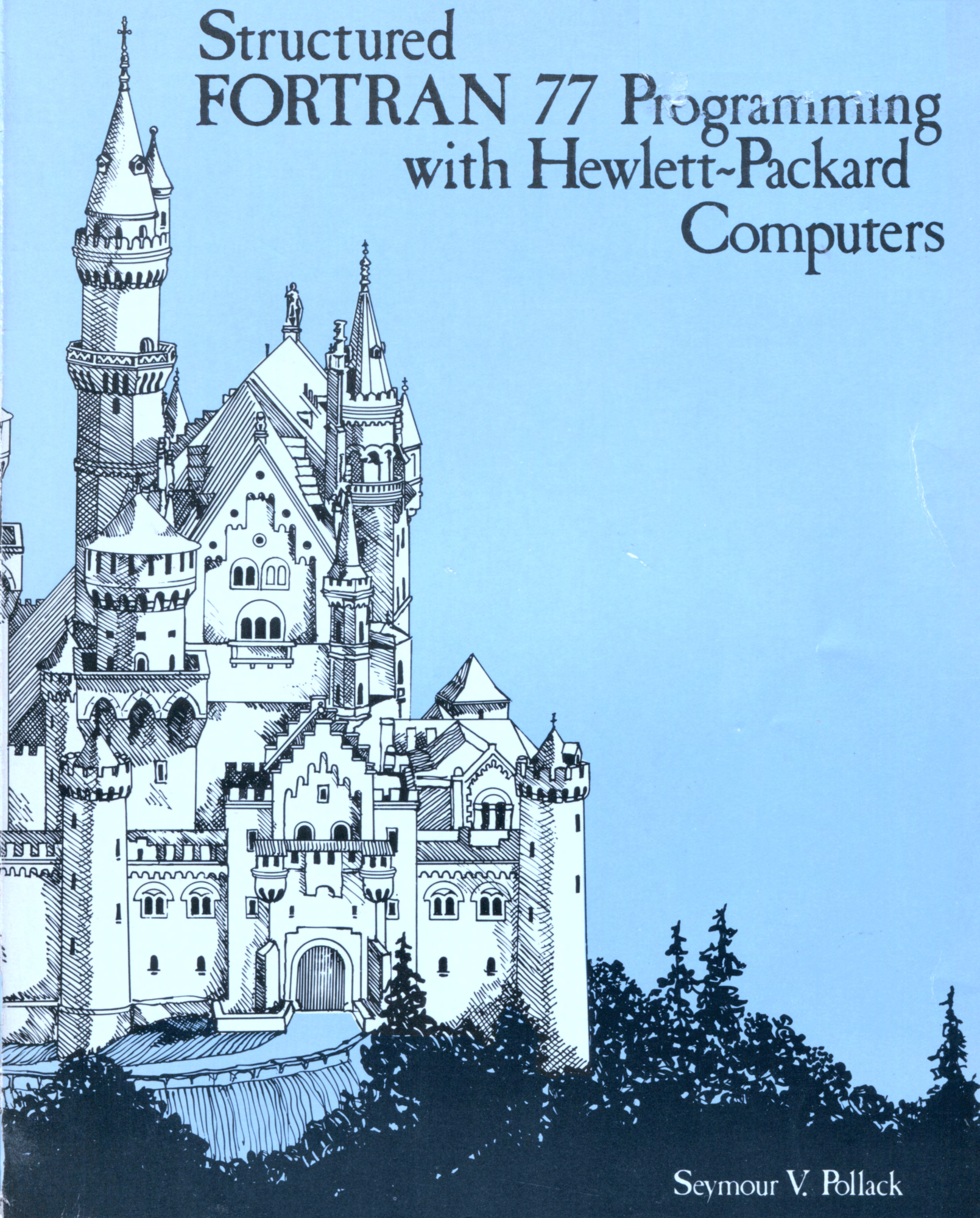
# Structured FORTRAN 77 Programming with Hewlett-Packard Computers



Seymour V. Pollack

Structured
FORTRAN 77 Programming
with Hewlett-Packard
Computers



Boyd & Fraser

bf

# Structured FORTRAN 77 Programming

with Hewlett-Packard Computers

**Other titles in the Boyd & Fraser Computer Science Series**

STANDARD BASIC PROGRAMMING: For Business and Management Applications
BEGINNING STRUCTURED COBOL
PASCAL PROGRAMMING: A Spiral Approach
STRUCTURED FORTRAN 77 PROGRAMMING
A STRUCTURED APPROACH TO GENERAL BASIC
A STRUCTURED APPROACH TO ESSENTIAL BASIC
INTRODUCTION TO COMPUTERS AND COMPUTER SCIENCE, Third Edition
AN INTRODUCTION TO ARTIFICIAL INTELLIGENCE: Can Computers Think?

# Structured FORTRAN 77 Programming

## with Hewlett-Packard Computers

**Seymour V. Pollack**

Department of Computer Science
School of Engineering and Applied Science
Washington University in St. Louis

Boyd & Fraser Publishing Company
SAN FRANCISCO

Credits:

Editor: Tom Walker
Production supervision: Dixie Clark
Design: Neil W. Kelley
Typesetting: Neil W. Kelley Graphic Services

© 1983 by Boyd & Fraser Publishing company. All rights reserved. No part of this work may be reproduced or used in any form or by any means—graphic, electronic, or mechanical including photocopying, recording, taping, or information and retrieval systems—without written permission from the publisher.

**Library of Congress Cataloging in Publication Data**

Pollack, Seymour V.
Structured Fortran 77 programming for Hewlett Packard computers.

Includes index.
1. Hewlett-Packard computers–Programming.
2. FORTRAN (Computer program language) I. Title.
QA76.8.H48P64 1983 001.64'2 87-72898
ISBN 0-87835-130-2

# *Contents*


**1 INTRODUCTORY CONCEPTS** 1

**1.1 Major Components of an HP Computer System** 1
1.1.1 The Central Processing Unit (CPU) 1
1.1.2 Main Storage 3
1.1.3 Input/Output Components 4
1.1.3.1 Keyboard Devices 4
1.1.3.2 Printing Devices 4
1.1.3.3 Secondary Storage 6

**1.2 Programs and Programming Languages** 6
1.2.1 Properties of a High Level Language 6
1.2.2 The High Level Language Compiler 8

**1.3 The HP Programming Environment** 10
1.3.1 FORTRAN 77 and the HP Operating System 10
1.3.2 Compiling and Running an HP FORTRAN 77 Program 12

**1.4 Glossary** 12

**Problems** 13

**2 THE STRUCTURED PROGRAMMING PROCESS** 15

**2.1 Specification of the Problem** 16

**2.2 Problem Solution and Algorithms** 16
2.2.1 What is an Algorithm? 16
2.2.2 Properties of Algorithms 17
Example 2.1 17
Example 2.2 18

**2.3 Algorithms Into Programs** 19
2.3.1 The Structured Program 19
2.3.1.1 The Operational Sequence 20
2.3.1.2 Simple Selection (Alternation) 22
2.3.1.3 Repetitive Action 23
Example 2.3 23
Example 2.4 28
2.3.1.4 Another Complete FORTRAN 77 Program 2.5 28
2.3.2 Additional Structured Elements 30

**2.4 Summary** 30

**Problems** 31

3 GETTING ACQUANTED WITH FORTRAN 77 36

3.1 **Overall Organization of a Program** 36
3.1.1 The Basic Form of a Program 36
3.1.2 Major Statement Types 36
3.1.2.1 PROGRAM and END statements 37
3.1.2.2 Declaration Statements 38
3.1.2.3 Assignent Statements 39
Example 3.1 39
Example 3.2 39
3.1.2.4 Input/Output Statements 40
Example 3.3 40
3.1.2.5 Control Statements 42
Example 3.3A 42
Example 3.4 44

3.2 **Preparation of FORTRAN Statements** 47
3.2.1 Format of a Statement 47
3.2.1.1 Statements Labels 50
3.2.1.2 Sequence Numbers 50
3.2.1.3 The Body of a Statement 50
3.2.1.4 Continuation of Statements on Additional Lines 51
3.2.1.5 Comments in Programs 51
3.2.1.6 A Brief Treatise on Blanks 51
3.2.2 The FORTRAN Character Set 51
3.2.3 Directives to the Compiler 52
3.2.3.1 Use of Directives 52
3.2.3.2 The $LIST Directives 52
3.2.3.3 The $PAGE Directive 52
3.2.3.4 The $TITLE Directive 52

3.3 **Summary** 53

**Problems** 54

4 DATA 62

4.1 **Types of Constants and Their Representation** 62
4.1.1 Numerical Constants 62
4.1.1.1 Integer Constants 62
4.1.1.2 Octal Constants 62
4.4.4.3 Hexadecimal Constants 63
4.1.1.4 Real Constants 63
4.1.1.4.1 The basic form for real constants 63
4.1.1.4.2 Scientific forms for real constants 63
4.1.1.4.3 Floating point numbers 65
4.1.1.4.4 Double precision numerical constants 65
4.1.1.5 Complex Numbers 66
4.1.2 Character Constants 66
4.1.3 Hollerith Constants 66
4.1.4 Logical Constants 66
4.1.5 Names for Constants—The PARAMETER Statement 67

