskell are analogs of each other; an interface in object-oriented programming and a type class in Haskell are analogs of each other as well (Table C.1).

The types `Int` and `Integer` are members of the `Integral` class, which is a subclass of the `Real` class. The types `Float` and `Double` are members of the `Floating` class, which is a subclass of the `Fractional` class. `Num` is the base class to which all numeric types belong. Other predefined Haskell type classes include `Eq`, `Show`, and `Ord`. A portion of the type class inheritance hierarchy in Haskell is shown in Figure C.1. The classes `Eq` and `Show` appear at the root of the hierarchy. The hierarchy involves multiple inheritance, which is akin to the ability of a Java class to implement more than one interface.

Returning to lines 7–8 in the previous transcript, the response `3 :: Num a => a` (line 8) indicates that if type `a` is in the class `Num`, then

Java	interface (Comparable)	class (Integer)
Haskell	type class (Ord)	type (Integer)

Table C.1 Conceptual Equivalence in Type Mnemonics Between Java and Haskell

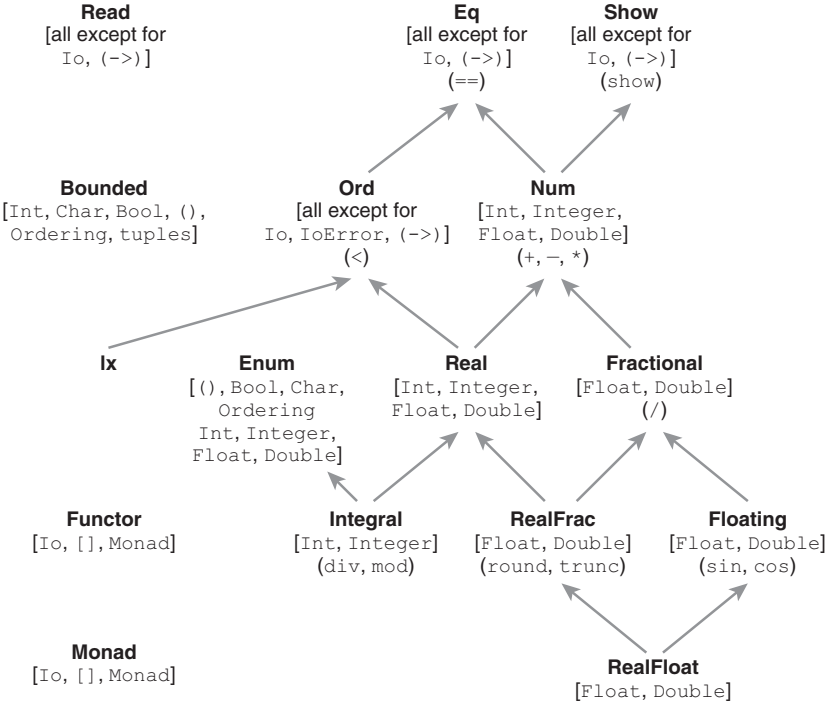


Figure C.1 A portion of the Haskell type class inheritance hierarchy. The types in brackets are the types that are members of the type class. The functions in parentheses are required by any instance (i.e., type) of the type class.

General:
$e :: C\ a \Rightarrow a$ means “If type a is in type class C , then e has type a .”
Example:
$3 :: \text{Num}\ a \Rightarrow a$ means “If type a is in type class Num , then 3 has type a .”

Table C.2 The General Form of a Qualified Type or Constrained Type and an Example

3 has the type a . In other words, 3 is of some type in the Num class. Such a type is called a *qualified type* or *constrained type* (Table C.2). The left-hand side of the $\Rightarrow$ symbol—which here is in the form $C\ a$ —is called the *class constraint* or *context*, where C is a type class and a is a type variable:

$$\begin{array}{ccccccc} & & \text{type class constraint} & & & & \\ & & \text{type class} & \text{type variable} & & & \\ \text{expression} & & & & & \text{type variable} & \\ \underbrace{e} & :: & \underbrace{C\ a}_{\text{context}} & \Rightarrow & \underbrace{a} & & \end{array}$$

We encounter qualified types again in our discussion of tuples and user-defined functions in Section C.8 and Section C.9, respectively.

C.5 Essential Operators and Expressions

Haskell was designed to have a terse syntax. For instance, in what follows notice that a `;` (semicolon) is almost never required in a Haskell program; the `cons` operator has been reduced from `cons` in Scheme to `::` in ML to `:` in Haskell; and the reserved words `define`, `lambda`, `|`, and `end` do not appear in function declarations and definitions. While programs written in a functional style are already generally more concise than their imperative analogs, “[a]lthough it is difficult to make an objective comparison, Haskell programs are often between two and ten times shorter than programs written in other current languages” (Hutton 2007, p. 4).

- **Character conversions.** The `ord` and `chr` functions in the `Data.Char` module are used for character conversions:

```

1  Prelude> Data.Char.ord('a')
2  97
3  Prelude> Data.Char.chr(97)
4  'a'
5  Prelude> :load Data.Char
6  Data.Char> ord('a')
7  97
8  Data.Char> chr(97)
9  'a'
10 Data.Char> chr(ord('a'))
11 'a'
```

A function within a *module* (i.e., a collection of related functions, types, and type classes) can be invoked with its fully qualified name (lines 1 and 3) or, once the module in which it resides has been loaded (line 5), with its unqualified name (lines 6, 8, and 10). From within a Haskell program file (or at the read-eval-print prompt of the interpreter), a module can be imported as follows:

```

1  Prelude> import Data.Char
2  Prelude Data.Char> ord('a')
3  97
4  Prelude Data.Char> chr(97)
5  'a'
```

A function within a module can also be individually imported:

```

1  Prelude> import Data.Char (ord)
2  Prelude Data.Char> ord('a')
3  97
4  Prelude Data.Char> chr(97)
5
6  <interactive>:3:1: error: Variable not in scope: chr :: t0 -> t
7  Prelude Data.Char>
```

Selected functions within a module can be collectively imported:

```

1  Prelude> import Data.Char (ord, chr)
2  Prelude Data.Char> ord('a')
3  97
```

```
4 Prelude Data.Char> chr (97)
5 'a'
```

- **String concatenation.** The ++ append operator is used for string concatenation:

```
Prelude> "hello" ++ " " ++ "world"
"hello world"
```

In Haskell, a string is a list of characters (i.e., [Char]).

- **Arithmetic.** The infix binary operators +, −, and * only accept two values whose types are members of the Num type class; the prefix unary minus operator negate only accepts a value whose type is a member of the Num type class; the infix binary division operator / only accepts two values whose types are members of the Fractional type class; the prefix binary division operator div only accepts two values whose types are members of the Integral type class; and the prefix binary modulus operator mod only accepts two values whose types are members of the Integral type class.

```
Prelude> 4.2 / 2.1
2.0
Prelude> div 4 2
2
Prelude> -1
-1
```

- **Comparison.** The infix binary operators == (equal to), <, >, <=, >=, and /= (not equal to) compare two integers, floating-point numbers, characters, or strings:

```
Prelude 4 == 2
False
Prelude> 4 > 2
True
Prelude> 4 /= 2
True
```

- **Boolean operators.** The infix operators || (or), && (and), and not are the or, and, and not boolean operators with their usual semantics. The operators || and && use *short-circuit evaluation* (or *lazy evaluation*, as discussed in Chapter 12):

```
Prelude> True || False
True
Prelude> False && False
False
Prelude> not False
True
```

- **Conditionals.** Use if-then-else expressions:

```
Prelude> if 1 /= 2 then "true branch" else "false branch"
"true branch"
```

There is no `if` expression without an `else` because all expressions must return a value.

- **Comments.**

- Single-line comments:

```
-- single-line comment until the end of the line
```

- Multi-line comments:

```
{- this is
   a
   multi-line
   comment -}
```

- Nested multi-line comments:

```
{- this is
   a
   {- nested
   multi-line -}
   comment -}
```

C.6 Running a Haskell Program

(Assuming a UNIX environment.)

- Enter `ghci` at the command prompt and enter expressions interactively to evaluate them:

```
$ ghci
Prelude> 2 + 3
5
Prelude> ^D
Leaving GHCi.
$
```

Using this method of execution, the programmer can create bindings and define new functions at the prompt of the interpreter:

```
Prelude> answer = 2 + 3
Prelude> answer
5
Prelude> f(x) = x + 1
Prelude> f(1)
2
```

Enter the EOF character (which is `<ctrl-d>` on UNIX systems and `<ctrl-z>` on Windows systems) or `:quit` (or `:q`) to quit the interpreter.

- Enter `ghci <filename>.hs` from the command prompt using file I/O, which causes the program in `<filename>.hs` to be evaluated:

```
$ cat first.hs

answer = 2 + 3

inc(x) = x + 1

$ ghci first.hs
*Main> answer
5
*Main> inc(1)
2
*Main>
```

After the program is evaluated, the read-eval-print loop of the interpreter is available to the programmer. Using this method of execution, the programmer cannot evaluate expressions within `<filename>.hs`, but can only create bindings and define new functions. However, once at the read-eval-print prompt, the programmer may evaluate expressions:

```
$ cat first.hs

2 + 3

$ ghci first.hs
GHCi, version 8.10.1: https://www.haskell.org/ghc/
:? for help
[1 of 1] Compiling Main                ( first.hs, interpreted )

first.hs:1:1: error:
  Parse error: module header, import declaration
               or top-level declaration expected.
   |
1  | 2 + 3
   | ^^^^
Failed, no modules loaded.
```

- Enter `ghci` at the command prompt and load a program by entering `:load "<filename>.hs"` into the read-eval-print prompt (or `:l "<filename>.hs"`), as shown in line 7:

```
0 $ cat first.hs
1
2 answer = 2 + 3
3
4 inc(x) = x + 1
5
6 $ ghci
7 Prelude> :load first.hs
8
9 *Main> answer
10 5
11 *Main>
```

If the program is modified, enter `:reload` (or `:r`) to reload it:

```
*Main> :reload # answer = 2+3 modified to answer = 2+4
*Main> answer
6
```

Again, using this method of execution, the programmer cannot evaluate expressions within `<filename>.hs`, but can only create bindings and define new functions.

- Redirect standard input into the interpreter from the keyboard to a file by entering `ghci < <filename>.hs` at the command prompt:¹

```
$ cat first.hs

2 + 3

$ ghci < first.hs
Prelude> 5
Prelude> Leaving GHCi.
$
```

Enter `:?` into the read-eval-print prompt to display all of the available interpreter commands.

C.7 Lists

The following are some important points about lists in Haskell.

- Lists in Haskell, unlike in Scheme, are *homogeneous*, meaning all elements of the list must be of the same type. For instance, the list `[1, 2, 3]` in Haskell is homogeneous, while the list `(1 "apple")` in Scheme is heterogeneous.
- In a type-safe language like Haskell, the values in a tuple (Section C.8) generally have different types, but the number of elements in the tuple must be fixed. Conversely, the values of a list must all have the same type, but the number of elements in the list is not fixed.
- The semantics of lexeme `[]` is the empty list.
- The cons operator, which accepts an element (the head) and a list (the tail), is `:` (e.g., `1:2:[3]`) and associates right-to-left.
- The expression `x:xs` represents a list of *at least one element*.
- The expression `xs` is pronounced *exes*.
- The expression `x:[ ]` represents a list of *exactly one element*, just as `[x]` does.
- The expression `x:y:xs` represents a list of *at least two elements*.
- The expression `x:y:[ ]` represents a list of *exactly two elements*.
- The functions `head` and `tail` are the Haskell analogs of the Scheme functions `car` and `cdr`, respectively.
- The element selection operator `(!!)` on a list uses zero-based indexing. For example, `[1, 2, 3, 4, 5]!!3` returns 4.
- The built-in function `len` returns the number of elements in its only list argument.
- The append operator `(++)` accepts two lists and appends them to each other. For example, `[1, 2]++[3, 4, 5]`; returns `[1, 2, 3, 4, 5]`). The append operator in Haskell is also inefficient, just as it is in Scheme.

1. The interpreter automatically exits once EOF is reached and evaluation is complete.

- The built-in function `elem` is a list member and returns `True` if its first argument is a member of its second list argument and `False` otherwise.

Examples:

```

Prelude> :type [1,2,3]
[1,2,3] :: Num a => [a]
Prelude> :type [1.1,2.2,3.3,4.4]
[1.1,2.2,3.3,4.4] :: Fractional a => [a]
Prelude> :type []
[] :: [a]
Prelude> 1:2:[3]
[1,2,3]
Prelude> 1:[]
[1]
Prelude> 1:2:[]
[1,2]
Prelude> :type head
head :: [a] -> a
Prelude> head (1:2:[3])
1
Prelude> tail (1:2:[3])
[2,3]
Prelude> head ([1,2,3])
1
Prelude> tail ([1,2,3])
[2,3]
Prelude> head "hello world"
h
Prelude> :load Data.Char
Data.Char> :type isDigit
isDigit :: Char -> Bool
Data.Char> isDigit (head "hello world")
False
Data.Char> [1,2,3] !! 2
3
Data.Char> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]

```

As can be seen, in Haskell a `String` is a list of `Chars`.

C.8 Tuples

A *tuple* is a sequence of elements of potentially mixed types. Formally, a tuple is an element e of a Cartesian product of a given number of sets: $e \in (S_1 \times S_2 \times \cdots \times S_n)$. A two-element tuple is called a *pair* [e.g., $e \in (A \times B)$]. A three-element tuple is called a *triple* [e.g., $e \in (A \times B \times C)$]. A tuple typically contains unordered, *heterogeneous* elements akin to a `struct` in C with the exception that a tuple is indexed by numbers (like a list) rather than by field names (like a `struct`). While tuples can be heterogeneous, in a list of tuples, each tuple in the list must be of the same type. Elements of a pair (i.e., a 2-tuple) are accessible with the functions `fst` and `snd`:

```

1 Prelude> :type (1, "Mary")
2 (1, "Mary") :: Num a => (a, [Char])

```

```

3 Prelude> fst (1, "Mary")
4 1
5 Prelude> snd (1, "Mary")
6 "Mary"
7 Prelude> :type (1, "Mary", 3.76)
8 (1, "Mary", 3.76) :: (Fractional c, Num a) => (a, [Char], c)

```

The response from the interpreter when `:type (1, "Mary", 3.76)` is entered (line 7) is `(1, "Mary", 3.76) :: (Fractional c, Num a) => (a, [Char], c)` (line 8). The expression `(Fractional c, Num a) => (a, [Char], c)` is a *qualified type*. Recall that the `a` means “any type” and is called a *type variable*; the same holds for type `c` in this example. The expression `(1, "Mary", 3.76) :: (Fractional c, Num a) => (a, [Char], c)` (line 8) indicates that if type `c` is in the class `Fractional` and type `a` is in the class `Num`, then the tuple `(1, "Mary", 3.76)` has type `(a, [Char], c)`. In other words, the tuple `(1, "Mary", 3.76)` consists of an instance of type `a`, a list of Characters, and an instance of type `c`.

The right-hand side of the response from the interpreter when a tuple is entered [e.g., `(a, [Char], c)`] demonstrates that a tuple is an element of a Cartesian product of a given number of sets. Here, the comma `(,)` is the analog of the Cartesian-product operator $\times$, and the data types `a`, `[Char]`, and `c` are the sets involved in the Cartesian product. In other words, `(a, [Char], c)` is a type defined by the Cartesian product of the set of all instances of type `a`, where `a` is a member of the `Num` class; the set of all lists of type `Char`; and the set of all instances of type `c`, where `c` is a member of the `Fractional` class. An element of the Cartesian product of the set of all instances of type `a`, where `a` is a member of the `Num` class; the set of all lists of type `Char`; and the set of all instances of type `c`, where `c` is a member of the `Fractional` class, has the type `(a, [Char], c)`:

$$(1, \text{"Mary"}, 3.76) \in (\text{Num} \times [\text{Char}] \times \text{Fractional})$$

The argument list of a function in Haskell, described in Section C.9, is a tuple. Therefore, Haskell uses tuples to specify the domain of a function.

C.9 User-Defined Functions

A key language concept in Haskell is that all functions have types. Function, parameter, and value names must begin with a lowercase letter.