4.2 **Variables and their Declaration** 67
4.2.1 Variable Names 67
4.2.2 Declaration of Variables 68
4.2.2.1 The INTEGER Declaration 68

4.2.2.2 The REAL Declaration 69
4.2.2.3 The DOUBLE PRECISION Declaration 69
4.2.2.4 The CHARACTER Declaration 70
4.2.2.5 The LOGICAL Declaration 70
4.2.2.6 A Note for Old Times' Sake 70
4.2.3 Alternate Names for Variables—the EQUIVALENCE Statement 71
**4.3 Initialization of Variables** 72
**4.4 Summary** 73
**Problems** 74

**5 COMPUTATIONS** 79
**5.1 The Assignment Statement** 79
5.1.1 Construction of an Expression 79
5.1.1.1 Operators and Operands 80
5.1.1.2 Types of Arithmetic Operators 81
5.1.1.3 Simple and Complicated Arithmetic Operands 81
5.1.1.4 Parentheses in Arithmetic Expressions 82
Example 5.1 82
5.1.2 Representation of Computed Results 85
5.1.2.1 Real Expressions 85
5.1.2.2 Integer Expressions 86
5.1.3 Rules for Conversion in Assignment Operations 87
**5.2 How Computations are Performed** 88
5.2.1 Arithmetic Involving Different Types of Data 88
5.2.1.1 Conversion Rules for +, −, *, / 88
5.2.1.2 Conversion Rules for Exponentiation 89
5.2.2 Priorities in Arithmetic Expressions 90
5.2.2.1 Priorities for +, −, *, / 90
5.2.2.2 Expressions with Exponentiation 90
5.2.2.3 Changing Priorities with Parentheses 91
**5.3 Computational Examples** 92
Example 5.2 92
Example 5.3 92
Example 5.4 93
**5.4 Summary** 97
**Problems** 98

**6 COMPUTATIONS WITH BUILT-IN FUNCTIONS** 105
**6.1 Functions for Data Conversion** 106
6.1.1 Conversion to Integer: The INT Function 106
6.1.2 Conversion to Real Numbers: The REAL Function 107
6.1.3 Conversion to Double Precision: The DBL Function 107
6.1.4 Conversion of Complex Numbers 107
**6.2 Basic Numerical Manipulations** 107
6.2.1 Operations with a Number's Sign 107
6.2.1.1 The ABS Function 107
6.2.1.2 The Sign Function 108
6.2.2 Positive Differences (DIM) 108
6.2.3 Double Precision Multiplication—the DPROD Function 108
6.2.4 Obtaining a Remainder—the MOD Function 108
Example 6.1 109

6.2.5 Rounding and Truncation 111
6.2.5.1 Truncation—the AINT Function 111
6.2.5.2 The Nearest Integer—the NINT Function 111
6.2.5.3 The ANINT Function 112
6.2.5.4 Rounding Techniques 112
6.2.6 Functions for Extreme Values 113
6.2.6.1 The MAX Function 113
6.2.6.2 The MIN Function 113
Example 6.2 113

**6.3 Computational Functions** 115
6.3.1 Algebraic Functions 117
6.3.2 Trigonometric Functions 117
6.3.3 Random Number Generation 117
6.3.3.1 The URAN Function 118
6.3.3.2 The GRAN Function 119
6.3.3.3 The IRAND Function 119
6.3.4 Nested Invocation of Built-In Functions 119
6.3.5 Generic Functions and the Good Old Days 120

**6.4 Logical Arithmetic with Integer Data** 121
6.4.1 Internal Representation of Integer Values for the HP1000 121
6.4.2 Logical Operations on Integer Values 122
6.4.2.1 The .NOT. Operation 123
6.4.2.2 The .AND. Operation 123
6.4.2.3 The .OR. Operation 123
6.4.2.4 The .EQV. Operation (equivalence) 124
6.4.3 Bit-Handling with Logical Operations 124
6.4.3.1 Bit-Setting Operations 124
6.4.3.2 Bit-Testing Operations 125

**6.5 Functions for Bit Manipulation** 125
6.5.1 Basic Logical Functions 126
6.5.2 Bit-Setting Functions 126
6.5.3 Bit-Manipulating Subprograms 127
6.5.3.1 The IBITS Function 127
6.5.3.2 The MVBITS Subroutine 127
6.5.3.3 Bit-Shifting Functions 128
6.5.4 Bit-Testing with the BTEST Function 129

**6.6 Summary** 129

**Problems** 129

**7 ARRAYS** 135

**7.1 Organization of Arrays** 135
7.1.1 Dimensionality of Arrays 135
7.1.2 Representation of Arrays in Storage 137
7.1.3 A General Formula for Locating Array Elements 140
Example 7.1(a) 140
Example 7.1(b) 141

**7.2 Specification of Numerical Arrays** 141
7.2.1 Declaration of Array Bounds 141
7.2.2 Initialization of Arrays 142
7.2.2.1 Using Repetitions Factors to Initialize Arrays 143
7.2.2.2 Initializing Parts of Arrays to Different Values 143

7.2.3 Specification of Subscripts 143
7.2.4 The Dimension Statement 144
**7.3 Processing of Arrays** 145
7.3.1 Array Elements in Expressions 145
Example 7.2 145
7.3.2 Input/Output of Arrays 145
**7.4 Summary** 153
**Problems** 153

**8 CHARACTER DATA** 160
**8.1 Specification of Character Strings** 160
8.1.1 Declaration of Character Strings 160
8.1.2 Defining Substrings 160
8.1.3 EQUIVALENCE and Character Strings 161
8.1.4 Character Strings and Numerical Data 164
8.1.5 Character Arrays 164
**8.2 Processing of Character Strings** 165
8.2.1 Basic Character String Assignments 165
8.2.1.1 Assignments with Different String Lengths 165
8.2.1.2 Assignments with Substrings 166
8.2.2 The Concatenation Operation 167
**8.3 Decision Processes and Character Strings** 168
8.3.1 Comparisons Between Character Strings 168
Example 8.1 170
8.3.2 Searching for Character Strings—The INDEX Function 173
Example 8.2 177
8.3.3 Conversions Between Characters and Integer Values 181
**8.4 Another Example Program** 181
**8.5 Summary** 182
**Problems** 186

**9 CONTROL STRUCTURES AND DECISION MAKING** 192
**9.1 The IF-THEN-ELSE Construct** 192
9.1.1 Specification of Test Conditions 192
9.1.1.1 Combinations of Comparisons: the AND Operation 193
Example 9.1 193
9.1.1.2 A Choice of Conditions: the OR Operation 196
9.1.1.3 Combined Conditions 196
Example 9.2 196
9.1.1.4 Tests with Arbitrary Combinations 200
Example 9.3 200
9.1.2 Construction of Actions for THEN and ELSE 200
9.1.2.1 Empty IF-Blocks 201
9.1.2.2 Actions with Extended Decision Rules 202
9.1.2.3 Construction of Nested Decisions 205
Example 9.4 205
**9.2 The CASE Construct** 209
9.2.1 The ELSE IF Statement 210
Example 9.5 210
9.2.2 More Extensive CASE Constructs 214
Example 9.6

9.2.3 Another Way to Implement the CASE Construct 214
Example 9.7 216
**9.3 Summary** 216
**Problems** 217

**10 CYCLIC OPERATIONS** 231

**10.1 Loops as Structural Components** 231
10.1.1 The DO-WHILE Construct 231
10.1.1.1 Direct Implementation of the DO-WHILE Component 231
10.1.1.2 Implementation of DO-WHILE with the Logical IF Statement 232
10.1.1.3 DO-WHILE Implementation with the Block IF Statement 233
10.1.2 The DO-UNTIL Construction 234
Example 10.1 235
**10.2 Automatic Program Loops: the DO Statement** 238
Example 10.2 240
10.2.1 Useful Techniques with DO Loops 243
10.2.1.1 Use of the Index in Loop Calculations 244
Example 10.3 244
10.2.1.2 Use of Variables as Loop Controllers 245
Example 10.4 247
10.2.1.3 Mismatches Between the Limiting Value and the Increment 247
10.2.1.4 Use of an Index Outside Its Loop 249
Example 10.5 250
10.2.1.5 Negative Increments 250
Example 10.6 251
10.2.1.6 Extreme Conditions In Loop Controllers 251
10.2.1.7 Non-Integer Controller Values 253
10.2.2 Nested DO Loops 256
10.2.2.1 Processing of Nested Loops 256
10.2.2.2 Processing Of Multidimensional Arrays 258
10.2.2.3 Sorting 259
Example 10.7 260
**10.3 Summary** 264
**Problems** 265