C.9.1 Simple User-Defined Functions

The following are some simple user-defined functions:

```

1 Prelude> square(x) = x*x
2 Prelude>
3 Prelude> :type square
4 square :: Num a => a -> a
5 Prelude>
6 Prelude> add(x,y) = x+y

```

```

7  Prelude>
8  Prelude> :type add
9  add :: Num a => (a, a) -> a

```

Here, when `:type square` is entered (line 3), the response of the interpreter is `square :: Num a => a -> a` (line 4), which is a *qualified type*. Recall that the `a` means “any type” and is called a *type variable*. To promote flexibility, especially in function definitions, Haskell has *type classes*, which are collections of types. Also, recall that the types `Int` and `Integer` belong to the `Num` type class. The expression `square :: Num a => a -> a` indicates that if type `a` is in the class `Num`, then the function `square` has type `a -> a`. In other words, `square` is a function that maps a value of type `a` to a value of the same type `a`. If the argument to `square` is of type `Int`, then `square` is a function that maps an `Int` to an `Int`. Similarly, when `:type add` is entered (line 8), the response of the interpreter is `add :: Num a => (a, a) -> a` (line 9); this indicates that if type `a` is in the class `Num`, then the type of the function `add` is `(a, a) -> a`. In other words, `add` is a function that maps a pair `(a, a)` of values, both of the same type `a`, to a value of the same type `a`. Notice that the interpreter prints the domain of a function that accepts more than one parameter as a tuple (using the notation described in Section C.8). These functions are the Haskell analogs of the following Scheme functions:

```

(define (square x) (* x x))
(define (add x y) (+ x y))

```

Notice that the Haskell syntax involves fewer lexemes than Scheme (e.g., `define` is not included). Without excessive parentheses, Haskell is also more readable than Scheme.

C.9.2 Lambda Functions

Lambda functions (i.e., anonymous or literal functions) are introduced with `\` (which is visually similar to λ). They are often used, as in other languages, in concert with higher-order functions including `map`, which is built into Haskell as in Scheme:

```

Prelude> (\n -> n+1) (5)
6
Prelude> map (\n -> n+1) [1,2,3]
[2,3,4]

```

These expressions are the Haskell analogs of the following Scheme expressions:

```

> ((lambda (n) (+ n 1)) 5)
6
> (map (lambda (n) (+ n 1)) '(1 2 3))
(2 3 4)

```

Moreover, the functions

```
Prelude> add = (\(x,y) -> x+y)
Prelude>
Prelude> :type add
add :: Num a => (a, a) -> a
Prelude>
Prelude> square = (\x -> x*x)
Prelude>
Prelude> :type square
square :: Num a => a -> a
```

are the Haskell analogs of the following Scheme functions:

```
(define add (lambda (x y) (+ x y)))
(define square (lambda (x) (* x x)))
```

Anonymous functions are often used as arguments to higher-order functions.

C.9.3 Pattern-Directed Invocation

A key feature of Haskell is its support for the definition and invocation of functions using a pattern-matching mechanism called *pattern-directed invocation*. In pattern-directed invocation, the programmer writes multiple definitions of the same function. When that function is called, the determination of the particular definition of the function to be executed is made based on pattern matching the arguments passed to the function with the patterns used as parameters in the signature of the function. For instance, consider the following definitions of a greatest common divisor function:

```
1 -- (gcd is in Prelude.hs)
2 -- first version without pattern-directed invocation
3 gcd1(u,v) = if v == 0 then u else gcd1(v, (mod u v))
4
5 -- second version with pattern-directed invocation
6 gcd1(u,0) = u
7 gcd1(u,v) = gcd1(v, (mod u v))
```

The first version (defined on line 3) does not use pattern-directed invocation; that is, there is only one definition of the function. The second version (defined on lines 6–7) uses pattern-directed invocation. If the literal 0 is passed as the second argument to the function `gcd1`, then the first definition of `gcd1` is used (line 6); otherwise the second definition is used (line 7).

Pattern-directed invocation is not identical to operator/function overloading. Overloading involves determining which definition of a function to invoke based on the number and types of arguments it is passed at run-time. With pattern-directed invocation, no matter how many definitions of the function exist, all have the same type signature (i.e., number and type of parameters). Overloading implies that the number and types of arguments are used to select the applicable function definition from a collection of function definitions with the same name.

Native support for pattern-directed invocation is one of the most convenient features of user-defined functions in Haskell because it obviates the need for an `if-then-else` expression to differentiate between the various inputs to a function. Conditional expressions are necessary in languages without built-in pattern-directed invocation (e.g., Scheme). The following are additional examples of pattern-directed invocation:

```
factorial(0) = 1
factorial(n) = n * factorial(n-1)

fact(n) = product [1..n]

fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

Argument Decomposition Within Argument List: **reverse**

Readers with an imperative programming background may be familiar with composing an argument to a function within a function call. For instance, in C:

```
int f (int a, int b) {
    return (a+b);
}

int main() {
    return f(2+3, 4);
}
```

Here, the expression `2+3` is the first argument to the function `f`. Since C uses an *eager evaluation* parameter-passing strategy, the expression `2+3` is evaluated as 5 and then 5 is passed to `f`. However, in the body of `f`, there is no way to conveniently decompose 5 back to `2+3`.

Pattern-directed invocation allows Haskell to support the decomposition of an argument *from within the signature itself* by using a pattern in a parameter. For instance, consider these three versions of a `reverse` function:

```
1 Prelude> :{
2 Prelude | -- without pattern-directed invocation need
3 Prelude | -- an if-then-else and calls to head and tail
4 Prelude | -- reverse is built-in Prelude.hs
5 Prelude | reverse1(lst) =
6 Prelude |     if lst == [] then []
7 Prelude |     else reverse1(tail(lst)) ++ [head(lst)]
8 Prelude |
9 Prelude | -- with pattern-directed invocation;
10 Prelude | -- still need calls to head and tail
11 Prelude | reverse2([]) = []
12 Prelude | reverse2(lst) = reverse2(tail(lst)) ++ [head(lst)]
13 Prelude |
14 Prelude | -- with pattern-directed invocation;
15 Prelude | -- calls to head and tail unnecessary
16 Prelude | reverse3([]) = []
17 Prelude | reverse3(x:xs) = reverse3(xs) ++ [x]
```

```

18 Prelude | :}
19 Prelude >
20 Prelude > :type reverse1
21 reverse1 :: Eq a => [a] -> [a]
22 Prelude >
23 Prelude > :type reverse2
24 reverse2 :: [a] -> [a]
25 Prelude >
26 Prelude > :type reverse3
27 reverse3 :: [a] -> [a]

```

Functions can be defined at the Haskell prompt as shown here. If a function or set of functions requires multiple lines, use `: \{` and `: \}` lexemes (as shown on lines 1 and 18, respectively) to identify to the interpreter the beginning and ending of a block of code consisting of multiple lines.

While the pattern-directed invocation in `reverse2` (lines 11–12) obviates the need for the `if-then-else` expression (lines 6–7) in `reverse1`, the functions `head` and `tail` are required to decompose `lst` into its head and tail. Calls to the functions `head` and `tail` (lines 7 and 12) are obviated by using the pattern `x : xs` in the parameter to `reverse3` (line 17). When `reverse3` is called with a non-empty list, the second definition of it is executed (line 17), the head of the list passed as the argument is bound to `x`, and the tail of the list passed as the argument is bound to `xs`.

The `cases` form in the EOPL extension to Racket Scheme, which may be used to decompose the constituent parts of a variant record as described in Chapter 9 (Friedman, Wand, and Haynes 2001), is the Racket Scheme analog of the use of patterns in parameters to decompose arguments to a function. Pattern-directed invocation, including the use of patterns for decomposing parameters, and the pattern-action style of programming, is common in the programming language Prolog.

A Handle to Both Decomposed and Undecomposed Form of an Argument: @

Sometimes we desire access to both the decomposed argument and the undecomposed argument to a function without calling functions to decompose or recompose it. The use of `@` between a decomposed parameter and an undecomposed parameter maintains both throughout the definition of the function (line 4):

```

1 Prelude > :{
2 Prelude | konsMinHeadtoOther ([], _) = []
3 Prelude | konsMinHeadtoOther (_, []) = []
4 Prelude | konsMinHeadtoOther (l1@(x:xs), l2@(y:ys)) =
5 Prelude |     if x < y then x:l2 else y:l1
6 Prelude | :}
7 Prelude >
8 Prelude > :type konsMinHeadtoOther
9 konsMinHeadtoOther :: Ord a => ([a], [a]) -> [a]
10 Prelude >
11 Prelude > konsMinHeadtoOther ([1,2,3,4], [5,6,7,8])
12 [1,5,6,7,8]
13 Prelude >

```

```

14 Prelude> konsMinHeadtoOther ([9,2,3,4], [5,6,7,8])
15 [5,9,2,3,4]

```

Anonymous Parameters

The underscore (`_`) pattern on lines 2 and 3 of the definition of the `konsMinHeadtoOther` function represents an anonymous parameter—a parameter whose name is unnecessary to the definition of the function. As an additional example, consider the following definition of a list `member` function:

```

Prelude> :{
Prelude| -- elem is the Haskell member function in Prelude.hs
Prelude| member(_, []) = False
Prelude| member(e, x:xs) = (x == e) || member(e,xs)
Prelude| :}
Prelude>
Prelude> :type member
member :: Eq a => (a, [a]) -> Bool

```

Using anonymous parameters (lines 1–3), we can also define functions to access the elements of a tuple:

```

1 Prelude> get1st (e,_,_) = e
2 Prelude> get2nd (_,e,_) = e
3 Prelude> get3rd (_,_,e) = e
4 Prelude> get1st (1, "Mary", 3.76)
5 1
6 Prelude> get2nd (1, "Mary", 3.76)
7 "Mary"
8 Prelude> get3rd (1, "Mary", 3.76)
9 3.76

```

Polymorphism

While some functions, including `square` and `add`, require arguments of a particular type, others, including `reverse3` and `member`, accept arguments of any type or arguments whose types are partially restricted. For instance, the type of the function `reverse3` is `[a] -> [a]`. Here, the `a` means “any type.” Therefore, the function `reverse` accepts a list of a particular type `a` and returns a list of the same type. The `a` is called a *type variable*. In programming languages, the ability of a single function to accept arguments of different types is called *polymorphism* because *poly* means “many” and *morph* means “form.” Such a function is called *polymorphic*. A *polymorphic type* is a type expression containing type variables. The type of polymorphism discussed here is called *parametric polymorphism*, where a function or data type can be defined generically so that it can handle values identically without depending on their type.

Neither pattern-directed invocation nor operator/function overloading (sometimes called *ad hoc polymorphism*) is identical to (parametric) polymorphism. Overloading involves using the same operator/function name to refer to different definitions of a function, each of which is identifiable by the different number or

types of arguments to which it is applied. Parametric polymorphism, in contrast, involves only one operator/function name referring to only one definition of the function that can accept arguments of multiple types. Thus, ad hoc polymorphism typically only supports a limited number of such distinct types, since a separate implementation must be provided for each type.

C.9.4 Local Binding and Nested Functions: **let** Expressions

A `let-in` expression in Haskell is used to introduce local binding for the purposes of avoiding recomputation of common subexpressions and creating nested functions for both protection and factoring out so as to avoid passing (and copying) arguments that remain constant between recursive function calls.

Local Binding

Lines 8–11 of the following example demonstrate local binding in Haskell:

```

1  Prelude> :{
2  Prelude| insertineach(_, []) = []
3  Prelude| insertineach(item, x:xs) = (item:x):insertineach(item,xs)
4  Prelude|
5  Prelude| -- use of "let" prevents recomputation of powerset xs
6  Prelude| powerset([]) = [[]]
7  Prelude| powerset(x:xs) =
8  Prelude|     let
9  Prelude|         temp = powerset(xs)
10 Prelude|     in
11 Prelude|         (insertineach(x, temp)) ++ temp
12 Prelude| :}
13 Prelude>
14 Prelude> :type insertineach
15 insertineach :: (a, [[a]]) -> [[a]]
16 Prelude>
17 Prelude> :type powerset
18 powerset :: [a] -> [[a]]

```

The `powerset` function can also be defined using `where`:

```

powerset([]) = [[]]
powerset(x:xs) = (insertineach(x, temp)) ++ temp
                where temp = powerset(xs)

```

These functions are the Haskell analogs of the following Scheme functions:

```

(define (insertineach item l)
  (cond
    ((null? l) '())
    (else (cons (cons item (car l))
                 (insertineach item (cdr l))))))
(define (powerset l)
  (cond
    ((null? l) '())
    (else
     (let ((y (powerset (cdr l))))
       (append (insertineach (car l) y) y)))))

```


Nested Functions

Since the function `insertineach` is intended to be only visible, accessible, and called by the `powerset` function, we can also use a `let-in` to nest it within the `powerset` function (lines 4–10 in the next example):

```

1  Prelude> :{
2  Prelude| powerset([]) = [[]]
3  Prelude| powerset(x:xs) =
4  Prelude|     let
5  Prelude|         insertineach(_, []) = []
6  Prelude|         insertineach(item, x:xs) = (item:x):insertineach(item,xs)
7  Prelude|
8  Prelude|         temp = powerset(xs)
9  Prelude|     in
10 Prelude|         (insertineach(x, temp)) ++ temp
11 Prelude|
12 Prelude| {-
13 Prelude| -- powerset can be similarly defined with where
14 Prelude| powerset([]) = [[]]
15 Prelude| powerset(x:xs) = (insertineach(x, temp)) ++ temp
16 Prelude|         where
17 Prelude|             insertineach(_, []) = []
18 Prelude|             insertineach(item, x:xs) =
19 Prelude|                 (item:x):insertineach(item,xs)
20 Prelude|
21 Prelude|             temp = powerset(xs)
22 Prelude| -}
23 Prelude| :}
24 Prelude>
25 Prelude> :type powerset
26 powerset :: [a] -> [[a]]
27 Prelude>
28 Prelude> powerset([])
29 [[]]
30 Prelude>
31 Prelude> powerset([1])
32 [[1],[1]]
33 Prelude> powerset([1,2])
34 [[1,2],[1],[2],[1]]
35 Prelude>
36 Prelude> powerset([1,2,3])
37 [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[1]]

```

The following example uses a `let-in` expression to define a nested function that implements the *difference lists technique* to avoid appending in a definition of a `reverse` function:

```

Prelude> :{
Prelude| reverse5l([], m) = m
Prelude| reverse5l(x:xs, ys) = reverse5l(xs, x:ys)
Prelude|
Prelude| reverse5(lst) = reverse5l(lst, [])
Prelude| :}
Prelude>
Prelude> :type reverse5l
reverse5l :: ([a], [a]) -> [a]
Prelude>
Prelude> :type reverse5
reverse5 :: [a] -> [a]

```

We can nest `reverse51` within `reverse5` to hide and protect it:

```
Prelude> :{
Prelude| reverse5([]) = []
Prelude| reverse5(1) =
Prelude|     let
Prelude|         reverse51([], 1) = 1
Prelude|         reverse51(x:xs, ys) = reverse51(xs, x:ys)
Prelude|     in
Prelude|         reverse51(1, [])
Prelude| :}
Prelude>
Prelude> :type reverse5
reverse5 :: [a] -> [a]
```

Note that the polymorphic type of `reverse`, `[a] -> [a]`, indicates that `reverse` can reverse a list of any type.

C.9.5 Mutual Recursion

In Haskell, as in Scheme but unlike in ML, a function may call a function that is defined below it:

```
Prelude> :{
Prelude| f(x,y) = square(x+y)
Prelude| square(x) = x*x
Prelude| :}
Prelude>
Prelude> f(3,4)
49
```

This makes the definition of *mutually recursive* functions straightforward. For instance, consider the functions `isodd` and `iseven`, which rely on each other to determine if an integer is odd or even, respectively:

```
Prelude> :{
Prelude| isodd(1) = True
Prelude| isodd(0) = False
Prelude| isodd(n) = iseven(n-1)
Prelude|
Prelude| iseven(0) = True
Prelude| iseven(n) = isodd(n-1)
Prelude| :}
Prelude>
Prelude> :type isodd
isodd :: (Eq a, Num a) => a -> Bool
Prelude>
Prelude> :type iseven
iseven :: (Eq a, Num a) => a -> Bool
Prelude>
Prelude> isodd(9)
True
Prelude>
Prelude> isodd(100)
False
Prelude> iseven(100)
```

```

True
Prelude> iseven(1000000000)
True

```

Note that more than two mutually recursive functions can be defined.

C.9.6 Putting It All Together: Mergesort

Consider the following definitions of a recursive `mergesort` function.

Unnested, Unhidden, Flat Version