**11 INTRODUCTION TO SUBPROGRAMS** 272

**11.1 Basic Properties of Subprograms** 272
11.1.1 How Subprograms Work 272
11.1.1.1 Subprograms and the Main Program 272
11.1.1.2 Synthesis of Programs 273
11.1.1.3 Sequence of Execution 274
11.1.2 Subprograms and Program Development 275
11.1.2.1 Separate Development of Subprograms 276
11.1.2.2 Expectant Main Programs 276
11.1.3 Types of FORTRAN Subprograms 277
11.1.3.1 The Function Subprogram 277
11.1.3.2 Statement Functions 278
11.1.3.3 The Subroutine Subprogram 278

**11.2 Construction of Subprograms** 279
11.2.1 Definitions of Functions 280
11.2.1.1 The Function Name 280
11.2.1.2 The Dummy Argument List 280
Example 11.1 281
11.2.2 Definition of Subroutines 283
Example 11.2 283
**11.3 Invocation of Subprograms** 283
11.3.1 Invocation of Functions 284
Example 11.3 284
11.3.2 Invocation of Subroutines 288
Example 11.4 288
11.3.3 Multiple Entries in a Subprogram 289
11.3.4 Multiple Returns from a Subprogram 293
11.3.5 Multiple Destinations for Returns From a Subroutine 294
**11.4 Summary** 294
**Problems** 295

**12 DATA FOR ARGUMENTS TO SUBPROGRAMS** 304

**12.1 Single Data Values as Arguments** 304
12.1.1 Constants as Arguments 304
12.1.2 Expressions as Arguments 305
12.1.3 Array Elements as Arguments 306
12.1.4 Character Arguments with Adjustable Lengths 307
Example 12.1 307
**12.2 Arrays as Arguments** 308
Example 12.2 308
12.2.1 Size Differences Between Actual and Dummy Array Arguments 310
12.2.2 Adjustable Array Sizes 310
Example 12.3 314
12.2.3 A Special Note About Character Arrays 316
12.2.4 Parts of Arrays as Arguments 317
12.2.4.1 A General Rule for Parts of Arrays as Arguments 318
12.2.4.2 Another Way to Use Parts of Arrays as Arguments 318
12.2.4.3 Parts of Multidimensional Arrays as Arguments 318
12.2.5 Dimensional Differences Between Array Arguments 320
**12.3 Subprograms as Arguments** 321
12.3.1 The EXTERNAL Statement 322
Example 12.4 322
12.3.2 The INTRINSIC Statement 325
**12.4 Arguments In Nested Invocations** 325
**12.5 Summary** 325
**Problems** 326

**13 INTRODUCTION TO INPUT/OUTPUT** 335

**13.1 Data and Files** 335
13.1.1 Organization of files 335
13.1.1.1 Records 336
13.1.1.2 Relations Among Records In a File 336

13.1.1.3 Data Representation on Records 337
13.1.1.4 External and Internal Files 341
13.1.2 Files and Units 341
13.1.3 Operations With Files 343
13.1.3.1 Creating a File 343
13.1.3.2 Deleting a File 343
13.1.3.3 Opening a File 344
13.1.3.4 Closing a File 344
13.1.3.5 Reading and Writing File Records 344
13.1.3.6 Writing an ENDFILE record 344
13.1.3.7 Backspacing a File 345
13.1.3.8 Rewinding a File 345
13.1.3.9 File Inquiry 345
**13.2 Introduction to FORTRAN 77's Input/Output Statements** 345
13.2.1 Data Transfer Statements 345
13.2.1.1 The READ Statement 346
13.2.1.2 The WRITE Statement 347
13.2.1.3 The PRINT Statement 348
13.2.2 File Positioning Statement 349
13.2.2.1 The BACKSPACE Statement 349
13.2.2.2 The ENDFILE Statement 349
13.2.2.3 The REWIND Statement 350
13.2.3 Auxiliary Input/Output Statement 350
13.2.3.1 The OPEN Statement 350
13.2.3.2 The CLOSE Statement 352
13.2.3.3 The INQUIRE Statement 352
**13.3 Example Programs** 355
Example 13.1 355
Example 13.2 356
**13.4 Summary** 362
**Problems** 369

**14 LIST-DIRECTED INPUT/OUTPUT** 374

**14.1 Preparation of List-Directed Data** 374
14.1.1 Basic Requirements for List-Directed Input 374
14.1.2 List-Directed Data and Records 375
14.1.3 Treatment of Blanks 376
14.1.4 Missing Data 376
14.1.5 Additional Possibilities 376
14.1.5.1 Repeated Input Values 377
14.1.5.2 The Slash in List-Directed Input 377
14.1.6 List-Directed Output Data 377
**14.2 List-Directed Input/Output Statements** 378
14.2.1 The List-Directed READ Statement 378
14.2.1.1 Basic forms 378
14.2.1.2 Input with Implied Loops 378
14.2.2 The List-Directed WRITE Statements 380
14.2.2.1 Construction of the List-Directed WRITE 380
14.2.2.2 Literals in An Output List 380
14.2.2.3 Expressions in Output Lists 380
14.2.3 The PRINT Statement 381

**14.3 Summary** 381
**Problems** 382

**15 EDIT-DIRECTED INPUT/OUTPUT** 387

**15.1 Properties of Edit-Directed Data** 387
15.1.1 Interpretation of Edit-Directed Input 387
15.1.2 Appearance of Edit-Directed Output 389
**15.2 Construction of Edit-Directed Input Formats** 391
15.2.1 The X-Specification and Input 391
15.2.2 Numerical Specifications 391
15.2.2.1 The I-Specifications 391
15.2.2.2 Integer Output in Octal Form 391
15.2.2.3 Integer Output in Hexadecimal Form 392
15.2.2.4 The E-Specification for Input 392
15.2.2.5 The D-Specification 392
15.2.2.6 Decimal Points in Real Input Data 392
15.2.2.7 Versatility Versus Confusion 393
15.2.3 Character Input—The A- and R-Specification 393
15.2.3.1 Appearance of Edit-Directed Character Input 394
15.2.3.2 Reading Parts of Character Strings 394
15.2.3.3 Substrings as Input Items 395
15.2.4 Multi-Record Input Formats: The/-Specifications 396
15.2.5 Disagreements Between Input Lists and Format Descriptions 397
**15.3 Construction of Edit-Directed Output Formats** 398
15.3.1 Blank Output Columns—The X-Specification 398
15.3.2 Carriage Control 398
15.3.2.1 Starting a New Line 398
15.3.2.2 Other Carriage Control Symbols 398
15.3.2.3 Overprinting 398
15.3.3.4 The /-Specification for Carriage Control 400
15.3.3 Formatting of Numerical Output 401
15.3.3.1 Integer Values 401
15.3.3.2 Real Output in Conventional Form 402
15.3.3.3 Real Output in Single Precision form 402
15.3.3.4 Real Output in Double Precision Form 403
15.3.4 Formatting of Character String Output 403
15.3.4.1 The A- and R-Specification for Output 403
15.3.4.2 Writing Parts of Character Strings 404
15.3.4.3 Literal Output Strings 405
**15.4 Additional Format Specification Techniques** 406
15.4.1 Repeated Specifications 406
15.4.2 Repeated Combinations of format Descriptions 406
15.4.2.1 Formation of Repeated Patterns 407
15.4.2.2 Mismatches with Repeated Patterns 407
15.4.3 Nested Format Patterns 408
15.4.4 Automatically Repeating Format Descriptions 411
**15.5 Examples** 411
Example 15.1 411
Example 15.2 413
**15.6 Summary** 427
**Problems** 428

16 ADDITIONAL FORMATTING FEATURES 434

16.1 Further Specification of Numerical Values 434
16.1.1 The G-Specification 434
16.1.2 Format Descriptions for Sign Control 436
16.1.2.1 Producing Visible + Signs: The S-Specification 436
16.1.2.2 Restoring the Sign Default: The S-Specification 436
16.1.2.3 Suppression of Visible + Signs: The SS-Specification 437
16.1.3 Scaling of Edit-directed Values 438
16.1.3.1 Scaling of Input Data 439
16.1.3.2 Scaling of Output Values 439
16.1.3.3 Scaling and the G-Specification 439
16.1.4 Treatment of Input Blanks 440
16.1.4.1 The BN-Specification 440
16.1.4.2 Blanks as Zeros—The BZ-Specification 440

16.2 Position Control in an Edit-directed Record 440
16.2.1 The T-Specification 440
16.2.2 The TR-Specification 441
16.2.3 The TL-Specification 442

16.3 Run-time Format Descriptions 442
Example 16.1 443

16.4 Internal Files and Variable Formatting 448
16.4.1 Internal Files 449
16.4.2 Data Transmission with Internal Files 450
Example 16.2 451
16.4.3 Another Technique for Using Variable Formats 452

16.5 Summary 456
Problems 457

17 ADDITIONAL INPUT/OUTPUT TECHNIQUES 460

17.1 Unformatted Files and Records 460
17.1.1 Preparation of Unformatted Files 460
17.1.2 Transmission of Unformatted Data Records 461
Example 17.1 463

17.2 Direct Files 469
17.2.1 Construction of Direct Files 471
17.2.2 Opening a Direct File 472
17.2.3 Direct Input/Output 472
Example 17.2 473
Example 17.3 473

17.3 Summary 476
Problems 476

18 SHARED DATA AMONG PROGRAM COMPONENTS 481

18.1 Characteristics of Common Storage 481
18.1.1 Unnamed Common Storage 483
18.1.1.1 Arrangement of Space in Common Storage 483
18.1.1.2 Arrays in Blank Common Storage 483
Example 18.1 484
18.1.1.3 Variable Names in a Common Block 485
18.1.1.4 Different Declarations for a Common Block 487

18.1.2 Named Common Blocks 488
Example 18.2 489
**18.2 Initialization of Values in Common Blocks** 490
**18.3 Use and Misuse of Common Storage** 494
**18.4 Summary** 495
**Problems** 496

**19 LOGICAL VARIABLES** 499

**19.1 Logical Operations and Expressions** 499
19.1.1 Basic Logical Operations 499
19.1.1.1 The OR Operation 500
19.1.1.2 The AND Operation 501
19.1.1.3 The NOT Operation 501
19.1.1.4 The EQV Operation 501
19.1.1.5 The NEQV Operation 502
19.1.2 Priorities in Logical Expressions 502
19.1.3 Relational Operators and Logical Expressions 504
Example 19.1 505
19.1.4 Logical Operations on Numerical Variables 505
**19.2 Input/Output of Logical Data** 507
19.2.1 Logical Data Input 507
19.2.1.2 Edit-directed Logical Input 507
19.2.2 Output of Logical Data 507
19.2.2.1 List-directed Output of Logical Values 508
19.2.2.2 Edit-directed Output of Logical Values 508
**19.3 Summary** 508
**Problems** 509

**APPENDIX A—COMPLEX DATA** 511

**A.1 Declaration of Complex Variables** 511
**A.2 Computations with Complex Numbers** 511
**A.3 Built-in Functions and Complex Data** 512
**A.4 Input/Output of Complex Data** 513

**APPENDIX B—CHARACTER CODING SYSTEMS** 514

**B.1 The HP Character Set** 514
**B.2 The FORTRAN Character Set** 514
**B.3 Special Functions for the Collating Sequence** 514

**APPENDIX C—ADDITIONAL FORTRAN FEATURES** 517

**C.1 Declarations—The IMPLICIT Statement** 517
**C.2 Control Statements** 517
**C.2.1 The Assigned GO TO Statement** 517
**C.2.2 The Arithmetic IF Statement** 518
**C.2.2 The PAUSE Statement** 518
**C.3 Alternate Return Addresses for a Subroutine** 519

**INDEX** 521

# *Preface*

Since its appearance in 1956, the FORTRAN programming language has been so widely adopted that only a very few of today's computer manufacturers venture to market their processing systems without some type of FORTRAN support. This apparent universality, while providing strong endorsement of the language's simplicity and directness, hides the fact that the continued use of FORTRAN has required numerous compromises, many of them serious. As implied by the name FORTRAN (FORmula TRANslation), the designers' primary intent was to provide a convenient vehicle for specifying programs to perform scientific computations. The original language design reflects the state of computer science and technology in the 1950s and the perception of scientific computation at that time. Since then, there have been numerous attempts to expand the language so that its users could take advantage of the major advances in the field. These efforts often were spectacularly diverse, resulting in a proliferation of FORTRAN dialects. Some of these came to be used very widely, but many more remained very local, often finding use only at the installations that developed them. This continuing divergence emphasized the need for a recognized language standard whose enforcement would guarantee some level of consistency across a wide range of computing machinery and installations.

Such a standard, developed under the auspices of the influential American National Standards Institute, was published in 1966. This provided a set of language features and rules which had to be met if producers of FORTRAN programs were to market their wares to a broad range of customers. By implication, then, it meant that FORTRAN compilers, i.e., the language processing programs that convert FORTRAN programs into equivalent sequences of computer operations, had to meet these same requirements. (A particular version of that language might be extended to include other features, but programs using these features would not be applied so easily to different types of computers.)

As the standard took hold, people ran into difficulty interpreting some of its specifications, thereby necessitating a series of revisions and amendments. More significantly, advancements in computer science and technology continued at an accelerated pace. It is no surprise, therefore, that work on a new standard began to intensify, coming to fruition in 1977 with official acceptance occurring in the following year. It is this newly defined standardized version of FORTRAN, and extensions to it defined by Hewlett Packard, that will be the central focus of this book.

While it is not crucial here to construct a history of the developments that produced FORTRAN 77, one growth factor in computer science should be discussed because of its influence on the language. During the mid-60s, inquiries into the process of writing and testing programs began to produce stronger and stronger arguments against the relatively undisciplined nature of programming. The basic contention was that traditional practices encourage the production of programs that are unnecessarily complex and, therefore, inherently difficult to analyze, correct, and modify. Moreover, a methodology was taking shape whereby the development of a program could be handled in a systematic way, with

the design, implementation and testing following a well-controlled course. By now this orderly approach, which has come to be called structured programming, has been applied successfully over a sufficiently wide range of problems to place its effectiveness beyond dispute.

A crucial aspect of structured programming lies in a set of building blocks from which well-structured programs are formed. Availability of these structural elements varies from one programming language to another; thus, recently developed languages provide extensive support for structured programming as an explicit design objective. In older languages, the inclusion of structured programming features is essentially accidental. When it came to FORTRAN (including the 1966 standard version) there was little to be found in direct support of structured programming. Attempts to follow this discipline within the available language characteristics led to awkward contrivances which, in some instances, intensified the complexity they were supposed to relieve.

Acknowledgment of these difficulties, together with the recognition of structured programming's advantages, provided important motivation for many of the major features introduced in FORTRAN 77. Agreement was not unanimous on the extent to which the language should be changed. Consequently, the resulting standard can be viewed as a somewhat cautious step forward, with individual implementers enhancing the language as they saw fit. HP has taken a more decisive approach by providing extensions that strengthen FORTRAN's support of structured programming. Thus, HP FORTRAN 77 and structured programming are bound together and this book emphasizes that interrelation. Accordingly, we shall describe the features of HP FORTRAN 77 by illustrating their use in numerous well-structured examples. In this way, the characteristics of the language and the principles and techniques of structured programming reinforce each other. Since the HP extensions blend smoothly and logically into the fabric of the language, the book does not make an explicit distinction between the standard features and the enhancements to them. The result is an attractive text for users of HP FORTRAN 77 regardless of the extent of their previous FORTRAN background. For those students with programming background in any language, the book is an orderly presentation of HP FORTRAN 77 emphasizing effective design and implementation programs through the use of good, clear, logical structure.

There is no intent to give equal weight to all of HP FORTRAN 77's features. (A complete definition is given in the FORTRAN 77 Reference Manual supplied with your system. Many of the features are in the language because they were always there, despite the addition of improvements meant to replace them, so that earlier FORTRAN programs still can be compiled. Accordingly, such features receive minimal attention, in some cases only a brief mention in an appendix. The practices they forced programmers to adopt receive no mention at all. There is no reason to memorialize poor technique. Other features, while not particularly "good" or "bad," tend to be rather specialized, so that their use is likely to be infrequent. As a matter of convenience, these features have been placed in separate parts of the book, so that the instructor may choose to address them or to bypass them without undue effect. Chapters 16, 17, 18, 19, and Appendix A are cases in point. Chapter 1 provides specific information to support routine program preparation and execution on HP operating systems, with particular emphasis on HP's powerful interactive facilities. A more complete description of these ancillary facilities is given in the appropriate Getting Started Manual for your HP system.

People familiar with introductory programming concepts may find it appropriate to skip either or both of the first two chapters, and they may do so without loss of continuity. Each concept and feature is well illustrated by means of procedural fragments and complete programs. After fulfilling its tutorial purpose, the book can continue to serve as a comprehensive reference for HP FORTRAN 77 and its effective use. There is an unusually wide selection of problems and exercises so that the instructor can select those that are most consistent with the background and interests of the particular class.

# *Acknowledgments*

I would like to express my appreciation to Jack Taylor and Tom Walker of Boyd and Fraser, and Van Diehl, Caryl Schoppet, and Jeff Lujack of Hewlett Packard for their valuable suggestions with regard to the book's organization. Thanks also are due to Tom Bugnitz of Washington University and to Phil Samuels, Howard Bomze, and Wayne Tennenbaum of Universal Sewing Supply for their help in making computerized text preparation live up to its advertised merits. Finally, special love and thanks go to our son Mark Pollack for his incisive comments about the material, to our daughter Sherie Pollack for the initial cover design (isn't it terrific?) and to my wife Sydell for being who she is.