```

Prelude> :(
Prelude| split([]) = ([], [])
Prelude| split([x]) = ([], [x])
Prelude| split(x:y:excess) =
Prelude|   let
Prelude|     (left, right) = split(excess)
Prelude|   in
Prelude|     (x:left, y:right)
Prelude|
Prelude| merge(l, []) = l
Prelude| merge([], l) = l
Prelude| merge(l:ls, r:rs) =
Prelude|   if l < r then l:merge(ls, r:rs)
Prelude|   else r:merge(l:ls, rs)
Prelude|
Prelude| mergesort([]) = []
Prelude| mergesort([x]) = [x]
Prelude| mergesort(lat) =
Prelude|   let
Prelude|     -- split it
Prelude|     (left, right) = split(lat)
Prelude|
Prelude|     -- mergesort each side
Prelude|     leftsorted = mergesort(left)
Prelude|     rightsorted = mergesort(right)
Prelude|   in
Prelude|     -- merge
Prelude|     merge(leftsorted, rightsorted)
Prelude|
Prelude| {-
Prelude| -- alternatively
Prelude| mergesort([]) = []
Prelude| mergesort([x]) = [x]
Prelude| mergesort(lat) =
Prelude|   -- merge
Prelude|   merge(leftsorted, rightsorted)
Prelude|   where
Prelude|     -- split it
Prelude|     (left, right) = split(lat)
Prelude|
Prelude|     -- mergesort each side
Prelude|     leftsorted = mergesort(left)
Prelude|     rightsorted = mergesort(right)
Prelude| -}
Prelude| :)
Prelude>
Prelude> :type split

```

```

split :: [a] -> ([a], [a])
Prelude>
Prelude> :type merge
merge :: Ord a => ([a], [a]) -> [a]
Prelude>
Prelude> :type mergesort
mergesort :: Ord a => [a] -> [a]

```

Nested, Hidden Version

```

Prelude> :{
Prelude| mergesort([]) = []
Prelude| mergesort([x]) = [x]
Prelude| mergesort(lat) =
Prelude|   let
Prelude|     split([]) = ([], [])
Prelude|     split([x]) = ([], [x])
Prelude|     split(x:y:excess) =
Prelude|       let
Prelude|         (left, right) = split(excess)
Prelude|       in
Prelude|         (x:left, y:right)
Prelude|
Prelude|     merge(l, []) = l
Prelude|     merge([], l) = l
Prelude|     merge(l:ls, r:rs) =
Prelude|       if l < r then l:merge(ls, r:rs)
Prelude|       else r:merge(l:ls, rs)
Prelude|
Prelude|     -- split it
Prelude|     (left, right) = split(lat)
Prelude|
Prelude|     -- mergesort each side
Prelude|     leftsorted = mergesort(left)
Prelude|     rightsorted = mergesort(right)
Prelude|   in
Prelude|     -- merge
Prelude|     merge(leftsorted, rightsorted)
Prelude| :}
Prelude> :type mergesort
mergesort :: Ord a => [a] -> [a]

```

Nested, Hidden Version Accepting a Comparison Operator as a Parameter

```

1  Prelude> :{
2  Prelude| mergesort(_, []) = []
3  Prelude| mergesort(_, [x]) = [x]
4  Prelude| mergesort(compop, lat) =
5  Prelude|
6  Prelude|   let
7  Prelude|     split([]) = ([], [])
8  Prelude|     split([x]) = ([], [x])
9  Prelude|     split(x:y:excess) =
10  Prelude|       let
11  Prelude|         (left, right) = split(excess)
12  Prelude|       in

```

```

13 Prelude|             (x:left, y:right)
14 Prelude|
15 Prelude|         merge(_, l, []) = l
16 Prelude|         merge(_, [], l) = l
17 Prelude|         merge(compop, l:ls, r:rs) =
18 Prelude|             if compop(l, r) then l:merge(compop, ls, r:rs)
19 Prelude|             else r:merge(compop, l:ls, rs)
20 Prelude|
21 Prelude|         -- split it
22 Prelude|         (left, right) = split (lat)
23 Prelude|
24 Prelude|         -- mergesort each side
25 Prelude|         leftsorted = mergesort(compop, left)
26 Prelude|         rightsorted = mergesort(compop, right)
27 Prelude|     in
28 Prelude|         -- merge
29 Prelude|         merge(compop, leftsorted, rightsorted)
30 Prelude| :}
31 Prelude>
32 Prelude> :type mergesort
33 mergesort :: ((a, a) -> Bool, [a]) -> [a]
34 Prelude>
35 Prelude> mergesort((\ (x,y) -> (x<y)), [9,8,7,6,5,4,3,2,1])
36 [1,2,3,4,5,6,7,8,9]
37 Prelude>
38 Prelude> mergesort((\ (x,y) -> (x>y)), [1,2,3,4,5,6,7,8,9])
39 [9,8,7,6,5,4,3,2,1]

```

We pass a user-defined function as the comparison argument on lines 35 and 38 because the passed function must be invoked using prefix notation (line 18). Since the operators `<` and `>` are infix operators, we cannot pass them to this version of `mergesort` without first converting each to prefix form. We can convert an infix operator to prefix form by wrapping it in a user-defined function (lines 35 and 38) or parentheses:

```

1 Prelude> :type (<)
2 (<) :: Ord a => a -> a -> Bool
3 Prelude> :type (>)
4 (>) :: Ord a => a -> a -> Bool
5 Prelude>
6 Prelude> :type (\ (x,y) -> (x<y))
7 (\ (x,y) -> (x<y)) :: Ord a => (a, a) -> Bool
8 Prelude> :type (\ (x,y) -> (x>y))
9 (\ (x,y) -> (x>y)) :: Ord a => (a, a) -> Bool

```

However, the type of these operators, once converted to prefix form, is `a -> a -> Bool` (lines 2 and 4) which does not match the expected type `(a, a) -> Bool` of the first parameter to `mergesort` (line 33). Wrapping an operator in parentheses not only converts it to prefix form, but also curries the operator. *Currying* refers to converting an n -ary function into one that accepts only one argument and returns a function that also accepts only one argument, which returns a function that accepts only one argument, and so on. (See Section 8.3 for the details of currying.) Thus, for this version of `mergesort` to accept `(<)` or `(>)` as a first argument, we must replace the subexpression `compop(l, r)` in line 18 of the definition of `mergesort` with `(compop l r)`.

This changes the type of mergesort from $((a, a) \rightarrow \text{Bool}, [a]) \rightarrow [a]$ to $(a \rightarrow a \rightarrow \text{Bool}, [a]) \rightarrow [a]$:

```

1 Prelude> :type mergesort
2 mergesort :: (a -> a -> Bool, [a]) -> [a]
3 Prelude>
4 Prelude> mergesort(<), [9,8,7,6,5,4,3,2,1])
5 [1,2,3,4,5,6,7,8,9]
6 Prelude>
7 Prelude> mergesort(>), [9,8,7,6,5,4,3,2,1])
8 [9,8,7,6,5,4,3,2,1]

```

Of course, unlike the previous version, this new definition of mergesort cannot accept an uncurried function as its first argument.

Final Version

The following code is the final version of mergesort using nested, protected functions and accepting a comparison operator as a parameter, which is factored out to avoid passing it between successive recursive calls:

```

Prelude> :{
Prelude| mergesort(_, []) = []
Prelude| mergesort(_, [x]) = [x]
Prelude| mergesort(compop, lat) =
Prelude|     let
Prelude|         mergesort1([]) = []
Prelude|         mergesort1([x]) = [x]
Prelude|         mergesort1(lat1) =
Prelude|             let
Prelude|                 split([]) = ([], [])
Prelude|                 split([x]) = ([], [x])
Prelude|                 split(x:y:excess) =
Prelude|                     let
Prelude|                         (left, right) = split(excess)
Prelude|                     in
Prelude|                         (x:left, y:right)
Prelude|
Prelude|         merge(l, []) = l
Prelude|         merge([], l) = l
Prelude|         merge(l:ls, r:rs) =
Prelude|             if compop(l, r) then l:merge(ls, r:rs)
Prelude|             else r:merge(l:ls, rs)
Prelude|
Prelude|         -- split it
Prelude|         (left, right) = split(lat1)
Prelude|
Prelude|         -- mergesort each side
Prelude|         leftsorted = mergesort1(left)
Prelude|         rightsorted = mergesort1(right)
Prelude|     in
Prelude|         -- merge
Prelude|         merge(leftsorted, rightsorted)
Prelude|     in
Prelude|         mergesort1(lat)
Prelude| :}
Prelude>
Prelude> :type mergesort
mergesort :: ((a, a) -> Bool, [a]) -> [a]

```

Notice also that we factored the argument `compop` out of the function `merge` in this version since it is visible from an outer scope.

C.10 Declaring Types

The reader may have noticed in the previous examples that Haskell infers the types of values (e.g., lists, tuples, and functions) that have not been explicitly declared by the programmer to be of a particular type with the `::` operator.

C.10.1 Inferred or Deduced

The following transcript demonstrates type inference.

```
Prelude> ans1 = [1,2,3]
Prelude>
Prelude> :type ans1
ans1 :: Num a => [a]
Prelude>
Prelude> ans2 = (1, "Mary", 3.76)
Prelude>
Prelude> :type ans2
ans2 :: (Fractional c, Num a) => (a, [Char], c)
Prelude>
Prelude> square(x) = x*x
Prelude>
Prelude> :type square
square :: Num a => a -> a
Prelude>
Prelude> square(2)
4
Prelude> square(2.0)
4.0
Prelude> :{
Prelude| reverse3([]) = []
Prelude| reverse3(h:t) = reverse3(t) ++ [h]
Prelude| :}
Prelude>
Prelude> :type reverse
reverse :: [a] -> [a]
```

C.10.2 Explicitly Declared

The following transcript demonstrates the use of explicitly declared types.

```
Prelude> :{
Prelude| ans1 :: [Integer]
Prelude| ans1 = [1,2,3]
Prelude| :}
Prelude>
Prelude> :type ans1
ans1 :: [Integer]
Prelude>
Prelude> :{
Prelude| ans2 :: (Integer, String, Float)
Prelude| ans2 = (1, "Mary", 3.76)
```

```

Prelude| :}
Prelude>
Prelude> :type ans2
ans2 :: (Integer, String, Float)
Prelude>
Prelude> :{
Prelude| square :: Int -> Int
Prelude| square(x) = x*x
Prelude| :}
Prelude>
Prelude> :type square
square :: Int -> Int
Prelude>
Prelude> :type square(2)
square(2) :: Int
Prelude>
Prelude> :type square(2.0)

<interactive>:1:8: error:
  No instance for (Fractional Int) arising from the literal '2.0'
  In the first argument of 'square', namely '(2.0)'
  In the expression: square (2.0)
Prelude>
Prelude> :{
Prelude| reverse3 :: [Int] -> [Int]
Prelude| reverse3([]) = []
Prelude| reverse3(h:t) = reverse3(t) ++ [h]
Prelude| :}
Prelude>
Prelude> :type reverse3
reverse3 :: [Int] -> [Int]
Prelude>
Prelude> reverse3([1,2,3,4,5])
[5,4,3,2,1]
Prelude>
Prelude> reverse3([1.1,2.2,3.3,4.4,5.5])

<interactive>:37:11: error:
  No instance for (Fractional Int) arising from the literal '1.1'
  In the expression: 1.1
  In the first argument of 'reverse3', namely
    '([1.1, 2.2, 3.3, 4.4, 5.5])'
  In the expression: reverse3 ([1.1, 2.2, 3.3, 4.4, 5.5])

```

Programming Exercises for Appendix C

Exercise C.1 Define a recursive Haskell function `remove` that accepts only a list and an integer `i` as arguments and returns another list that is the same as the input list, but with the `i`th element of the input list removed. If the length of the input list is less than `i`, return the same list. Assume that `i = 1` refers to the first element of the list.

Examples:

```

Prelude> remove(1, [9,10,11,12])
[10,11,12]
Prelude> remove(2, [9,10,11,12])
[9,11,12]

```



```
Prelude> remove(3, [9,10,11,12])
[9,10,12]
Prelude> remove(4, [9,10,11,12])
[9,10,11]
Prelude> remove(5, [9,10,11,12])
[9,10,11,12]
```

Exercise C.2 Define a Haskell function called `makeset` that accepts only a list of integers as input and returns the list with any repeating elements removed. The order in which the elements appear in the returned list does not matter, as long as there are no duplicate elements. Do not use any user-defined auxiliary functions, except `elem`.

Examples:

```
Prelude> makeset([1,3,4,1,3,9])
[4,1,3,9]
Prelude> makeset([1,3,4,9])
[1,3,4,9]
Prelude> makeset(["apple","orange","apple"])
["orange","apple"]
```

Exercise C.3 Define a Haskell function `cycle1` that accepts only a list and an integer `i` as arguments and cycles the list `i` times. Do not use any user-defined auxiliary functions.

Examples:

```
Prelude> cycle1(0, [1,4,5,2])
[1,4,5,2]
Prelude> cycle1(1, [1,4,5,2])
[4,5,2,1]
Prelude> cycle1(2, [1,4,5,2])
[5,2,1,4]
Prelude> cycle1(4, [1,4,5,2])
[1,4,5,2]
Prelude> cycle1(6, [1,4,5,2])
[5,2,1,4]
Prelude> cycle1(10, [1])
[1]
Prelude> cycle1(9, [1,4])
[4,1]
```

Exercise C.4 Define a Haskell function `transpose` that accepts a list as its only argument and returns that list with adjacent elements transposed. Specifically, `transpose` accepts an input list of the form $[e_1, e_2, e_3, e_4, e_5, e_6 \cdots, e_n]$ and returns a list of the form $[e_2, e_1, e_4, e_3, e_6, e_5, \cdots, e_n, e_{n-1}]$ as output. If n is odd, e_n will continue to be the last element of the list. Do not use any user-defined auxiliary functions and do not use `++` (i.e., append).

Examples:

```
Prelude> transpose([1,2,3,4])
[2,1,4,3]
```

```

Prelude> transpose ([1,2,3,4,5,6])
[2,1,4,3,6,5]
Prelude> transpose ([1,2,3])
[2,1,3]

```

Exercise C.5 Define a Haskell function `oddevensum` that accepts only a list of integers as an argument and returns a pair consisting of the sum of the odd and even positions of the list. Do not use any user-defined auxiliary functions.

Examples:

```

Prelude> oddevensum([])
(0,0)
Prelude> oddevensum([6])
(6,0)
Prelude> oddevensum([6,3])
(6,3)
Prelude> oddevensum([6,3,8])
(14,3)
Prelude> oddevensum([1,2,3,4])
(4,6)
Prelude> oddevensum([1,2,3,4,5,6])
(9,12)
Prelude> oddevensum([1,2,3])
(4,2)

```

Exercise C.6 Define a Haskell function `permutations` that accepts only a list representing a set as an argument and returns a list of all permutations of that list as a list of lists. You will need to define some nested auxiliary functions. Try to define only one auxiliary function and pass a λ -function to `map` within the body of that function and within the body of the `permutations` function to simplify their definitions. *Hint:* Use the built-in Haskell function `concat`.

Examples:

```

Prelude> permutations([])
[]
Prelude> permutations([1])
[[1]]
Prelude> permutations([1,2])
[[1,2],[2,1]]
Prelude> permutations([1,2,3])
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
Prelude> permutations([1,2,3,4])
[[1,2,3,4],[1,2,4,3],[1,3,2,4],[1,3,4,2],[1,4,2,3],
 [1,4,3,2],[2,1,3,4],[2,1,4,3],[2,3,1,4],[2,3,4,1],
 [2,4,1,3],[2,4,3,1],[3,1,2,4],[3,1,4,2],[3,2,1,4],
 [3,2,4,1],[3,4,1,2],[3,4,2,1],[4,1,2,3],[4,1,3,2],
 [4,2,1,3],[4,2,3,1],[4,3,1,2],[4,3,2,1]]
Prelude> permutations(["oranges", "and", "tangerines"])
[["oranges","and","tangerines"], ["oranges","tangerines","and"],
 ["and","oranges","tangerines"], ["and","tangerines","oranges"],
 ["tangerines","oranges","and"], ["tangerines","and","oranges"]]

```

Hint: This solution requires approximately 10 lines of code.

C.11 Thematic Takeaways

- While a goal of the functional style of programming is to bring programming closer to mathematics, Haskell and its syntax, as well as the responses of the Haskell interpreter (particularly for tuples and functions), make the connection between functional programming and mathematics salient.
- Native support for pattern-directed invocation is one of the most convenient features of user-defined functions in Haskell because it obviates the need for an `if-then-else` expression to differentiate between the various inputs to a function.
- Use of pattern-directed invocation (i.e., pattern matching) introduces declarative programming into Haskell.
- Pattern-directed invocation is not operator/function overloading.
- Operator/function overloading (sometimes called *ad hoc polymorphism*) is not *parametric polymorphism*.

C.12 Appendix Summary

Haskell is a statically typed and type-safe programming language that primarily supports functional programming. Haskell uses homogeneous lists with list operators `:` (i.e., cons) and `++` (i.e., append). The language supports *anonymous*/ λ *functions* (i.e., unnamed or literal functions). A key language concept in Haskell is that all functions have types. Another key language concept in Haskell is *pattern-directed invocation*—a pattern-action rule-oriented style of programming, involving *pattern matching*, for defining and invoking functions. This appendix provides an introduction to Haskell so that readers can explore type concepts of programming languages and lazy evaluation through Haskell in Chapters 7–9, and 12. Table 9.7 compares the main concepts in Standard ML and Haskell.

C.13 Notes and Further Reading

Haskell is a descendant of the programming language Miranda, which sprang from a series of purely functional languages developed by David A. Turner in the late 1970s and 1980s. Haskell is a result of the efforts of a committee in the late 1980s to consolidate the existing lazy, purely functional languages into a standard intended to serve as the basis for future research in the design of functional programming languages. While designed by committee, Haskell was developed primarily at Yale University and University of Glasgow.

Appendix D

Getting Started with the Camille Programming Language

CAMILLE is a programming language, inspired by Friedman, Wand, and Haynes (2001), for learning the concepts and implementation of computer languages through the development of a series of interpreters for it written in Python (Perugini and Watkin 2018). In Chapters 10–12 of this text, we implement a variety of an environment-passing interpreters for Camille, in the tradition of Friedman, Wand, and Haynes (2001), in Python.

D.1 Appendix Objective

This appendix is a guide to getting started with Camille and includes details of its syntax and semantics, how to acquire access to the Camille Git repository necessary for using Camille, and the pedagogical approach to using the language.

D.2 Grammar

The grammar in EBNF for Camille (version 4.0) is given in Figure D.1. Comments in Camille programs begin with three consecutive dashes (i.e., `---`) and continue to the end of the line. Multi-line comments are not supported. Comments are ignored by the Camille scanner. Camille can be used for functional or imperative programming, or both. To use it for functional programming, use the `<program> ::= <expression>` grammar rule; to use it for imperative programming, use the `<program> ::= <statement>` rule.