Seymour V. Pollack
University City, Missouri

# 1
# *Introductory Concepts*

We shall be considering organizational concepts and logical techniques which provide a set of powerful tools for developing programs—sequences of instructions that control the activities of a computer. These notions, which formed and developed over recent years, have crystallized into a way of designing and writing programs known as *structured programming*. HP FORTRAN 77, the language in which we shall write programs, was designed to make it convenient to take advantage of the structured programming approach. HP FORTRAN 77 fully implements the American National Standards Institute X3.9-1978 standard (ANSI 77) for FORTRAN. It has many extensions to provide a more structured approach to program development and more flexibility in computing for scientific applications. As part of its extensions, HP FORTRAN 77 fully implements the MIL-STD-1753 Military Standard FORTRAN.

Successful use of a computer to solve a problem will depend on a combination of two basic factors:

1. Our understanding of the problem and our ability to describe its solution clearly and precisely.
2. Our ability to express that solution as a clear, correct program.

By learning and applying structured programming principles, the job of writing good, reliable programs will turn out to be a surprisingly easy one. There is nothing magic about this. The ideas in structured programming simply provide guidelines that help us attack a problem in an orderly way, making sure that we know what problem we are solving before we write a program to solve it. Another factor that makes the programming process easier than it was lies in the language itself. HP FORTRAN 77 enables us to specify complicated computations in simple ways. Thus, once we know what we want to do, we shall not have much difficulty in directing the computer to do it.

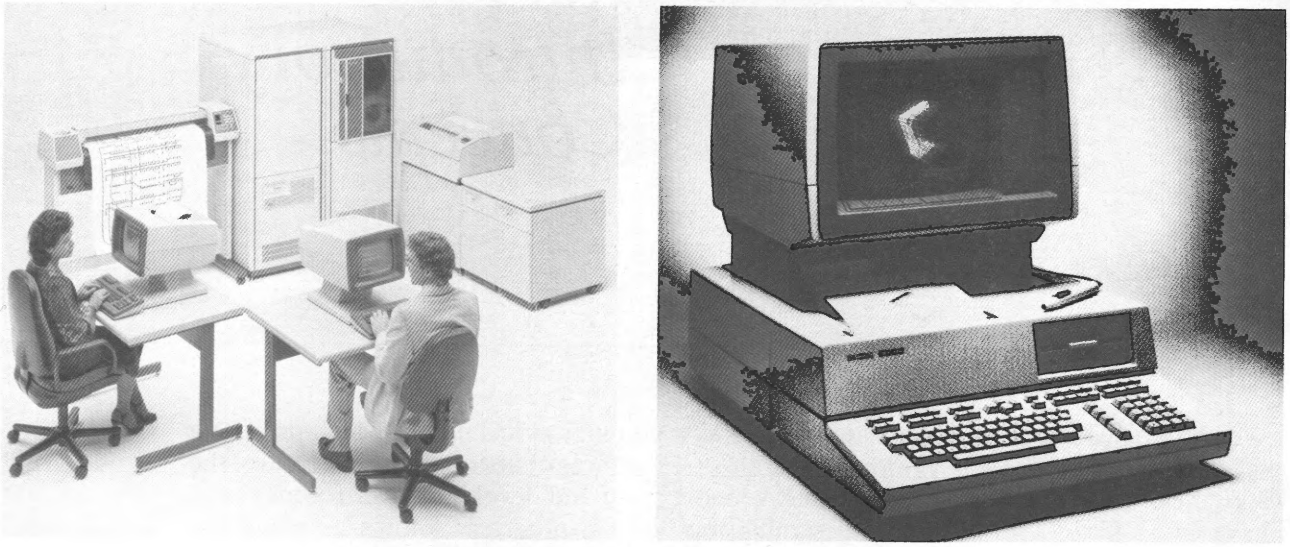
To help build a useful perception of the programming process as a whole, we shall examine briefly the overall functional principles of an HP computer system and the role of FORTRAN 77 in the use of such a system.

## 1.1 MAJOR COMPONENTS OF AN HP COMPUTER SYSTEM

Figure 1.1A shows an HP 1000 processing system and Figure 1.1B shows an HP 9000 desktop computer system. Figure 1.2 shows the major functional components of a computer and how they are related to each other. We shall be referring to Figure 1.2 as we examine each of the functional components in turn.

### 1.1.1 The Central Processing Unit (CPU)

The heart of the computer system is the processor, that component where the actual computing is performed. The capabilities of a particular processor are embodied in its machine language, a collection of instruction types that the processor is designed to



(a) (b)

**FIGURE 1.1** **(a)** HP 1000 Computer System. **(b)** HP 9000 Computer System.

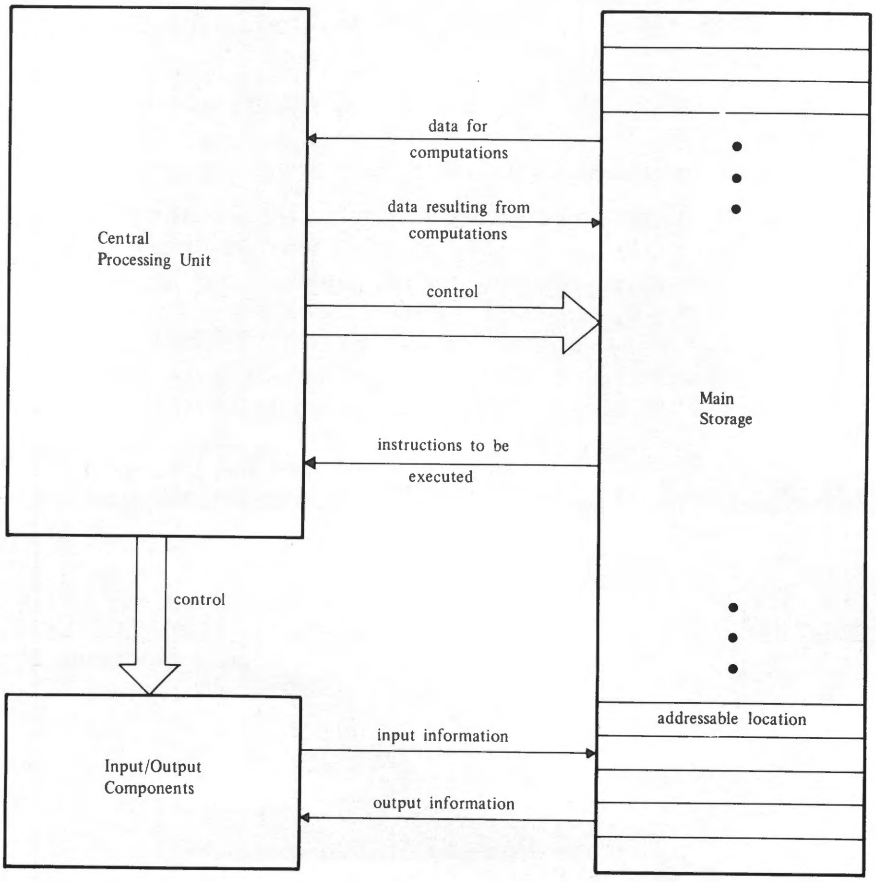



**FIGURE 1.2** Major Functional Relationships in a Computer.

recognize and execute. Each instruction represents a particular type of activity. Examples of such instruction types include the following:

"Add these two values together."

"Compare these two values and set a signal if they are equal."

"Copy this value into that part of the machine."

HP processors recognize well over one hundred instruction types, with single instructions describing relatively complicated mathematical or logical activities. When we want to do anything on the processor, therefore, we must specify that task as a sequence of instructions (i.e., a *program*) in the particular machine language for the processor to be used. Later in this chapter we shall discuss the automatic mechanisms that relieve us of the inconvenience involved in preparing such specifications.

### 1.1.2 Main Storage

Whenever any processor does anything, it is running (*executing*) a program. Certain processors, designed to do the same specific job each time they are activated, will have the program specifying that job built into the circuitry. These *special purpose* systems appear in a wide variety of devices including such diverse items as typewriters, burglar alarms, sewing machines, and video games. HP computer systems are *general purpose* processing systems where the type of job done by the processor depends on the particular program it is running. There is no limit to the number or diversity of programs that can be designed for it, and a given program can replace any other program in an instant.

Earlier, we said that programs are sequences of instructions. Thus, in order for a program to be run, its instructions have to be brought to the processor where each one is examined, decoded, and executed in turn. The component that supplies these instructions is called main storage or main memory. Unlike the processor, this component is passive in that it does not do anything. Rather, it acts very much like a library, supplying instructions to the processor on demand. When the processor executes one of these instructions, some particular operation is performed on data available in the processor. These data also are brought to the processor from main storage when they are needed and returned to main storage when they are not.

Main storage on HP processors is divided into individual cells, with each one uniquely identified within the system design. This identification is expressed as a permanent *address* used by the processor to refer to a desired part of main storage. Thus, a processor's reference to main storage, when represented in human terms, might say something like "Give me the information currently stored in the cell located at address 568." As a result, the information in the specified cell will be copied and that copy will be delivered to the processor. The original information, still retained in the cell at address 568, may be copied again and again, on demand, until we replace it with other information.

Each cell in HP main storage is designed to hold a single character of information (a numerical digit, a letter of the alphabet, a punctuation symbol, or some other special character). Storage that can accommodate a single character of information is called a byte. Main storage also is divided into *words*, where each word is two bytes long for HP's 16-bit computers and four bytes long for HP's 32-bit computers. The HP hardware and FORTRAN 77 build on these fundamental organizational properties to provide capabilities for dealing with multiple-word groupings as single computational entities.

One of the principal ways of expressing the size of a computing system is to report the number of addressable cells in main storage (i.e., the number of different addresses for which the system is equipped with storage cells). HP main storage can accommodate as many as 4 million individually addressable bytes.

The speed of a computer depends to a considerable extent on the design and physical construction of the processor and main storage circuitry. A common basis for comparison

**FIGURE 1.3** HP 2647A CRT Terminal.

is to report the time it takes to perform an elementary operation (i.e., adding two numbers together). HP processors can execute in excess of one million such instructions per second.

### 1.1.3 Input/Output Components

To complete a general purpose computer system it is necessary to include facilities that enable the central processor to communicate with the outside world. A wide variety of input devices are available from which processors can read programs and/or data on which those programs operate. As indicated in Figure 1.2, the information thus read is placed in main storage. Similarly, when a program directs the processor to send out some product of its computational labors, the resulting activity will write the specified information from main storage onto an *output* device. The enormous variety of input/output devices make a complete description totally impractical. (As you read this sentence, approximately 18.73 new input and/or output devices are being announced.) However, we shall mention a few of the more common types.

**1.1.3.1 Keyboard Devices** A good deal of the input supplied to computers is prepared originally on some type of device equipped with a keyboard much like a typewriter. One such device, has the keyboard as part of a cathode ray tube (CRT) (Figure 1.3). The keystrokes are converted into electronic impulses for direct reading by the processor. At the same time, the CRT produces a visual image on a screen. The image may be a direct transcript of the information typed on the keyboard or it may be information sent to the screen from the processor in response to the signals received (by the processor) from the keyboard.

**1.1.3.2 Printing Devices** There are many situations in which computers are programmed to control other machines. In such cases the computed results take the form of electronic signals which, in turn, influence the activity of some automatic device. Here, however, our interests will focus on those types of processes in which the results are to be examined by humans. Accordingly, there is a wide range of printing devices which can be connected directly to the HP processor. One such device is shown in Figure 1.4. The video terminal serves double duty in that it will display the processor's output (when directed to do so) as well as typed input. In addition, there are printing devices (such as the one in Figure 1.5) designed to produce graphic output with sufficiently high resolution to serve as engineering drawings.

**FIGURE 1.4** HP 2608 Line Printer.

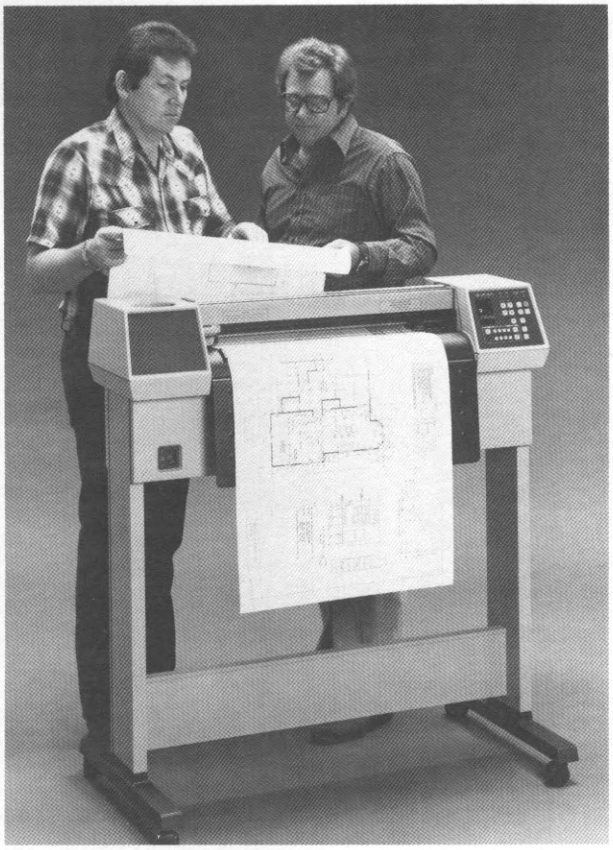


**FIGURE 1.5** HP 7585A Graphics Plotter.

**FIGURE 1.6** HP 7970E Magnetic Tape Drive.

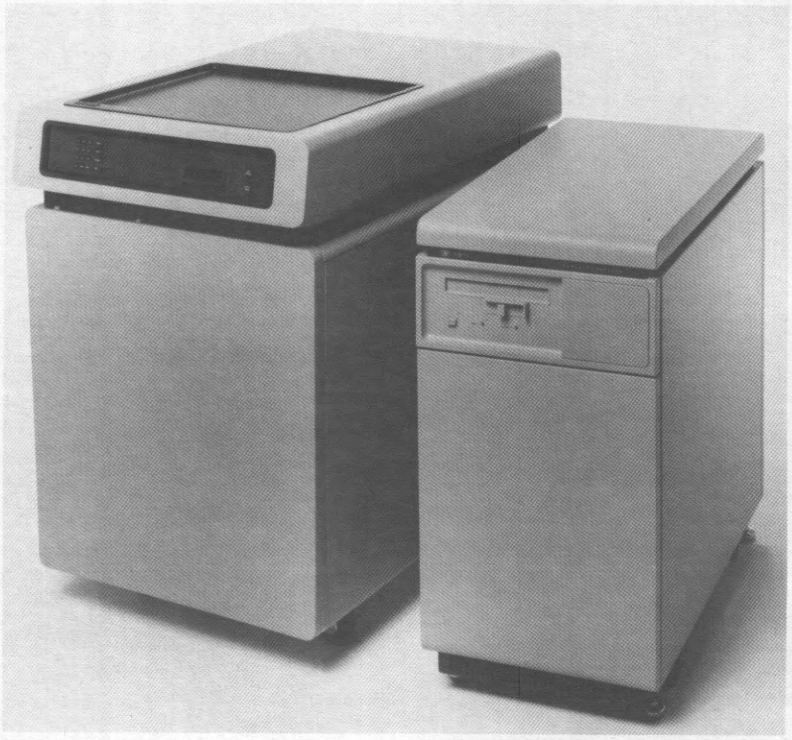


**FIGURE 1.7** HP 7933 and HP 7912 Disk Drives.

**1.1.3.3. Secondary Storage** In addition to these devices, HP systems are equipped with auxiliary storage devices which have both input and output capabilities. These devices use some type of magnetic material for storing large amounts of data which can be transmitted rapidly to and from main storage on demand. Main storage, though also magnetic, is constructed differently. Of greater importance is the fact that main storage is the only place from which instructions can be delivered to the processor. Popular forms for such magnetic secondary storage include reels of magnetic tape, flat circular disks with magnetic surfaces, and tape cassettes. Every HP system is equipped with a library of programs, any one of which may be required for use at some time. Attempts to keep such a library in main storage would require far greater capacity than is available even in the largest systems. Instead, the library is stored on an auxiliary device (such as a magnetic disk unit). Then, when a particular program is needed, it is copied from the disk into main storage. Another example is seen in the situation where a program, in the course of its activity, produces an enormous amount of data. As these data are generated, they are sent out of main storage and written onto a magnetic medium for later use, thereby making that part of main storage available for the next set of data items. In this situation, the auxiliary unit serves as an output device. Moreover, HP's magnetic disk units incorporate their own "secondary" storage device in the form of a magnetic tape cartridge to which the disk's contents can be copied easily. When the program concludes, the magnetic tape can be removed from its unit and stored for subsequent use. Later on, that same tape can be installed on the device from which it was removed for use as input. Examples of tape and disk units are shown in Figures 1.6 and 1.7, respectively.

## 1.2 PROGRAMS AND PROGRAMMING LANGUAGES

To provide a processor with instructions that its circuits can execute, each instruction has to be stored in main memory as a coded string of ones and zeros. As was pointed out earlier, many of these machine language instructions represent relatively simple activities, so that any type of substantial computation is likely to require an extensive program.

When computers first were introduced, people had to write programs in machine language. Suppose a single simple instruction on one of these machines appeared as follows:

```
11110010001001010011000000011110
```

This example is not unrealistic. With that in mind, imagine the trouble required to prepare and keep track of several hundred such strings. It is not surprising, then, that many programmers were happy to promote the image that programming was Highly Exotic and Mysterious Work performable only by those who had been given the Special Key to Open the Great Lock.