```

<program> ::= <expression>
<program> ::= <statement>
<expression> ::= <number> | <string>
<expression> ::= <identifier>
<expression> ::= if <expression> <expression> else <expression>
<expression> ::= let {<identifier> = <expression>}+ in <expression>
<expression> ::= let* {<identifier> = <expression>}+ in <expression>
<expression> ::= <primitive> ({<expression>}+(⋅))
  <primitive> ::= + | - | * | incl | decl | zero? | eqv? |
                array | arrayreference | arrayassign
<expression> ::= fun ({<identifier>}+(⋅)) <expression>
<expression> ::= (<expression> {<expression>}+(⋅))
<expression> ::= letrec {<identifier> = <function>}+ in <expression>
<expression> ::= assign! <identifier> = <expression>
<statement> ::= <identifier> = <expression>
<statement> ::= writeln (<expression>)
<statement> ::= {{<statement>}+(⋅)}
<statement> ::= if <expression> <statement> else <statement>
<statement> ::= while <expression> do <statement>
<statement> ::= variable {<identifier>}+(⋅) ; <statement>

```

Figure D.1 The grammar in EBNF for the Camille programming language (Perugini and Watkin 2018).

User-defined functions are first-class entities in Camille. This means that a function can be the return value of an expression (i.e., an expressed value), can be bound to an identifier and stored in the environment of the interpreter (i.e., a denoted value), and can be passed as an argument to a function. As the production rules in Figure D.1 indicate, Camille supports side effect (through variable assignment) and arrays. The primitives `array`, `arrayreference`, and `arrayassign` create an array, dereference an array, and update an array, respectively. While we have multiple versions of Camille, each supporting varying concepts, in version 4.0

expressed value	=	integer $\cup$ string $\cup$ closure
denoted value	=	reference to an expressed value

Thus, akin to Java or Scheme, all denoted values are references, but are implicitly dereferenced. For more details of the language, we refer the reader to Perugini and Watkin (2018). See Appendix E for the individual grammars for the progressive versions of Camille.

D.3 Installation

To install the environment necessary for running Camille, follow these steps:

1. Install Python v3.8.5 or later.
2. Install PLY v3.11 or later.
3. Clone the latest Camille repository.

The following series of commands demonstrates the installation of the packages (using the `apt` package manager) necessary to use Camille:

```
$ sudo apt install python3
$ sudo apt install python3-pip
$ sudo python3 -m pip install ply
$ git clone \
  https://bitbucket.org/camilleinterpreter/camille-interpreter-in-python-release.git
```

D.4 Git Repository Structure and Setup

The release versions of the Camille interpreters in Python are available in a Git repository in BitBucket at <https://bitbucket.org/camilleinterpreter/camille-interpreter-in-python-release/>. The repository is organized into the following main subdirectories, indicating the recommended order in which instructors and students should explore them:

Directory in Repository	Description
0.x_FRONTEND_Chapter3_Parsing_and_Chapter9_DataAbstraction	Front end syntactical analyzer for the language
1.x_INTRODUCTION_Chapter10_Conditionals	Interpreters with support for local binding and conditionals
2.x_INTERMEDIATE_Chapter11_Functions	Interpreters with support for functions and closures
3.x_ADVANCED_Chapter12_ParameterPassing	Interpreters with support for a variety of parameter-passing mechanisms, including lazy evaluation
4.x_IMPERATIVE_Chapter12_ParameterPassing	Interpreters with support statements and sequential execution

Each subdirectory contains a `README.md` file indicating the recommended order in which instructors should explore the individual interpreters.

D.5 How to Use Camille in a Programming Languages Course

D.5.1 Module 0: Front End (Scanner and Parser)

The first place to start is with the front end of the interpreter, which contains the scanner (i.e., lexical analyzer) and parser (i.e., syntactic analyzer). The scanner and parser for Camille were developed using Python Lex-Yacc (PLY v3.11)—a scanner/parser generator for Python—and have been tested in Python 3.8.5. For the details of PLY, see <http://www.dabeaz.com/ply/>. The use of a scanner/parser generator facilitates an incremental development approach, which leads to a malleable interpreter/language. Thus, the following components can be given directly to students as is:

Description	File or Directory in Repository
Camille installation instructions	README.md
Scanner for Camille	0.x_FRONTEND_Chapter3_Parsing_and_Chapter9_DataAbstraction/Chapter3_Scanner/
Parser for Camille	0.x_FRONTEND_Chapter3_Parsing_and_Chapter9_DataAbstraction/Chapter3_Parser/
AST for Camille	0.x_FRONTEND_Chapter3_Parsing_and_Chapter9_DataAbstraction/Chapter10_DataAbstraction/

D.5.2 Chapter 10 Module: Introduction (Local Binding and Conditionals)

Given the parser, students typically begin by implementing only primitive operations, with the exception of array manipulations (Figure D.1; 1.x_INTRODUCTION_Chapter10_Conditionals/simpleinterpreter). Then, students develop an `evaluate_expr` function that accepts an expression and an environment as argument, evaluates the passed expression in the passed environment, and returns the result. This function, which is at the heart of any interpreter, constitutes a large conditional structure based on the type of expression passed (e.g., a variable reference or function definition).

Adding a support for a new concept or feature to the language typically involves adding a new grammar rule (in `camilleparse.py`) and/or primitive (in `camillelib.py`), adding a new field to the abstract-syntax representation of an expression (in `camilleinterpreter.py`), and adding a new case to the `evaluate_expr` function (in `camilleinterpreter.py`)—a theme running through Chapter 3 of *Essentials of Programming Languages* (Friedman, Wand, and Haynes 2001). All of the explorable concepts in the purview of interpreter building for this language are shown in Table D.1. Note that not all implementation options are available for use with the nameless environment.

Therefore, students start by adding support for conditional evaluation and local binding. Support for local binding requires a lookup environment, which leads to the possibility of testing a variety of representations for that environment, as long as it adheres to the well-defined interface used by `evaluate_expr`. From

Interpreter Design Options			Language Semantic Options			
Type of Environment	Representation of Environment	Representation of Functions	Scoping Method	Environment Binding	Parameter Passing Mechanism	
named	abstract syntax	abstract syntax	static	deep	by value	
nameless	list of lists	closure	dynamic		by reference	
	closure	by value-result				
		by name (lazy evaluation)				
		by need (lazy evaluation)				

Table D.1 Configuration Options in Camille (Perugini and Watkin 2018)

there, students add support for non-recursive functions, which raises the issue of how to represent a function and there are a host of options from which to choose.

In what follows, each directory corresponds to the different (progressive) version of the interpreter:

Interpreter Description	Version	Directory in Repository
simple interpreter with primitives	1.0	1.x_INTRODUCTION_Chapter10_Conditionals/simpleinterpreter/
local binding and conditionals	1.2	1.x_INTRODUCTION_Chapter10_Conditionals/localbindingconditional/

Each individual interpreter directory contains its own `README.md` describing the highlights of the particular version of the interpreter in that the directory.

D.5.3 Configuring the Language

Table D.1 enumerates the configuration options available in Camille for aspects of the design of the interpreter (e.g., choice of representation of referencing environment) as well as for the semantics of implemented concepts (e.g., choice of parameter-passing mechanism). As we vary the latter, we get a different version of the language (Table D.2).

The configuration file (i.e., `camilleconfig.py`) allows the user to switch between different representations of closure (e.g., Camille closure, abstract syntax, or Python closure) and the environment structure (e.g., closure, list of lists, or abstract syntax), as well as modify the verbosity of output from the interpreter. These parameters can be adjusted by setting `__closure_switch__`, `__env_switch__`, and `__debug_mode__`, respectively, to the appropriate value. The `detailed_debug` flag is intended to be used to debug the interpreter, while the `simple_debug` flag is intended to be used in normal operation (i.e., running and debugging Camille programs). [The nameless environments are available for use with neither the interpreter supporting dynamic scoping nor any of the interpreters in Chapter 12 (i.e., `3.x_ADVANCED_Chapter12_ParameterPassing`

and 4.x_IMPERATIVE_Chapter12_ParameterPassing). Furthermore, not all environment representations are available with all implementation options. For instance, all of the interpreters in Chapter 12 use exclusively the named ASR environment.]

```
$ pwd
camille-interpreter-in-python-release
$ cd pass-by-value-recursive
$ cat camilleconfig.py
...
...
closure_closure = 0 #static scoping our closure representation of closures
asr_closure = 1 # static scoping our asr representation of closures
python_closure = 2 # dynamic scoping python representation of closures
__closure_switch__ = asr_closure # for lexical scoping
#__closure_switch__ = python_closure # for dynamic scoping

closure = 1
asr = 2
lovr = 3
__env_switch__ = lovr

detailed_debug = 1 # full stack trace through Python exception
simple_debug = 2 # camille interpreter output only
__debug_mode__ = simple_debug
$
```

At this point, students can also explore implementing dynamic scoping as an alternative to the default static scoping. This amounts to little more than storing the calling environment, rather than the lexically enclosing environment, in the representation of the function. This is configured through the configuration file identified previously.

D.5.4 Chapter 11 Module: Intermediate (Functions and Closures)

Next, students implement recursive functions, which require a modified environment. At this point, students have implemented Camille version 2.1—a language supporting only (pure) functional programming—and explored the use of multiple configuration options for both aspects of the design of the interpreter and the semantics of implemented concepts (Table D.2).

Interpreter Description	Version	Directory in Repository
non-recursive functions using pass-by-value	2.0	2.x_INTERMEDIATE_Chapter11_Functions/pass-by-value-non-recursive/
recursive functions using pass-by-value	2.1	2.x_INTERMEDIATE_Chapter11_Functions/pass-by-value-recursive/

	Version of Camille	1.x	2.x	3.x	4.x
Design Choices	Expressed Values	ints	ints $\cup$ cls	ints $\cup$ cls	ints $\cup$ cls
	Denoted Values	ints	ints $\cup$ cls	references to expressed values	references to expressed values
	Representation of Environment	N/A	ASR CLS LOLR	ASR	ASR
	Representation of Closures	N/A	ASR CLS	ASR CLS	ASR CLS
Language Semantic Options	Representation of References	N/A	N/A	ASR	ASR
	Local Binding	$\uparrow$ let, let* $\uparrow$	$\uparrow$ let, let* $\uparrow$	$\uparrow$ let, let* $\uparrow$	$\uparrow$ let, let* $\uparrow$
	Conditionals	$\downarrow$ if/else $\downarrow$	$\downarrow$ if/else $\downarrow$	$\downarrow$ if/else $\downarrow$	$\downarrow$ if/else $\downarrow$
	Non-recursive Functions	$\times$	$\uparrow$ fun $\uparrow$	$\uparrow$ fun $\uparrow$	$\uparrow$ fun $\uparrow$
	Recursive Functions	$\times$	$\uparrow$ letrec $\uparrow$	$\uparrow$ letrec $\uparrow$	$\uparrow$ letrec $\uparrow$
	Scoping	N/A	lexical	lexical	lexical
	Environment Bound to Closure	N/A	deep	deep	deep
	References	$\times$	$\times$	$\checkmark$	$\checkmark$
	Parameter Passing	N/A	$\uparrow$ by value $\uparrow$	$\uparrow$ by reference/lazy $\uparrow$	$\uparrow$ by value $\uparrow$
	Side Effects	$\times$	$\times$	$\uparrow$ assign! $\uparrow$	$\downarrow$ multiple $\downarrow$
	Statement Blocks	N/A	N/A	N/A	$\checkmark$
	Repetition	N/A	recursion	recursion	$\downarrow$ while $\downarrow$

Table D.2 Design Choices and Implemented Concepts in Progressive Versions of Camille. The symbol $\downarrow$ indicates that the concept is supported through its implementation in the defining language (here, Python). The Python keyword included in each cell, where applicable, indicates which Python construct is used to implement the feature in Camille. The symbol $\uparrow$ indicates that the concept is implemented manually. The Camille keyword included in each cell, where applicable, indicates the syntactic construct through which the concept is operationalized. (Key: ASR = abstract-syntax representation; CLS = closure; and LOLR = list-of-list representation. Cells in boldface font highlight the enhancements across the versions.) Reproduced from Perugini, S., and J. L. Watkin. 2018. “ChAmEIoN: A Customizable Language for Teaching Programming Languages.” *Journal of Computing Sciences in Colleges* 34(1): 44–51.

D.5.5 Chapter 12 Modules: Advanced (Parameter Passing, Including Lazy Evaluation) and Imperative (Statements and Sequential Evaluation)

Next, students start slowly to morph Camille, through its interpreter, into a language with imperative programming features by adding provisions for side effect (e.g., through variable assignment). Variable assignment requires a modification of the representation of the environment. Now, the environment must store references to expressed values, rather than the expressed values themselves. This raises the issue of implicit versus explicit dereferencing, and naturally leads to exploring a variety of parameter-passing mechanisms (e.g., pass-by-reference or pass-by-name/lazy evaluation). Finally, students close the loop on the imperative approach by eliminating the need to use recursion for repetition by instrumenting the language, through its interpreter, to support sequential execution of statements. This involves adding support for statement blocks, while loops, and I/O operations. Since this module involves modifications to the environment, we exclusively use the named ASR environment in this module to simplify matters.

Interpreter Description	Version	Directory in Repository
variable assignment (i.e., side effect)	3.0	3.x_ADVANCED_Chapter12_ParameterPassing/assignment/
pass-by-reference parameter passing	3.1	3.x_ADVANCED_Chapter12_ParameterPassing/pass-by-reference/
lazy Camille supporting pass-by-name/need parameter passing	3.2	3.x_ADVANCED_Chapter12_ParameterPassing/lazy-fun-arguments-only/
imperative Camille with statements and sequential execution	4.0	4.x_IMPERATIVE_Chapter12_ParameterPassing/imperative/

D.6 Example Usage: Non-interactively and Interactively (CLI)

Once students have some experience implementing language interpreters, they can begin to discern how to use the language itself to support features that are not currently supported in the interpreter. For instance, prior to supporting recursive functions in Camille, students can simulate support for recursion by passing a function to itself:

```
$ pwd
camille-interpreter-in-python-release
```

```

$
$ cd pass-by-value-non-recursive
$
$ # running the interpreter non-interactively
$
$ cat recursionUnbound.cam
let
  sum = fun (x) if zero?(x) 0 else +(x, (sum decl(x)))
in
  (sum 5)
$
$ ./run recursionUnbound.cam
Runtime Error: Line 2: Unbound Identifier 'sum'
$
$ cat recursionBound.cam
let
  sum = fun (s, x) if zero?(x) 0 else +(x, (s s,decl(x)))
in
  (sum sum, 5)
$
$ ./run recursionBound.cam
15
$
$ # running the interpreter interactively (CLI)
$
$ ./run
Camille> let
      sum = fun (x) if zero?(x) 0 else +(x, (sum decl(x)))
      in
        (sum 5)

Runtime Error: Line 2: Unbound Identifier 'sum'

Camille> let
      sum = fun (s, x) if zero?(x) 0 else +(x, (s s,decl(x)))
      in
        (sum sum, 5)

15

```

Other example programs, including an example more faithful to the tenets of object orientation, especially encapsulation, are available in the Camille Git repository at <https://bitbucket.org/camilleinterpreter/camille-interpreter-in-python-release/>. These programs demonstrate that we can create object-oriented abstractions from within the Camille language.

D.7 Solutions to Programming Exercises in Chapters 10–12

A separate Git repository in BitBucket reserved for the solutions to Programming Exercises in Chapters 10–12, available only to instructors by request, contains the versions of the interpreter in Table D.3:

Interpreter Description	Version	Directory in Repository
named ASR environment interpreter with local binding and conditionals	1.2(named ASR) / PE 10.1	1.x_INTRODUCTION_Chapter10_Conditionals/named-asr-localbinding-conditionals/
named LOLR environment interpreter with local binding and conditionals	1.2(named LOLR) / PE 10.2	1.x_INTRODUCTION_Chapter10_Conditionals/named-lolr-localbinding-conditionals/
nameless environment interpreter with local binding and conditionals	1.2(nameless) / PE 10.3–10.5	1.x_INTRODUCTION_Chapter10_Conditionals/nameless-localbinding-conditionals/
nameless environment interpreter with non-recursive functions	2.0(nameless) / PE 11.2.9– 11.2.11	2.x_INTERMEDIATE_Chapter11_Functions/nameless-pass-by-value-non-recursive/
nameless environment interpreter with recursive functions	2.1(nameless) / PE 11.3.6–11.3.8	2.x_INTERMEDIATE_Chapter11_Functions/nameless-pass-by-value-recursive/
dynamic scoping for non-recursive functions	2.0(dynamic scoping) / PE 11.2.12	2.x_INTERMEDIATE_Chapter11_Functions/pass-by-value-non-recursive-dynamic-scoping/
dynamic scoping for recursive functions	2.1(dynamic scoping) / PE 11.3.9	2.x_INTERMEDIATE_Chapter11_Functions/pass-by-value-non-recursive-dynamic-scoping/
cells	3.0(cells) / PE 12.2.3	3.0_ADVANCED/cells/
arrays	3.0(arrays) / PE 12.2.4	3.0_ADVANCED/arrays/
pass-by-value-result parameter passing	3.0(pass-by-value- result) / PE 12.4.1	3.x_ADVANCED_Chapter12_ParameterPassing/pass-by-value-result/
Camille with lazy lets	3.2(lazy let) / PE 12.6.1	3.x_ADVANCED_Chapter12_ParameterPassing/lazy-fun-arguments-lets-only/
full lazy Camille with lazy primitives and if primitive	3.2(full lazy) / PE 12.6.2	3.x_ADVANCED_Chapter12_ParameterPassing/lazy-full/
do while loop	4.0(do while) / PE 12.7.1	4.x_IMPERATIVE_Chapter12_ParameterPassing/dowhile/

Table D.3 Solutions to the Camille Interpreter Programming Exercises in Chapters 10–12

D.8 Notes and Further Reading

This appendix is based on Perugini and Watkin (2018). An extended version of this appendix in Markdown is available at <https://bitbucket.org/camilleinterpreter/camille-interpreter-in-python-release/src/master/GUIDE/README.md>.

Appendix E

Camille Grammar and Language

W^E showcase the syntax, with some semantic annotations, of the Camille programming language in this appendix.

E.1 Appendix Objective

The objective of this appendix is to catalog the grammars and syntax (with some semantic annotations) of the major versions of Camille used and distributed throughout Part III of this text in one central location.

E.2 Camille 0.1: Numbers and Primitives

Comments

Camille has only a single-line comment, which consists of three consecutive dashes (i.e., ---) followed by any number of characters up to the next newline character.

Identifiers

Identifiers in Camille are described by the following regular expression: `[_a-zA-Z][_a-zA-Z0-9*?!]*`. However, an identifier cannot be a reserved word in the language (e.g., `let`).

Syntax

The following is a context-free grammar in EBNF for version 1.0 of the Camille programming language through Chapter 10:


```

<program> ::= <expression>

               ntNumber
<expression> ::= <number>

               ntPrimitive_op
<expression> ::= <primitive> ({<expression>}+(,))

               ntPrimitive
<primitive> ::= + | - | * | incl | decl | zero? | eqv?

```

Semantics

Currently,

```

expressed value = integer
denoted value  = integer

```

Thus,

expressed value = denoted value = integer

E.3 Camille 1.X: Local Binding and Conditional Evaluation

Syntax

The following is a context-free grammar in EBNF for versions 1.X of the Camille programming language through Chapter 10:

```

<program> ::= <expression>

               ntNumber
<expression> ::= <number>

               ntIdentifier
<expression> ::= <identifier>

               ntPrimitive_op
<expression> ::= <primitive> ({<expression>}+(,))

               ntPrimitive
<primitive> ::= + | - | * | incl | decl | zero? | eqv?

               ntIfElse
<expression> ::= if <expression> <expression> else <expression>

```

```

ntLet
<expression> ::= let {<identifier> = <expression>}+ in <expression>

ntLetStar
<expression> ::= let* {<identifier> = <expression>}+ in <expression>

```

Semantics

```

expressed value = integer
denoted value  = integer

expressed value = denoted value = integer

```

E.4 Camille 2.X: Non-recursive and Recursive Functions

Syntax

The following is a context-free grammar in EBNF for versions 2.X of the Camille programming language through Chapter 11:

```

<program> ::= <expression>

ntNumber
<expression> ::= <number>

ntIdentifier
<expression> ::= <identifier>

ntPrimitive_op
<expression> ::= <primitive> ({<expression>}+(,))

ntPrimitive
<primitive> ::= + | - | * | incl | decl | zero? | eqv?

ntIfElse
<expression> ::= if <expression> <expression> else <expression>

ntLet
<expression> ::= let {<identifier> = <expression>}+ in <expression>

ntLetStar
<expression> ::= let* {<identifier> = <expression>}+ in <expression>

ntFuncDecl
<expression> ::= fun ({<identifier>}*(,)) <expression>

```

ntFuncCall

$\langle \text{expression} \rangle ::= (\langle \text{expression} \rangle \{ \langle \text{expression} \rangle \}^* (\cdot))$

ntLetRec

$\langle \text{expression} \rangle ::= \text{letrec} \{ \langle \text{identifier} \rangle = \langle \text{function} \rangle \}^+ \text{in } \langle \text{expression} \rangle$

Semantics

We desire user-defined functions to be first-class entities in Camille. This means that a function can be the return value of an expression (altering the expressed values) and can be bound to an identifier and stored in the environment of the interpreter (altering the denoted values). Adding user-defined, first-class functions to Camille alters its expressed and denoted values:

expressed value	=	integer $\cup$ closure
denoted value	=	integer $\cup$ closure

Thus,

$$\text{expressed value} = \text{denoted value} = \text{integer} \cup \text{closure}$$

Recall, previously in Chapter 10 we had

$$\text{expressed value} = \text{denoted value} = \text{integer}$$

E.5 Camille 3.X: Variable Assignment and Support for Arrays

The following is a context-free grammar in EBNF for versions 3.X of the Camille programming language through Chapter 12:

Syntax

ntAssignment

$\langle \text{expression} \rangle ::= \text{assign! } \langle \text{identifier} \rangle = \langle \text{expression} \rangle$
 $\langle \text{primitive} \rangle ::= + \mid - \mid * \mid \text{inc1} \mid \text{dec1} \mid \text{zero?} \mid \text{eqv?} \mid$
 $\text{array} \mid \text{arrayreference} \mid \text{arrayassign}$

Semantics

With the addition of references, now in Camille

expressed value	=	integer $\cup$ closure
denoted value	=	reference to an expressed value

Thus,

denoted value != (expressed value = integer $\cup$ closure)

Also, the array creation, access, and modification primitives have the following semantics:

- `array`: creates an array
- `arrayreference`: dereferences an array
- `arrayassign`: updates an array

E.6 Camille 4.X: Sequential Execution

Syntax

The following is a context-free grammar in EBNF for versions 4.X of the Camille programming language through Chapter 12:

```

<program> ::= <statement>
               ntAssignmentStmt
<statement> ::= <identifier> = <expression>
               ntOutputStmt
<statement> ::= writeln (<expression>)
               ntCompoundStmt
<statement> ::= { {<statement>}*(i) }
               ntIfElseStmt
<statement> ::= if <expression> <statement> else <statement>
               ntWhileStmt
<statement> ::= while <expression> do <statement>
               ntBlockStmt
<statement> ::= variable {<identifier>}*(,) ; <statement>

```

Semantics

Thus far Camille is an *expression-oriented language*. We now implement the Camille interpreter to define a *statement-oriented language*. We want to retain:

expressed value = integer $\cup$ closure
denoted value = reference to an expressed value

Bibliography

- Abelson, H., and G. J. Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, MA: MIT Press.
- Aho, A. V., R. Sethi, and J. D. Ullman. 1999. *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley.
- Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. 1977. *A Pattern Language: Towns, Buildings, Construction*. New York, NY: Oxford University Press.
- Appel, A. W. 1992. *Compiling with Continuations*. Cambridge, UK: Cambridge University Press.
- Appel, A. W. 1993. "A Critique of Standard ML." *Journal of Functional Programming* 3 (4): 391–429.
- Appel, A. W. 2004. *Modern Compiler Implementation in ML*. Cambridge, UK: Cambridge University Press.
- Arabnia, H. R., L. Deligiannidis, M. R. Grimaila, D. D. Hodson, and F. G. Tinetti. 2019. *CSC'19: Proceedings of the 2019 International Conference on Scientific Computing*. Las Vegas, NV: CSREA Press.
- Backus, J. 1978. "Can Programming Be Liberated from the von Neumann Style?: A Functional Style and Its Algebra of Programs." *Communications of the ACM* 21 (8): 613–641.
- Bauer, F. L., and J. Eickel. 1975. *Compiler Construction: An Advanced Course*. New York, NY: Springer-Verlag.
- Boole, G. 1854. *An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities*. London, UK: Walton and Maberly.
- Carroll, L. 1958. *Symbolic Logic and the Game of Logic*. Mineola, NY: Dover Publications.
- Christiansen, T., b. d. foy, L. Wall, and J. Orwant. 2012. *Programming Perl: Unmatched Power for Text Processing and Scripting*. 4th ed. Sebastopol, CA: O'Reilly Media.
- Codognet, P., and D. Diaz. 1995. "wamcc: Compiling Prolog to C." In *Proceedings of Twelfth International Conference on Logic Programming (ICLP)*, 317–331.

- Computing Curricula 2020 Task Force. 2020. *Computing Curricula 2020: Paradigms for Global Computing Education*. Technical report. Association for Computing Machinery and IEEE Computer Society. Accessed March 26, 2021. <https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2020.pdf>.
- Conway, M. E. 1963. "Design of a Separable Transition-Diagram Compiler." *Communications of the ACM* 6 (7): 396–408.
- Coyle, C., and P. Crogono. 1991. "Building Abstract Iterators Using Continuations." *ACM SIGPLAN Notices* 26 (2): 17–24.
- Dijkstra, E. W. 1968. "Go To Statement Considered Harmful." *Communications of the ACM* 11 (3): 147–148.
- Dybvig, R. K. 2003. *The Scheme Programming Language*. 3rd ed. Cambridge, MA: MIT Press.
- Dybvig, R. K. 2009. *The Scheme Programming Language*. 4th ed. Cambridge, MA: MIT Press.
- Eckroth, J. 2018. *AI Blueprints: How to Build and Deploy AI Business Projects*. Birmingham, UK: Packt Publishing.
- Feeley, M. 2004. *The 90 Minute Scheme to C Compiler*. Accessed May 20, 2020. <http://churchturing.org/y/90-min-scc.pdf>.
- Felleisen, M., R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. 2018. "A Programmable Programming Language." *Communications of the ACM* 61 (3): 62–71.
- Flanagan, D. 2005. *Java in a Nutshell*. 5th ed. Beijing: O'Reilly.
- Foderaro, J. 1991. "LISP: Introduction." *Communications of the ACM* 34 (9). <https://doi.org/10.1145/114669.114670>.
- Friedman, D. P., and M. Felleisen. 1996a. *The Little Schemer*. 4th ed. Cambridge, MA: MIT Press.
- Friedman, D. P., and M. Felleisen. 1996b. *The Seasoned Schemer*. Cambridge, MA: MIT Press.
- Friedman, D. P., and M. Wand. 2008. *Essentials of Programming Languages*. 3rd ed. Cambridge, MA: MIT Press.
- Friedman, D. P., M. Wand, and C. Haynes. 2001. *Essentials of Programming Languages*. 2nd ed. Cambridge, MA: MIT Press.
- Gabriel, R. P. 2001. "The Why of Y." Accessed March 5, 2021. <https://www.dreamsongs.com/Files/WhyOfY.pdf>.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley.
- Garcia, M., T. Gandhi, J. Singh, L. Duarte, R. Shen, M. Dantu, S. Ponder, and H. Ramirez. 2001. "Esdiabetes (an Expert System in Diabetes)." *Journal of Computing Sciences in Colleges* 16 (3): 166–175.
- Giarratano, J. C. 2008. *CLIPS User's Guide*. Cambridge, MA: MIT Press.
- Graham, P. 1993. *On Lisp*. Upper Saddle River, NJ: Prentice Hall. Accessed July 26, 2018. <http://paulgraham.com/onlisp.html>
- Graham, P. 1996. *ANSI Common Lisp*. Upper Saddle River, NJ: Prentice Hall.

- Graham, P. 2002. *The Roots of Lisp*. Accessed July 19, 2018. <http://lib.store.yahoo.net/lib/paulgraham/jmc.ps>.
- Graham, P. 2004a. "Beating the Averages." In *Hackers and Painters: Big Ideas from the Computer Age*. Beijing: O'Reilly. Accessed July 19, 2018. <http://www.paulgraham.com/avg.html>.
- Graham, P. 2004b. *Hackers and Painters: Big Ideas from the Computer Age*. Beijing: O'Reilly.
- Graham, P. n.d. *[Haskell] Pros and Cons of Static Typing and Side Effects?* <http://paulgraham.com/lispfaq1.html>; <https://mail.haskell.org/pipermail/haskell/2005-August/016266.html>
- Graham, P. n.d. *LISP FAQ*. Accessed July 19, 2018. <http://paulgraham.com/lispfaq1.html>.
- Graunke, P., R. Findler, S. Krishnamurthi, and M. Felleisen. 2001. "Automatically Restructuring Programs for the Web." In *Proceedings of the Sixteenth IEEE International Conference on Automated Software Engineering (ASE)*, 211–222.
- Harbison, S. P., and G. L. Steele Jr. 1995. *C: A Reference Manual*. 4th ed. Englewood Cliffs, NJ: Prentice Hall.
- Harmelen, F. van, and A. Bundy. 1988. "Explanation-Based Generalisation = Partial Evaluation." *Artificial Intelligence* 36 (3): 401–412.
- Harper, R. n.d.a. "Teaching FP to Freshman." Accessed July 19, 2018. <http://existentialtype.wordpress.com/2011/03/15/teaching-fp-to-freshmen/>.
- Harper, R. n.d.b. "What Is a Functional Language?" Accessed July 19, 2018. <http://existentialtype.wordpress.com/2011/03/16/what-is-a-functional-language/>.
- Haynes, C. T., and D. P. Friedman. 1987. "Abstracting Timed Preemption with Engines." *Computer Languages* 12 (2): 109–121.
- Haynes, C. T., D. P. Friedman, and M. Wand. 1986. "Obtaining Coroutines with Continuations." *Computer Languages* 11 (3/4): 143–153.
- Heeren, B., D. Leijen, and A. van IJzendoorn. 2003. "Helium, for Learning Haskell." In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, 62–71. New York, NY: ACM Press.
- Hieb, R., K. Dybvig, and C. Bruggeman. 1990. "Representing Control in the Presence of First-Class Continuations." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY: ACM Press.
- Hoare, T. 1980. *The 1980 ACM Turing Award Lecture*. <https://www.cs.fsu.edu/~engelen/courses/COP4610/hoare.pdf>
- Hofstadter, D. R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York, NY: Basic Books.
- Hughes, J. 1989. "Why Functional Programming Matters." *The Computer Journal* 32 (2): 98–107. Also appears as: Hughes, J. 1990. "Why Functional Programming Matters." In *Research Topics in Functional Programming*, edited by D. A. Turner, 17–42. Boston, MA: Addison-Wesley.
- Hutton, G. 2007. *Programming in Haskell*. Cambridge, UK: Cambridge University Press.

- Interview with Simon Peyton-Jones*. 2017. *People of Programming Languages: An interview project in conjunction with the Forty-Fifth ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2018)*. Interviewer: Jean Yang. Accessed January 20, 2021. <https://www.cs.cmu.edu/popl-interviews/peytonjones.html>.
- Iverson, K. E. 1999. *Math for the Layman*. JSoftware Inc. <https://www.jsoftware.com/books/pdf/mftl.zip>
- The Joint Task Force on Computing Curricula: Association for Computing Machinery (ACM) and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Technical report. Association for Computing Machinery and IEEE Computer Society. Accessed January 26, 2021. https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf.
- Jones, N. D. 1996. "An Introduction to Partial Evaluation." *ACM Computing Surveys* 28 (3): 480–503.
- Kamin, S. N. 1990. *Programming Languages: An Interpreter-Based Approach*. Reading, MA: Addison-Wesley.
- Kay, A. 2003. *Dr. Alan Kay on the Meaning of "Object-Oriented Programming, July 23, 2003"*. Accessed January 14, 2021. http://www.purl.org/stefan_ram/pub/doc_kay_oop_en.
- Kernighan, B. W., and R. Pike. 1984. *The UNIX Programming Environment*. 2nd ed. Upper Saddle River, NJ: Prentice Hall.
- Kernighan, B. W., and P. J. Plauger. 1978. *The Elements of Programming Style*. 2nd ed. New York, NY: McGraw-Hill.
- Knuth, D. E. 1974a. "Computer Programming as an Art." *Communications of the ACM* 17 (12): 667–673.
- Knuth, D. E. 1974b. "Structured Programming with `go to` Statements." *ACM Computing Surveys* 6 (4): 261–301.
- Kowalski, R. A. 1979. "Algorithm = Logic + Control." *Communications of the ACM* 22 (7): 424–436.
- Krishnamurthi, S. 2003. *Programming Languages: Application and Interpretation*. Accessed February 27, 2021. <http://cs.brown.edu/sk/Publications/Books/ProgLangs/2007-04-26/plai-2007-04-26.pdf>.
- Krishnamurthi, S. 2008. "Teaching Programming Languages in a Post-Linnaean Age." *ACM SIGPLAN Notices* 43 (11): 81–83.
- Krishnamurthi, S. 2017. *Programming Languages: Application and Interpretation*. 2nd ed. Accessed February 27, 2021. <http://cs.brown.edu/courses/cs173/2012/book/book.pdf>.
- Lämmel, R. 2008. "Google's MapReduce Programming Model—Revisited." *Science of Computer Programming* 70 (1): 1–30.
- Landin, P. J. 1966. "The Next 700 Programming Languages." *Communications of the ACM* 9 (3): 157–166.
- Levine, J. R. 2009. *Flex and Bison*. Cambridge, MA: O'Reilly.
- Levine, J. R., T. Mason, and D. Brown. 1995. *Lex and Yacc*. 2nd ed. Cambridge, MA: O'Reilly.

- MacQueen, D. B. 1993. "Reflections on Standard ML." In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991–1992*, McMaster University, Hamilton, Ontario, Canada, 32–46. London, UK: Springer-Verlag.
- MacQueen, D., R. Harper, and J. Reppy. 2020. "The History of Standard ML." *Proceedings of the ACM on Programming Languages* 4 (HOPL): article 86.
- Matthews, C. 1998. *An Introduction to Natural Language Processing Through Prolog*. London, UK: Longman.
- McCarthy, J. 1960. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." *Communications of the ACM* 3 (4): 184–195.
- McCarthy, J. 1981. "History of Lisp." In *History of Programming Languages*, edited by R. Wexelblat. Cambridge, MA: Academic Press.
- Miller, J. S. 1987. "Multischeme: A Parallel Processing System Based on MIT Scheme," PhD dissertation. Massachusetts Institute of Technology.
- Milner, R. 1978. "A Theory of Type Polymorphism in Programming." *Journal of Computer and System Sciences* 17:348–375.
- Muehlbauer, J. 2002. "Orbitz Reaches New Heights." *New Architect*. Accessed February 10, 2021. <https://people.apache.org/jim/NewArchitect/newarch/2002/04/new1015626014044/index.html>
- Murray, P., and L. Murray. 1963. *The Art of the Renaissance*. London, UK: Thames and Hudson.
- Niemann, T. n.d. *Lex and Yacc Tutorial*. ePaperPress. <http://epaperpress.com/lexandyacc/>.
- Parr, T. 2012. *The Definitive ANTLR4 Reference*. Dallas, TX: Pragmatic Bookshelf.
- Pereira, F. 1993. "A Brief Introduction to Prolog." *ACM SIGPLAN Notices* 28 (3): 365–366.
- Pérez-Quiñones, M. A. 1996. "Conversational Collaboration in User-Initiated Interruption and Cancellation Requests." PhD dissertation, George Washington University.
- Perlis, A. J. 1982. "Epigrams on Programming." *ACM SIGPLAN Notices* 17 (9): 7–13.
- Perugini, S., and J. L. Watkin. 2018. "ChAmElEoN: A Customizable Language for Teaching Programming Languages." *Journal of Computing Sciences in Colleges* 34 (1): 44–51.
- Peters, T. 2004. *PEP 20: The Zen of Python*. Accessed January 12, 2021. <https://www.python.org/dev/peps/pep-0020/>
- Peyton Jones, S. L. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Upper Saddle River, NJ: Prentice-Hall.
- Quan, D., D. Huynh, D. R. Karger, and R. Miller. 2003. "User Interface Continuations." In *Proceedings of the Sixteenth Annual ACM Symposium on User Interface Software and Technology (UIST)*, 145–148. New York, NY: ACM Press.
- Queinnec, C. 2000. "The Influence of Browsers on Evaluators or, Continuations to Program Web Servers." In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 23–33. New York, NY: ACM Press.

- Rich, E., K. Knight, and S. B. Nair. 2009. *Artificial Intelligence*. 3rd ed. India: McGraw-Hill India.
- Robinson, J. A. 1965. "A Machine-Oriented Logic Based on the Resolution Principle." *Journal of the ACM* 12 (1): 23–41.
- Savage, N. 2018. "Using Functions for Easier Programming." *Communications of the ACM* 61 (5): 29–30.
- Scott, M. L. 2006. *Programming Languages Pragmatics*. 2nd ed. Amsterdam: Morgan Kaufmann.
- Sinclair, K. H., and D. A. Moon. 1991. "The Philosophy of Lisp." *Communications of the ACM* 34 (9): 40–47.
- Somogyi, Z., F. Henderson, and T. Conway. 1996. "The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language." *The Journal of Logic Programming* 29:17–64.
- Sperber, M., R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews, eds. 2010. *Revised⁶ Report on the Algorithmic Language Scheme*. Cambridge, UK: Cambridge University Press.
- Sussman, G. J., and G. L. Steele Jr. 1975. "Scheme: An Interpreter for Extended Lambda Calculus." *AI Memo* 349. Accessed May 22, 2020. <https://dspace.mit.edu/handle/1721.1/5794>.
- Sussman, G. J., G. L. Steele Jr., and R. P. Gabriel. 1993. "A Brief Introduction to Lisp." *ACM SIGPLAN Notices* 28 (3): 361–362.
- Swaine, M. 2009. "It's Time to Get Good at Functional Programming: Is It Finally Functional Programming's Turn?" *Dr. Dobbs' Journal* 34 (1): 14–16.
- Thompson, S. 2007. *Haskell: The Craft of Functional Programming*. 2nd ed. Harlow, UK: Addison-Wesley.
- Ullman, J. 1997. *Elements of ML Programming*. 2nd ed. Upper Saddle River, NJ: Prentice Hall.
- Venners, B. 2003. *Python and the Programmer: A Conversation with Bruce Eckel, Part I*. Accessed July 28, 2021. <https://www.artima.com/articles/python-and-the-programmer>
- Wang, C.-I. 1990. "Obtaining Lazy Evaluation with Continuations in Scheme." *Information Processing Letters* 35 (2): 93–97.
- Warren, D. H. D. 1983. "An Abstract Prolog Instruction Set," Technical Note 309. Menlo Park, CA: SRI International.
- Watkin, J. L., A. C. Volk, and S. Perugini. 2019. "An Introduction to Declarative Programming in CLIPS and PROLOG." In *Proceedings of the 17th International Conference on Scientific Computing (CSC)*, edited by H. R. Arabnia, L. Deligianidis, M. R. Grimaila, D. D. Hodson, and F. G. Tinetti, 105–111. Computer Science Research, Education, and Applications Press (Publication of the World Congress in Computer Science, Computer Engineering, and Applied Computing (CSCE)). CSREA Press. <https://csce.ucmss.com/cr/books/2019/LFS/CSREA2019/CSC2488.pdf>.
- Webber, A. B. 2008. *Formal Languages: A Practical Introduction*. Wilsonville, OR: Franklin, Beedle and Associates.

- Weinberg, G. M. 1988. *The Psychology of Computer Programming*. New York, NY: Van Nostrand Reinhold.
- Wikström, Å. 1987. *Functional Programming Using Standard ML*. United Kingdom: Prentice Hall International.
- Wright, A. 2010. "Type Theory Comes of Age." *Communications of the ACM* 53 (2): 16–17.

Index

Note: Page numbers followed by *f* and *t* indicate figures and tables respectively.

A

- abstract data type (ADT), 337, 366
- abstract syntax, 356–359
 - programming exercises for, 364–365
 - representation in Python, 372–373
- abstract-syntax tree, 115
 - for arguments lists, 401–403
 - for Camille, 359
 - parser generator with tree builder, 360–364
 - programming exercises for, 364–365
 - TreeNode, 359–360
- abstraction, 104
 - binary search, 151–152
 - binary tree, 150–151
 - building blocks as, 174–175
 - programming exercises for, 152–153
- activation record, 201
- actual parameters, 131
- ad hoc binding, 236–238
- ad hoc polymorphism. *See* overloading; operator/function overloading
- addcf function, 298
- ADT. *See* abstract data type (ADT)
- aggregate data types
 - arrays, 338
 - discriminated unions, 343
 - programming exercises for, 343–344
 - records, 338–340
 - undiscriminated unions, 341–343

- agile methods, 25
- all-or-nothing proposition, 613–614
- alphabet, 34
- ambiguity, 52
- ambiguous grammar, 51
- ancestor blocks, 190
- antecedent, definition of, 651
- ANTLR (ANOther Tool for Language Recognition), 81
- append, primitive nature of, 675–676
- applicative-order evaluation, 493, 512
- apply_environment_
reference function, 462
- arguments. *See* actual parameters
- Armstrong, Joe, 178
- arrays, 338
- assembler, 106
- assignment statement, 457–458
 - conceptual and programming exercises for, 465–467
 - environment, 462–463
 - illustration of pass-by-value in Camille, 459–460
 - reference data type, 460–461
 - stack object, 463–465
 - use of nested `lets` to simulate sequential evaluation, 458–459
- associativity, 50
 - of operators, 57–58
- asynchronous callbacks, 620
- atom?, list-of-atoms?, and list-of-numbers?, 153–154
- atomic proposition, 642
- attribute grammar, 66
- automobile concepts, 7

B

- backtracking, 651
- Backus–Naur Form (BNF), 40–41
- backward chaining, 659–660
- balanced pairs of lexemes, 43
- bash script, 404
- β -reduction, 492–495
 - examples of, 495–499
- biconditional, 644