### 1.2.1 Properties of a High Level Language

Relief for the machine language programmer did not come from drastic changes in machine languages. Rather, a growing understanding of information processing brought help in the form of *high level languages.* By using such a language, a programmer can "instruct" the machine in a more convenient form much closer to natural language. Instead of expressing a procedure in terms of the minute operations that the machine can interpret, the task can be specified as a sequence of steps that reflect the requirements of the problem being solved. For example, suppose we wanted to compute the distance DISTANCE covered in time TIME by a moving object with starting velocity INITIAL_VEL and acceleration ACCEL. The result is obtained by means of the simple formula

$$\text{DISTANCE} = \text{INITIAL_VEL} \times (\text{TIME}) + 0.5 \times (\text{ACCEL}) \times (\text{TIME})$$

If we had to break this down into the elementary activities required at the machine's level, a rough translation of the machine language instructions would be as follows:

1. "Move INITIAL_VEL from main storage to the machine's computational component."
2. "Multiply INITIAL_VEL by TIME."
3. "Move the product INITIAL_VEL × TIME out of the computational component for temporary storage."
4. "Move ACCEL into the computational component."
5. "Multiply ACCEL by TIME."
6. "Multiply the value just computed by TIME"
7. "Multiply the value just completed in step 6 by 0.5."
8. "Add the product INITIAL_VEL × TIME to the value just computed."
9. "Assign the value just computed to the variable DISTANCE."

The same computation, when expressed in the FORTRAN 77 language, can be written like this:

```
DISTANCE = INITIAL_VEL * TIME + 0.5 * ACCEL * TIME * TIME
```

While this does not look exactly like the formula given before, it comes close enough so that it is a simple matter for us to adapt from traditional algebraic forms to those required by the high level language.

Thus, a high level language gives us a vehicle for expression that can be more comprehensive (and more meaningful, in our terms) than the simple machine instruction. Each unit of expression in a high level language is called a *statement.* It is basically the same as a sentence in a natural language. A program written in a high level language is called a *source program.*

```
PROGRAM              EX101
REAL                 INITIAL_VEL, TIME, ACCEL, DISTANCE

PRINT *, 'SUBMIT VALUES FOR INITIAL_VEL, ACCEL, AND TIME. '
READ *, INITIAL_VEL, ACCEL, TIME

DISTANCE = INITIAL_VEL *TIME + 0.5 *ACCEL *TIME *TIME

PRINT *, 'INITIAL VELOCITY: ', INITIAL_VEL
PRINT *, 'ACCELERATION: ', ACCEL
PRINT *, 'TRAVEL TIME: ', TIME
PRINT *, 'RESULTING DISTANCE; ', DISTANCE

PRINT *, 'RUN COMPLETED. '
STOP
END
```

**FIGURE 1.8** FORTRAN Statements for Example 1.1

**Example 1.1** To illustrate the convenience of a high level language, we shall take a look at our first FORTRAN PROGRAM (Figure 1.8). This program, named EX101, computes a distance DISTANCE for a set of INITIAL_VEL, ACCEL, and TIME submitted to it, using the formula that was shown earlier in this section. At the program's start, the variables INITIAL_VEL, TIME, ACCEL and DISTANCE are defined, and the user is asked to submit input values for INITIAL_VEL, ACCEL, and TIME. These are typed in by the user, and the program brings them into the processor. Then, DISTANCE is computed and the program displays five lines of output: Each of the input values is produced on a separate line, along with a brief description of its meaning. For example, the statement

```
PRINT*, 'TRAVEL TIME: ', TIME
```

displays the actual message inside the apostrophes, followed by TIME's value. The computed result (DISTANCE) is printed on the fourth line, and a final line indicates that the program has finished processing. A sample run for this program is shown in Figure 1.9.

```
SUBMIT VALUES FOR INITIAL_VEL, ACCEL, AND TIME.
100.5    12.6    21.8 [RETURN]              (typed by user)
INITIAL VELOCITY:     0.1005000E  03
ACCELERATION:    0.1260000E  02
TRAVEL TIME:      0.2180000E  02
RESULTING DISTANCE:      0.5047572E  04
RUN COMPLETED.
```

**FIGURE 1.9** Sample Run for Example 1.1

Of course, this little program is unrealistically simple, but it does give some idea of the flavor of the language. We shall find that other language features are equally convenient and easy to understand.

### 1.2.2 The High Level Language Compiler

Since we have already noted that the computer itself is not equipped to recognize and handle anything as complicated as a statement in a high level language, there must be something between the high level language and the computer that transforms such statements into groups of machine language instructions which, when used together, perform the same activity. This conversion is done by a program called a *compiler* that analyzes each statement to determine the type of activities it specifies. The analytical process is supported by a dictionary and a set of rules, both of which are built into the compiler, thereby enabling it to determine what kind of statement has been submitted and whether or not its construction follows the rules of the language. Another set of

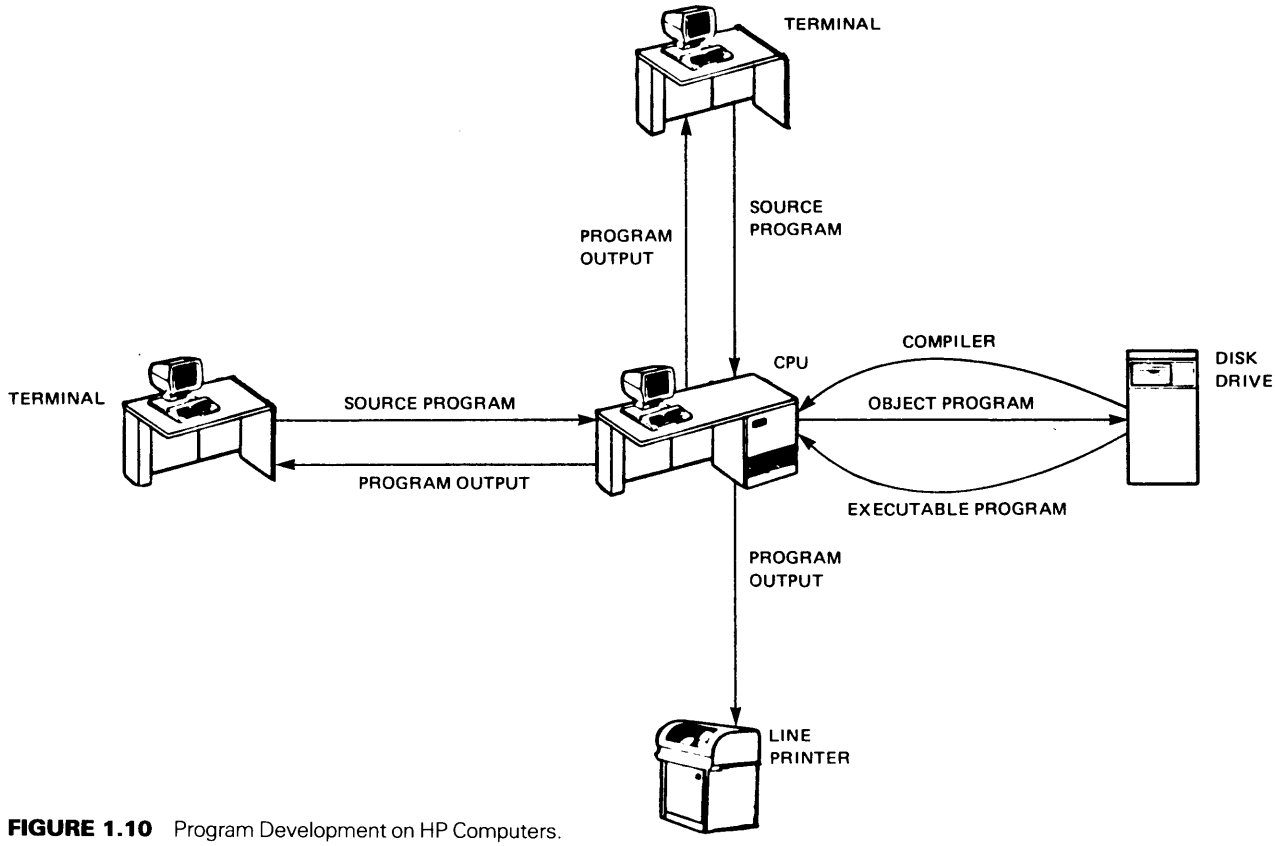



**FIGURE 1.10** Program Development on HP Computers.

internal rules guides the compiler in defining the machine instructions required to carry out the intent of the statement. The result of the compiling process is a new program (the one that the machine actually will execute) in which each statement has been replaced by the equivalent activity expressed in terms of individual machine instructions. This new program is called an *object program*.