- binary search tree abstraction, 151–152
- binary tree abstraction, 150–151
- binary tree example, 667–672
- binding and scope
 - deep, shallow, and ad hoc binding, 233–234
 - ad hoc binding, 236–238
 - conceptual exercises for, 239–240
 - deep binding, 234–235
 - programming exercises for, 240
 - shallow binding, 235–236
- dynamic scoping, 200–202
 - vs. static scoping, 202–207
- free or bound variables, 196–198
 - programming exercises for, 198–199
- FUNARG problem, 213–214
 - addressing, 226–228
 - closures vs. scope, 224–225
 - conceptual exercises for, 228
 - downward, 214
 - programming exercises for, 228–233
 - upward, 215–224
 - upward and downward FUNARG problem in single function, 225–226
 - uses of closures, 225

- introduction, 186–187
 - lexical addressing, 193–194
 - conceptual exercises for, 194–195
 - programming exercise for, 195
 - mixing lexically and dynamically scoped variables, 207–211
 - conceptual and programming exercises for, 211–213
 - preliminaries
 - closure, 186
 - static vis-à-vis dynamic properties, 186
 - static scoping
 - conceptual exercises for, 192–193
 - lexical scoping, 187–192
 - vs. dynamic scoping, 202–207
 - bindings times, 6–7
 - block-structured language, 188
 - Blub Paradox, 21
 - BNF. *See* Backus–Naur Form (BNF)
 - bootstrapping a language, 540
 - bottom-up parser, 75, 80–81
 - bottom-up parsing, 48
 - bottom-up programming, 15–16, 177, 716–717
 - bottom-up style of programming, 540
 - bound variables, 196–198. *See also* formal parameters
 - programming exercises for, 198–199
 - breakpoints, 560–562
 - built-in functions, in Haskell, 301–307
- C**
- C language, 105–106
 - call chain, 200
 - call-with-current-continuation, 550–554
 - callbacks, 618–620
 - call/cc, defining, 622–625
 - call/cc vis-à-vis CPS, 617–618
 - Camille, 86–89
 - adding support for recursion in, 440–441
 - for user-defined functions to, 423–426
 - assignment statement in, 457–458
 - implementing
 - pass-by-name/need in, 522–526
 - programming exercises for, 526–527
 - pass-by-value in, 459–460
 - properties of new versions of, 465†
 - sequential execution in, 527–532
 - programming exercise for, 533
 - Camille abstract-syntax tree for, 359
 - data type: `TreeNode`, 359–360
 - parser generator with tree builder, 360–364
 - programming exercises for, 364–365
 - Camille interpreter, 533–537
 - conceptual and programming exercises for, 537–539
 - implementing
 - pass-by-reference in, 485
 - programming exercise for, 490–492
 - reimplementation of
 - `evaluate_operand` function, 487–490
 - revised implementation of references, 486–487
 - candidate sentences, 34
 - capability to impart control, 701
 - category theory, 385
 - choices of representation, 367
 - Chomsky hierarchy, 41
 - class constraint, 257
 - clausal form, 651–653
 - resolution with propositions in, 657–660
 - CLIPS programming language, 14, 705
 - asserting facts and rules, 705–706
 - conditional facts in rules, 708
 - programming exercises for, 708–709
 - templates, 707
 - variables, 706–707
 - Closure, 116
 - closed-world assumption, 701
 - closure, 186
 - non-recursive functions, 426–427
 - representation
 - in Python, 371–372
 - of recursive environment, 442
 - in Scheme, 367–371
 - uses of, 225
 - vs. scope, 224–225
 - CNF. *See* conjunctive normal form (CNF)
 - code indentation, 727
 - coercion, 249
 - combining function, 319
 - Common Lisp, 128
 - compilation, 106
 - low-level view of execution by, 110†
 - compile time, 6
 - compiler, 194
 - compiler translates, 104
 - advantages and disadvantages of, 115†
 - vs. interpreters, 114–115
 - complete function application, 286
 - complete graph, 680
 - complete recursive-descent parser, 76–79
 - compound propositions, 642
 - compound term, 645
 - concepts, relationship of, 714–715
 - concrete syntax, 356
 - representation, 74
 - `concrete2abstract` function, 358
 - conjunctive normal form (CNF), 646–648
 - cons cells, 135–136
 - conceptual exercise for, 141
 - list-box diagrams, 136–140
 - list representation, 136
 - consequent, definition of, 651
 - constant function, 130
 - constructor, 352
 - context-free languages and, 42–44
 - context-sensitive grammar, 64–67
 - conceptual exercises for, 67
 - continuation-passing style (CPS)
 - all-or-nothing proposition, 613–614
 - call/cc vis-à-vis CPS, 617–618
 - growing stack or growing continuation, 610–613
 - introduction, 608–610
 - trade-off between time and space complexity, 614–617
 - transformation, 620–622
 - conceptual exercises for, 625–626
 - defining call/cc in, 622–625
 - programming exercises for, 626–635
 - control abstraction, 585–586
 - applications of first-class continuations, 589
 - conceptual exercises for, 591–593
 - coroutines, 586–589

- power of first-class
 - continuations, 590
 - programming exercises for, 593–594
 - control and exception handling, 547
 - callbacks, 618–620
 - continuation-passing style
 - all-or-nothing proposition, 613–614
 - `call/cc` vis-à-vis CPS, 617–618
 - growing stack or growing continuation, 610–613
 - introduction, 608–610
 - trade-off between time and space complexity, 614–617
 - transformation, 620–635
 - control abstraction, 585–586
 - applications of first-class continuations, 589
 - conceptual exercises for, 591–593
 - coroutines, 586–589
 - power of first-class continuations, 590
 - programming exercises for, 593–594
 - first-class continuations
 - `call/cc`, 550–554
 - concept of continuation, 548–549
 - conceptual exercises for, 554–555
 - programming exercises for, 555–556
 - global transfer of control with breakpoints, 560–562
 - conceptual exercises for, 564–565
 - first-class continuations in Ruby, 562–563
 - nonlocal exits, 556–560
 - other mechanisms for, 570–579
 - programming exercises for, 565–570
 - levels of exception handling in programming languages, 579
 - dynamically scoped
 - exceptions, 582–583
 - first-class continuations, 583–584
 - function calls, 580–581
 - lexically scoped exceptions, 581
 - programming exercises for, 584–585
 - stack unwinding/crawling, 581–582
 - tail recursion
 - iterative control behavior, 596–598, 596f
 - programming exercises for, 606–608
 - recursive control behavior, 594–595
 - space complexity and lazy evaluation, 601–606
 - tail-call optimization, 598–600
 - coroutines, 586–589
 - CPS. *See* continuation-passing style (CPS)
 - curried form, 292–294
 - currying
 - all built-in functions in Haskell are, 301–307
 - conceptual exercises for, 310–311
 - curried form, 292–294
 - flexibility in, 297–301
 - form through first-class closures, 307–308
 - ML analogs, 308–310
 - partial function application, 285–292
 - programming exercises for, 311–313
 - and `uncurry` functions in Haskell, 295–297
 - and uncurrying, 294–295
- D**
- dangling else problem, 58–60
 - data abstraction, 337–338
 - abstract syntax, 356–359
 - programming exercises for, 364–365
 - abstract-syntax tree for Camille, 359
 - Camille abstract-syntax tree
 - data type: `TreeNode`, 359–360
 - Camille parser generator with tree builder, 360–364
 - programming exercises for, 364–365
 - aggregate data types
 - arrays, 338
 - discriminated unions, 343
 - programming exercises for, 343–344
 - records, 338–340
 - undiscriminated unions, 341–343
 - case study, 366–367
 - abstract-syntax representation
 - in Python, 372–373
 - choices of representation, 367
 - closure representation in Python, 371–372
 - closure representation in Scheme, 367–371
 - programming exercises for, 373–382
 - conception and use of data structure, 366
 - inductive data types, 344–347
 - ML and Haskell
 - analysis, 385
 - applications, 383–385
 - comparison of, 383
 - summaries, 382–383
 - variant records, 347–348
 - in Haskell, 348–352
 - programming exercises for, 354–356
 - in Scheme, 352–354
 - decision trees, 710
 - declaration position, 193
 - declarative programming, 13–15
 - deduction theorem, 644
 - deep binding, 234–235
 - deferred callback, 620
 - defined language vis-à-vis defining language, 395
 - defined programming language, 115, 395
 - `delay` function, 504
 - denotation, 186
 - denotational construct, 37, 39
 - denoted value, 345
 - `dereference` function, 461, 522
 - difference lists technique, 144–146
 - discriminated unions, 343
 - docstrings, 728
 - domain-specific language, 15
 - dot notation, 136
 - downward FUNARG problem, 214
 - in single function, 225–226
 - DrRacket* IDE, 352
 - Dyck language, 43
 - dynamic binding, 6–7
 - dynamic scoping, 200–202
 - advantages and disadvantages of, 203f
 - vs. static scoping, 202–207
 - dynamic semantics, 67
 - dynamic type system, 245
 - dynamically scoped exceptions, 582–583

E

eager evaluation, 493
 EBNF. *See* Extended Backus–Naur Form (EBNF)
 embedded specific language, 15
 empty string, 34
 entailment, 643
 environment, 366–382, 441–445, 462–463
 environment frame. *See* activation record
 Erlang, 178
 evaluate-expression function, 393
 evaluate_expr function, 427–430, 445–446
 evaluate_operand function, reimplementing of, 487–490
 execute_stmt function, 532
 expert system, 705
 explicit conversion, 252
 explicit/implicit typing, 268
 expressed values vis-à-vis denoted values, 394–395
 Extended Backus–Naur Form (EBNF), 45, 60–61
 conceptual exercises for, 61–64
 external representation, 356

F

fact, 656
 factorial function, 610
 fifth-generation languages. *See* logic programming;
 declarative programming
 finite-state automaton (FSA), 38–39, 73, 74f
 two-dimensional array modeling, 75t
 first Camille interpreter
 abstract-syntax trees for arguments lists, 401–403
 front end for, 396–399
 how to run Camille program, 404–405
 read-eval-print loop, 403–404
 simple interpreter for, 399–401
 first-class closures, supporting curried form through, 307–308
 first-class continuations
 applications of, 589
 call/cc, 550–554
 concept of continuation, 548–549
 conceptual exercises for, 554–555

 levels of exception handling in programming languages, 583–584
 power of, 590
 programming exercises for, 555–556
 in Ruby, 562–563
 first-class entity, 11, 126
 first-order predicate calculus, 14, 644–645
 conjunctive normal form, 646–648
 representing knowledge as predicates, 645–646
 fixed-format languages, 72
 fixed point, 505
 fixed-point Y combinator, 714
 folding function, 319
 folding lists, 319–324
 foldl vis-à-vis foldr, 323–324
 in Haskell, 319–320
 in ML, 320–323
 foldl, use of, 606
 foldl', use of, 606
 foldr, use of, 606
 formal grammar, 40
 formal languages, 34–35
 formal parameters, 131
 formalism gone awry, 660
 Fortran, 22
 forward chaining, 649, 657, 660
 fourth-generation languages, 81
 free-format languages, 72
 free or bound variables, 196–198
 programming exercises for, 198–199
 free variables, 196–198
 programming exercises for, 198–199
 fromRational function, 257
 front end, 73
 for Camille, 396–399
 source code, 394
 FSA. *See* finite-state automaton (FSA)
 full FUNARG programming language, 226
 FUNARG problem, 213–214
 addressing, 226–228
 closures vs. scope, 224–225
 conceptual exercises for, 228
 downward, 214
 in single function, 225–226
 programming exercises for, 228–233
 upward, 215–224
 in single function, 225–226
 uses of closures, 225

function annotations, 738
 function calls, 580–581
 function currying, 155
 function hiding. *See* function overriding
 function overloading, 738
 function overriding, 267–268
 functional composition, 315–316
 functional mapping, 313–315
 functional programming, 11–12
 advanced techniques
 eliminating expression recomputation, 167
 more list functions, 166–167
 programming exercises for, 170–174
 repassing constant arguments across recursive calls, 167–170
 binary search tree abstraction, 151–152
 binary tree abstraction, 150–151
 concurrency, 177–178
 cons cells, 135–136
 conceptual exercise for, 141
 list-box diagrams, 136–140
 list representation, 136
 functions on lists
 append and reverse, 141–144
 difference lists technique, 144–146
 list length function, 141
 programming exercises for, 146–149
 hallmarks of, 126
 lambda calculus, 126–127
 languages and software engineering, 174
 building blocks as abstractions, 174–175
 language flexibility supports program modification, 175
 malleable program design, 175
 prototype to product, 175–176
 layers of, 176–177
 Lisp
 introduction, 128
 lists in, 128–129
 lists in, 127–128
 local binding
 conceptual exercises for, 164
 let and let* expressions, 156–158
 letrec expression, 158
 programming exercises for, 164–165

- using `let` and `letrec` to
 - define, 158–161
- other languages supporting, 161–164
- programming project for, 178–179
- recursive-descent parsers,
 - Scheme predicates as, 153
 - `atom?`, `list-of-atoms?`, and `list-of-numbers?`, 153–154
 - `list-of` pattern, 154–156
 - programming exercise for, 156
- Scheme
 - conceptual exercise for, 134
 - homoiconicity, 133–134
 - interactive and illustrative session with, 129–133
 - programming exercises for, 134–135
- functions, 126
 - non-recursive functions
 - adding support for
 - user-defined functions to Camille, 423–426
 - augmenting
 - `evaluate_expr` function, 427–430
 - closures, 426–427
 - conceptual exercises for, 431–432
 - programming exercises for, 432–440
 - simple stack object, 430–431
- recursive functions
 - adding support for recursion in Camille, 440–441
 - augmenting
 - `evaluate_expr` with new variants, 445–446
 - conceptual exercises for, 446–447
 - programming exercises for, 447–450
 - recursive environment, 441–445
- functions on lists
 - `append` and `reverse`, 141–144
 - difference lists technique, 144–146
 - `list length` function, 141
 - programming exercises for, 146–149
- functor, 645

G

- generate-filter style of programming, 507
- generative construct, 41

- global transfer of control
 - with continuations
 - breakpoints, 560–562
 - conceptual exercises for, 564–565
 - first-class continuations in Ruby, 562–563
 - nonlocal exits, 556–560
 - programming exercises for, 565–570
 - other mechanisms for, 570
 - conceptual exercises for, 578
 - `goto` statement, 570–571
 - programming exercises for, 578–579
 - `set jmp` and `long jmp`, 571–578
- goal. *See* headless Horn clause
- `goto` statement, 570–571
- grammars, 40–41
 - conceptual exercises for, 61–64
 - context-free languages and, 42–44
 - disambiguation
 - associativity of operators, 57–58
 - classical dangling else problem, 58–60
 - operator precedence, 57
 - generate sentences from, 44–46
 - language recognition, 46f, 47–48
 - regular, 41–42
- growing continuation, 610–613
- growing stack, 610–613

H

- handle, 48
- hardware description languages, 17
- Haskell languages, 162, 258–259
 - all built-in functions in, 301–307
 - analysis, 385
 - applications, 383–385
 - comparison of, 383
 - `curry` and `uncurry` functions in, 295–297
 - folding lists in, 319
 - sections in, 316–319
 - summaries, 382–383
 - variant records in, 348–352
- headed Horn clause, 653, 656
- headless Horn clause, 653, 656, 665
- heterogeneous lists, 128
- higher-order functions (HOFs), 155, 716
 - analysis, 334–335

- conceptual exercises for, 329–330
- crafting cleverly conceived functions with curried, 324–328
- folding lists, 319–324
- functional composition, 315–316
- functional mapping, 313–315
- programming exercises for, 330–334
 - sections in Haskell, 316–319
- Hindley–Milner algorithm, 270
- HOFs. *See* higher-order functions (HOFs)
- homoiconic language, 133, 540
- homoiconicity, 133–134
- Horn clauses, 653–654
 - limited expressivity of, 702
 - in Prolog syntax, casting, 663
- host language, 115
- hybrid language
 - implementations, 109
- hybrid systems, 112
- hypothesis, 656

I

- imperative programming, 10
- implication function, 643
- implicit conversion, 248–252
- implicit currying, 301
- implicit typing, 268
- `implode` function, 325–326
- independent set, 680
- inductive data types, 344–347
- instance variables, 216
- instantiation, 651
- interactive or incremental testing, 146
- interactive top-level. *See* read-eval-print loop
- interface polymorphism, 267
- interpretation vis-à-vis compilation, 103–109
- interpreter, 103
 - advantages and disadvantages of, 115f
 - vs. compilers, 114–115
- introspection, 703
- iterative control behavior, 596–598, 596f

J

- JIT. *See* Just-in-Time (JIT)
 - implementations
- `join` functions, 728
- Just-in-Time (JIT)
 - implementations, 111

K

keyword arguments, 735–737
 Kleene closure operator, 34

L

LALR(1) parsers, 90
 lambda (λ) calculus, 11,
 126–127
 abstract syntax, 356–359
 scope rule for, 187–188
 lambda functions, 738–739
 Python primer, 738–739
 Language-INtegrated Queries
 (LINQ), 18
 languages
 defined, 4
 definition time, 6
 development, factors
 influencing, 21–25
 generator, 79–80
 implementation time, 6
 and software engineering, 174
 building blocks as
 abstractions, 174–175
 language flexibility supports
 program modification, 175
 malleable program design,
 175
 prototype to product,
 175–176
 themes revisited, 714
 late binding. *See* dynamic
 binding
 LATEX compiler, 106
 lazy evaluation, 160
 analysis of, 511–512
 applications of, 511
 β -reduction, 492–495
 C macros to demonstrate
 pass-by-name, 495–499
 conceptual exercises for,
 513–517
 enables list comprehensions,
 506–511
 implementing, 501–505
 introduction, 492
 programming exercises for,
 517–522
 purity and consistency, 512–513
 tail recursion, 601–606
 two implementations of,
 499–501
 learning language concepts,
 through interpreters, 393–394
 left-linear grammars, 41
 leftmost derivation, 45
 length function, 141

let expressions, 156–158
 let* expressions, 156–158
 letrec expression, 158
 lexemes, 40, 72
 lexical addressing, 193–194
 conceptual exercises for,
 194–195
 programming exercise for, 195
 lexical analysis, 72
 lexical closures, 716, 739–740
 lexical depth, 193
 lexical scoping, 187–192, 425
 and dynamically scoped
 variables, 207–211
 exceptions, 581
 linear grammar, 41
 link time, 6
 LINQ. *See* Language-INtegrated
 Queries (LINQ)
 Lisp, 11, 176
 introduction, 128
 lists in, 128–129
 list-and-symbol representation.
 See S-expression
 list-box diagrams, 136–140
 list-of pattern, 154–156
 list-of-vectors representation
 (LOVR), 379
 lists
 comprehensions, lazy
 evaluation, 506–511
 in functional programming, 127
 functions
 append and reverse,
 141–144
 difference lists technique,
 144–146
 list length function, 141
 programming exercises for,
 146–149
 in Lisp, 128–129
 and pattern matching in,
 672–674
 predicates in Prolog, 674–675
 Python primer, 731–733
 representation, 136
 literal function, 130
 literate programming, 23
 load time, 6
 local binding
 conceptual exercises for, 164
 and conditional evaluation
 Camille grammar and
 language, 395–396
 checkpoint, 391–393
 conditional evaluation in
 Camille, 410–411
 first Camille interpreter,
 396–405

 interpreter essentials, 394–395
 learning language concepts
 through interpreters,
 393–394
 programming exercises for,
 417–419
 putting it all together, 411–417
 syntactic and operational
 support for local binding,
 405–410
 let and let* expressions,
 156–158
 letrec expression, 158
 other languages supporting,
 161–164
 programming exercises for,
 164–165
 Python primer, 742
 using let and letrec to
 define, 158–161
 local block, 190
 local reference, 190
 logic programming
 analysis of Prolog
 metacircularProlog
 interpreter and WAM,
 704–705
 Prolog vis-à-vis predicate
 calculus, 701–703
 reflection in, 703–704
 applications of
 decision trees, 710
 natural language processing,
 709
 CLIPS programming language,
 705
 asserting facts and rules,
 705–706
 conditional facts in rules, 708
 programming exercises for,
 708–709
 templates, 707
 variables, 706–707
 first-order predicate calculus,
 644–645
 conjunctive normal form,
 646–648
 representing knowledge as
 predicates, 645–646
 imparting more control in,
 691–697
 conceptual exercises for,
 697–698
 programming exercises for,
 698–701
 introduction, 641–642
 from predicate calculus to
 clausal form, 651–653
 conversion examples, 654–656

formalism gone awry, 660
 Horn clauses, 653–654
 motif of, 656
 resolution with propositions
 in clausal form, 657–660
 Prolog programming language,
 660–662
 analogs between Prolog and
 RDBMS, 681–685
 arithmetic in, 677–678
 asserting facts and rules,
 662–663
 casting Horn clauses in
 Prolog syntax, 663
 conceptual exercises for,
 685–686
 graphs, 679–681
 list predicates in, 674–675
 lists and pattern matching in,
 672–674
 negation as failure in, 678–679
 primitive nature of `append`,
 675–676
 program control in, 667–672
 programming exercises for,
 686–691
 resolution, unification, and
 instantiation, 665–667
 running and interacting with,
 663–665
 tracing resolution process,
 676–677
 propositional calculus, 642–644
 resolution
 in predicate calculus, 649–651
 in propositional calculus,
 648–649
 logical equivalence, 644
 logician Haskell Curry, 292
 LOVR. *See* list-of-vectors
 representation (LOVR)

M

macros, 716
 operator, 176
 malleable program design, 175
 manifest typing, 132. *See also*
 implicit typing
 Match-Resolve-Act cycle, 705
 memoized lazy evaluation. *See*
 pass-by-need
 Mercury programming
 language, 14
 mergesort function, 744–748
 metacharacters, 36
 metacircular interpreters,
 539–540, 704–705
 programming exercise for,
 540–542

MetaLanguage (ML), 36, 162
 analogs, 308–310
 analysis, 385
 applications, 383–385
 comparison of, 383
 summaries, 382–383
 metaphor, 24
 metaprogramming, 716
 ML. *See* MetaLanguage (ML)
 modus ponens, 648
 monomorphic, 253
 multi-line comments, Python,
 727
 mutual recursion, Python
 primer, 744

N

named keyword arguments,
 735–737
 natural language processing,
 709
 nested functions, Python primer,
 743
 nested `lets`, to simulate
 sequential evaluation,
 458–459
 non-recursive functions
 adding support for
 user-defined functions to
 Camille, 423–426
 augmenting `evaluate_expr`
 function, 427–430
 closures, 426–427
 conceptual exercises for,
 431–432
 programming exercises for,
 432–440
 simple stack object, 430–431
 non-terminal alphabet, 40
 nonfunctional requirements,
 19
 nonlocal exits, 553, 556–560
 normal-order evaluation, 493
`ntExpressionList` variant,
 402

O

object-oriented programming in,
 12–13, 748–750
 occurs-bound?, 197–198
 occurs-free?, 197–198
 operational semantics, 19
 operator precedence, 57
 operator/function overloading,
 263–267
 overloading, 258

P

palindromes, 34
`papply` function, 288
 parameter passing
 assignment statement, 457–458
 conceptual and programming
 exercises for, 465–467
 environment, 462–463
 illustration of pass-by-value
 in Camille, 459–460
 reference data type, 460–461
 stack object, 463–465
 use of nested `lets` to
 simulate sequential
 evaluation, 458–459
 Camille interpreters, 533–537
 conceptual and programming
 exercises for, 537–539
 implementing
 pass-by-name/need in
 Camille, 522–526
 programming exercises for,
 526–527
 implementing
 pass-by-reference in Camille
 interpreter, 485
 programming exercise for,
 490–492
 reimplementation of
 `evaluate_operand`
 function, 487–490
 revised implementation of
 references, 486–487
 lazy evaluation
 analysis of, 511–512
 applications of, 511
 β -reduction, 492–495
 C macros to demonstrate
 pass-by-name, 495–499
 conceptual exercises for,
 513–517
 enables list comprehensions,
 506–511
 implementing, 501–505
 introduction, 492
 programming exercises for,
 517–522
 purity and consistency,
 512–513
 two implementations of,
 499–501
 metacircular interpreters,
 539–540
 programming exercise for,
 540–542
 sequential execution in
 Camille, 527–532
 programming exercise for, 533
 survey of

- conceptual exercises for, 482–484
- pass-by-reference, 472–477
- pass-by-result, 477–478
- pass-by-value, 467–472
- pass-by-value-result, 478–480
- programming exercises for, 484–485
- summary, 481–482
- parametric polymorphism, 253–262
- parse trees, 51–56
- parser, 258
- parser generator, 81
- parsing, 46*f*, 47–48, 74–76
 - bottom-up, shift-reduce, 80–82
 - complete example in *lex* and *yacc*, 82–84
 - conceptual exercises for, 90
 - infuse semantics into, 50
 - programming exercises for, 90–100
- Python *lex-yacc*, 84
 - Camille scanner and parser generators in, 86–89
 - complete example in, 84–86
- recursive-descent, 76
 - complete recursive-descent parser, 76–79
 - language generator, 79–80
 - top-down vis-à-vis bottom-up, 89–90
- partial argument application. *See* partial function application
- partial function application, 285–292
- partial function instantiation. *See* partial function application
- pass-by-copy, 144. *See also* pass-by-value
- pass-by-name, 499–500
 - C macros to demonstrate, 495–499
 - implementing in Camille, 522–526
 - programming exercises for, 526–527
- pass-by-need, 499–500
 - implementing in Camille, 522–526
 - programming exercises for, 526–527
- pass-by-reference, 472–477
- pass-by-result, 477–478
- pass-by-sharing, 471
- pass-by-value, 459–460, 467–472
- pass-by-value-result, 478–480
- pattern-directed invocation, 17
- Perl, 207
 - program demonstrating dynamic scoping, 208
 - whose run-time call chain depends on its input, 210
- + operator, 36
- polymorphic, 144, 253
- polysemes, 55, 56*t*
- positional vis-à-vis keyword arguments, 735–738
- pow function, 131
- powerset function, 327–328
- powucf function, 293
- precedence, 50
- predicate calculus
 - to logic programming clausal form, 651–653
 - conversion examples, 654–656
 - formalism gone awry, 660
 - Horn clauses, 653–654
 - motif of, 656
 - resolution with propositions in clausal form, 657–660
- representing knowledge as, 645–646
- resolution in, 649–651
- vis-à-vis predicate calculus, 701–703
- primitive *car*, 142
- primitive *cdr*, 142
- primitive *cons*, 142
- problem solving, thought process for, 20–21
- procedure, 126
- program-compile-debug-recompile loop, 175
- program, definition of, 4
- programming language bindings, 6–7
- concept, 4–5, 7–8
- concepts, 7–8
- definition of, 4
- features of type systems used in, 248*t*
- fundamental questions, 4–6
- implementation
 - influence of language goals on, 116
- interpretation vis-à-vis compilation, 103–109
- interpreters and compilers, comparison of, 114–115
- programming exercises for, 117–121
- run-time systems, 109–114
- levels of exception handling in, 579
- dynamically scoped exceptions, 582–583
- first-class continuations, 583–584
- function calls, 580–581
- lexically scoped exceptions, 581
- programming exercises for, 584–585
- stack unwinding/crawling, 581–582
- recurring themes in, 25–26
- scope rules of, 187
- programming styles
 - bottom-up programming, 15–16
 - functional programming, 11–12
 - imperative programming, 8–10
 - language evaluation criteria, 19–20
 - logic/declarative programming, 13–15
 - object-oriented programming, 12–13
 - synthesis, 16–19
 - thought process for problem solving, 20–21
- Prolog programming language, 14, 660–662
 - analysis of metacircularProlog interpreter and WAM, 704–705
 - Prolog vis-à-vis predicate calculus, 701–703
 - reflection in, 703–704
 - arithmetic in, 677–678
 - asserting facts and rules, 662–663
 - casting Horn clauses in Prolog syntax, 663
 - conceptual exercises for, 685–686
 - graphs, 679–681
 - imparting more control in, 691–697
 - conceptual exercises for, 697–698
 - programming exercises for, 698–701
 - list predicates in, 674–675
 - lists and pattern matching in, 672–674
 - negation as failure in, 678–679
 - primitive nature of *append*, 675–676
 - program control in, 667–672
 - programming exercises for, 686–691
 - and RDBMS, analogs between, 681–685

resolution, unification, and
 instantiation, 665–667
 running and interacting with,
 663–665
 tracing resolution process,
 676–677
 promise, 501
 proof by refutation, 649
 propositional calculus, 642–644
 resolution in, 648–649
 pure interpretation, 112
 purity, concept of, 12
 pushdown automata, 43
 Python, 19
 abstract-syntax representation
 in, 372–373
 closure data type in, 426
 closure representation in,
 371–372
 FUNARG problem, 222
 lex-yacc, 84
 Camille scanner and parser
 generators in, 86–89
 complete example in, 84–86
 Python primer
 data types, 722–725
 essential operators and
 expressions, 725–731
 exception handling, 750–751
 introduction, 722
 lists, 731–733
 object-oriented programming
 in, 748–750
 overview, 721–722
 programming exercises for,
 751–754
 tuples, 733–734
 user-defined functions
 lambda functions, 738–739
 lexical closures, 739–740
 local binding and nested
 functions, 742–743
 mergesort, 744–748
 more user-defined functions,
 740–742
 mutual recursion, 744
 positional vis-à-vis keyword
 arguments, 735–738
 simple user-defined
 functions, 734–735

Q

qualified type or constrained
 type, 257

R

Racket programming language,
 128

RDBMS. *See* relational database
 management system
 (RDBMS)
 read-eval-print loop (REPL), 130,
 175, 394, 403–404
 read-only reflection, 703
 records, 338–340
 recurring themes in study of
 languages, 25–27
 recursive-control behavior, 143,
 594–595
 recursive-descent parsers
 Scheme predicates as, 153
 atom?, list-of-atoms?,
 and list-of-numbers?,
 153–154
 list-of pattern, 154–156
 programming exercise for, 156
 recursive-descent parsing, 48, 76
 complete recursive-descent
 parser, 76–79
 language generator, 79–80
 recursive environment
 abstract-syntax representation
 of, 443–444
 list-of-lists representation of,
 444–445
 recursive functions
 adding support for recursion in
 Camille, 440–441
 augmenting evaluate_expr
 with new variants, 445–446
 conceptual exercises for,
 446–447
 programming exercises for,
 447–450
 recursive environment, 441–445
 reduce-reduce conflict, 51
 reducing, 48
 reference data type, 460–461
 referencing environment, 130,
 366
 referential transparency, 10
 regular expressions, 35–38
 conceptual exercises for, 39–40
 regular grammars, 41–42
 regular language, 39
 conceptual exercises for, 39–40
 relational database management
 system (RDBMS), analogs
 between Prolog and, 681–685
 REPL. *See* read-eval-print loop
 (REPL)
 representation
 abstract-syntax representation
 in Python, 372–373
 choices of, 367
 closure representation in
 Python, 371–372

 closure representation in
 Scheme, 367–371
 resolution, 383
 in predicate calculus,
 649–651
 proof by contradiction, 659
 in propositional calculus,
 648–649
 resumable exceptions, 583
 resumable semantics, 583
 Rete Algorithm, 705
 revised implementation of
 references, 486–487
 ribcage representation, 377
 right-linear grammar, 41
 rightmost derivation, 46
 Ruby
 first-class continuations in,
 562–563
 Scheme implementation of
 coroutines, 588–589
 rule of detachment, 648
 run-time complexity, 141–144
 run-time systems, 109–114

S

S-expression, 129, 356–357
 same-fringe problem, 511
 Sapir–Whorf hypothesis, 5
 scanning, 72–74
 conceptual exercises for, 90
 programming exercises for,
 90–100
 Scheme programming language,
 540
 closure representation in,
 367–371
 conceptual exercise for, 134
 homoiconicity, 133–134
 interactive and illustrative
 session with, 129–133
 programming exercises for,
 134–135
 variable-length argument lists
 in, 274–278
 variant records in, 352–354
 Schönfinkel, Moses, 292
 scope, closure vs., 224–225
 scripting languages, 17
 self-interpreter, 539
 semantics, 64–67
 conceptual exercises for, 67
 consequence, 643
 in syntax, modeling some,
 49–51
 sentence derivations, 44–46
 sentence validity, 34
 sentential form, 44

sequential execution, in Camille,
527–532
 programming exercise for, 533
set-builder notation, 507
set-former, 507
set `jmp` and `longjmp`, 571–578
shallow binding, 235–236
shift-reduce conflict, 51
shift-reduce parsers, 81
shift-reduce parsing, 48
short-circuit evaluation, 492
side effect, 7
Sieve of Eratosthenes algorithm,
507
simple interpreter for Camille,
399–401
simple stack object, 430–431
simple user-defined functions,
 Python primer, 734–735
simulated-pass-by-reference, 475
single-line comments, Python,
727
single list argument, 276
SLLGEN, 354
Smalltalk programming
 language, 12–13, 225
`sortedElem` function, 507
space complexity
 tail recursion, 601–606
 trade-off between time and,
 614–617
split functions, 728
SQL query, 14
square function, 499
stack frame. *See* activation record
stack object, 463–465. *See also*
 simple stack object
stack of interpreted software
 interpreters, 112
stack unwinding/crawling,
581–582
static bindings, 6, 116
static call graph, 200
static scoping
 advantages and disadvantages
 of, 203*t*
 conceptual exercises for,
 192–193
 lexical scoping, 187–192
 vs. dynamic scoping, 202–207
static semantics, 67
static type system, 245
static vis-à-vis dynamic
 properties, 186, 188*t*
static/dynamic typing, 268
string, 34
`string2int` function,
326–327
`struct`. *See* records

SWI-Prolog, 663
symbol table, 194
symbolic logic, 642
syntactic ambiguity, 48–49
 conceptual exercises for, 61–64
 modeling some semantics in,
 49–51
 parse trees, 51–56
syntactic analysis. *See* parsing
syntactic sugar, 45
syntax, 34

T

table-driven, top-down parser,
75
tail-call optimization, 598–600
tail recursion
 iterative control behavior,
 596–598, 596*f*
 programming exercises for,
 606–608
 recursive control behavior,
 594–595
 space complexity and lazy
 evaluation, 601–606
tail-call optimization,
598–600
tautology, 644
terminals, 40
terminating semantics, 582
terms, definition of, 651
throwaway prototype, 22, 176
thunk, 501–505
time and space complexity,
 trade-off between, 614–617
top-down parser, 75
top-down parsing, 48
top-down vis-à-vis bottom-up
 parsing, 89–90
traditional compilation, 112
`TreeNode`, 359–360
tuples, 275, 338
 Python primer, 733–734
Turing-complete. *See*
 programming language
Turing machine, 5
type cast, 252
type checking, 246–248
type class, 257
type inference, 268–274
type signatures, 310
type systems
 conceptual exercises for,
 278–280
 conversion, coercion, and
 casting
 conversion functions, 252–253
 explicit conversion, 252
 implicit conversion, 248–252

function overriding, 267–268
inference, 268–274
introduction, 245–246
operator/function overloading,
263–267
parametric polymorphism,
253–262
static/dynamic typing vis-à-vis
 explicit/implicit typing, 268
type checking, 246–248
variable-length argument lists
 in Scheme, 274–278

U

undiscriminated unions, 341–343
unification, 651
UNIX shell scripts, 116
unnamed keyword arguments,
735–737
upward FUNARG problem,
215–224
 in single function, 225–226
user-defined functions
 Python primer
 lambda functions, 738–739
 lexical closures, 739–740
 local binding and nested
 functions, 742–743
 mergesort, 744–748
 more user-defined functions,
 740–742
 mutual recursion, 744
 positional vis-à-vis keyword
 arguments, 735–738
 simple user-defined
 functions, 734–735

V

variable assignment, 458
variable-length argument lists, in
 Scheme, 274–278
variadic function, 275
variant records, 347–348
 in Haskell, 348–352
 programming exercises for,
 354–356
 in Scheme, 352–354
very-high-level languages. *See*
 logic programming,
 declarative programming
virtual machine, 109
von Neumann architecture, 7

W

WAM. *See* Warren Abstract
 Machine (WAM)

Warren Abstract Machine
 (WAM), 705
weakly typed languages,
 247
web browsers, 106

web frameworks, 17
well-formed formulas (wffs),
 646
wffs. *See* well-formed formulas
 (wffs)

Y
Y combinatory, 159
yacc parser generator, 81
 shift-reduce, bottom-up parser,
 82–84

Colophon

This book was typeset with L^AT_EX 2_ε and B_IB_TE_X using a 10-point Palatino font. Figures were produced using *Xfig* (X11 diagramming tool) and *Graphviz* with the DOT language.