Figure 1.10 gives an overview of the general process: The source program, shown in the figure as having been prepared by the programmer on a terminal, is submitted to the processor via its telecommunication lines. That program is analyzed and processed by the compiler, which was copied into the processor from a program library stored on a magnetic disk. As a result of its work, the compiler produces a set of machine language instructions (the object program) which are sent to a disk for temporary storage. Then, the completed object program is brought back into the processor (once the compiler is no longer needed and has been cleared out), where it executes. The output produced by that program is shown as being sent to a printing device or (alternatively) to the terminal from which the program (and data) originated.

From these concepts we can form a picture of a *programming system* consisting of two parts:

1. The high level language itself, providing a vehicle for describing computerized procedures in convenient and meaningful form.
2. A compiler for automatically converting high level language descriptions to programs that the machine can execute.

If the compiler is designed properly, it will be invisible to its users. That is, its entire range of activities will be submerged within the computer's operating environment so that the

programmer working in a high level language will be able to write programs as if the machine were executing them directly in that language. The HP FORTRAN 77 programming system falls into this category. It can be used quite successfully without knowing a great deal about the nature or structure of the machine language that the compiler produces.

## 1.3 THE HP PROGRAMMING ENVIRONMENT

A modern computer's power and versatility cannot be fully exploited without a constellation of software to mediate between the user and the machinery. The HP FORTRAN 77 compiler is just one of many software products that interact to provide a hospitable environment for the preparation, storage, use, and maintenance of a program. This section introduces these aspects of the HP software resources that are directly involved in the development of FORTRAN 77 programs.

### 1.3.1 FORTRAN 77 and the HP Operating System

The integrated collection of software that oversees a computing system's activities is called an *operating system*. HP's operating systems include facilities that enable the FORTRAN 77 programmer to gain access to a wide range of the system's resources conveniently and "naturally." By this we mean that the programmer can oeprate successfully under the illustion that the HP machinery itself "understands" FORTRAN. The multiple layers of software between user and hardware intercommunicate smoothly and automatically so that they do not intrude to upset this illusion.

**Example 1.2** To get some idea of the relation between FORTRAN 77 and the operating system, we shall take a look at a modified version of Example 1.1. This time, we shall place the program in context as an item to be manipulated by the operating system. The program itself will be modified to enable it to process any number of input values, computing and displaying a new value of DISTANCE for each set of input. An initial velocity value of zero will be used to terminate the processing.

This ability to perform the computations repeatedly will be provided by setting up a cycle, called a *loop*, that automatically repeats itself as long as there are input values to process. A separate action, outside the loop, gets the loop started by bringing in the first set of values for INITIAL_VEL, ACCEL, and TIME. The resulting program is shown in Figure 1.11. Note that there are two READ statements, each containing the same list of names. The first READ statement is the one that gets the process started by providing an initial set of input values. Subsequent values for INITIAL_VEL, ACCEL, and TIME are

```
PROGRAM              EX102
REAL                 INITIAL_VEL, TIME, ACCEL, DISTANCE

PRINT *, 'SUBMIT FIRST VALUES FOR INITIAL_VEL, ACCEL, AND TIME. '
READ *, INITIAL_VEL, ACCEL, TIME

DO WHILE (INITIAL_VEL .NE. 0)
   DISTANCE = INITIAL_VEL * TIME + 0.5 *ACCEL *TIME *TIME
   PRINT * , 'INITIAL VELOCITY: ' , INITIAL_VEL
   PRINT * , 'ACCELERATION: ' , ACCEL
   PRINT * , 'TRAVEL TIME: ' , TIME
   PRINT *, 'RESULTING DISTANCE: ' , DISTANCE
   PRINT * , '          '
   PRINT *, 'SUBMIT NEW VALUES FOR INITIAL_VEL, ACCEL, AND TIME. '
   READ * , INITIAL_, ACCEL, TIME
END DO

PRINT * , 'RUN COMPLETED. '
STOP
END
```

**FIGURE 1.11** FORTRAN Statements for Example 1.2

```
SUBMIT FIRST VALUES FOR INITIAL_VEL, ACCEL, AND TIME.
60    1.5    75 [RETURN]
INITIAL VELOCITY:     0.6000000E  02
ACCELERATION:     0.1500000E  01
TRAVEL TIME: 0.7500000E  02
RESULTING DISTANCE:     0.8718750E  04

SUBMIT NEW VALUES FOR INITIAL_VEL, ACCEL, AND TIME.
5.8    2.2    30 [RETURN]
INITIAL VELOCITY:     0.5800000E  01
ACCELERATION:     0.2200000E  01
TRAVEL TIME: 0.3000000E  02
RESULTING VELOCITY:     0.1164000E  04

SUBMIT NEW VALUES FOR INITIAL_VEL, ACCEL, AND TIME.
200    0.72    41 [RETURN]
INITIAL VELOCITY:     0.2000000E  03
ACCELERATION:     0.7200000E  00
TRAVEL TIME: 0.4100000E  02
RESULTING DISTANCE:     0.8805160E  04

SUBMIT NEW VALUES FOR INITIAL_VEL, ACCEL, AND TIME.
0    0    0 [RETURN]
RUN COMPLETED.
```

**FIGURE 1.12** Sample Run for Example 1.2

read by the second READ statement that is executed over and over again as the loop goes through its repetitions. The DO WHILE and END DO statements mark the boundaries of the loop so that FORTRAN 77 "knows" how much of the program to repeat. Conttrol of the repetitions is built into the DO WHILE statement, and this mechanism operates as follows: When the loop is entered for the first time, the program goes to work on the initial set of values provided by the first READ statement. After computing and displaying DISTANCE, the program asks for (and, presumably, receives) the next set. Then, the END DO statement automatically sends the program back to the beginning of the loop (i.e., the DO WHILE statement) where INITIAL_VEL is tested. As long as it is not zero (the . NE. means "is not equal to"), the program is allowed to go through the loop another time. At the end of that cycle, we have a new set of input values, and the test for a zero value of INITIAL_VEL is performed again. As soon as the test finds a zero, the entire loop is bypassed automatically and the program continues at the statement immediately following the END DO. A sample run for this program is shown in Figure 1.12.

Now that the statements themselves are defined, we can turn our attention to the surrounding environment with which the program must interact. Before the program can be used to produce computed results, it must be coverted to a functionally equivalent sequence of machine language instructions that the HP hardware is designed to execute. Consequently, it must be translated by the HP FORTRAN 77 compiler. Until that happens, its identity as a program is only in our minds. As far as the HP operating system is concerned, the statements are nothing more than a collection of text materials, i.e., a sequence of letters, numbers, other symbols, and blanks. Such a collection of data is called a *file*.

A substantial resource within the HP operating system is designed to support the preparation, storage, retrieval, and maintenance of files. This includes a general text processing facility that is insensitive to the contents or "meaning" of the files on which it operates. Consequently, its details are not the subject of this book. (See the Text Editor Reference Manual supplied with your system.) We mention the file system to emphasize the fact that the FORTRAN 77 programmer's involvement with the operating system starts at the earliest stages of program preparation.

While the program statements are being written with the use of a text editor, they are treated simply as strings of text being submitted to the system to form a file. A text editor is designed to support the programmer in this initial stage of program preparation. Under its jurisdiction, the programmer can build the file, make any desired editorial changes (e.g., repair typographical errors, insert additional text, delete unwanted text), give the file a name, and store it in the system's library for use later on. The text in the file consists of the FORTRAN statements and compiler directives. The latter provides directions that the compiler can follow when it processes that file. This source file can now be processed by the activation of the FORTRAN 77 compiler.

### 1.3.2 Compiling and Running an HP FORTRAN 77 Program

Once a FORTRAN 77 program has an identity as a file, we can make it available to other system resources (such as the editor or FORTRAN 77 compiler). We are interested specifically in compiling and running the program (and re-editing if necessary). The typical sequence of events is outlined below. Remember, we are starting with a program consisting of a sequence of FORTRAN statements (called a source program) already edited and stored as a file within the operating system:

1. At the programmer's request, the operating system initiates a process whose first step is to bring in the HP FORTRAN 77 compiler. This program will analyze the programmer's FORTRAN statements and translate them into a functionally equivalent sequence of instructions in the computer's machine language. These instructions are said to be in *object code*.
2. Organizationally, the object code is another file within the operating system. As a program, it still is incomplete because its instructions include requests for other programs or parts of programs. These must be copied from one or more libraries that exist within the operating system, and the copies must be combined with the object code. This process, called *linking*, is automatically performed by the loader. In fact, the programmer often is unaware that these additional components have been requested. To perform this activity, the operating system dispenses with the FORTRAN compiler and places the object code at the disposal of the programs that will handle the linking operations. Once the linking is complete, the resulting program must be placed in main storage so that it can run. This process, called *loading*, is performed next.
3. The only thing remaining to be done is to turn control of the machine over to the program. When that happens (also automatically), the program begins executing.

Additional details regarding this process and the operating system facilities that support it are given in Chapter 7 of the "FORTRAN 77 Reference Manual" number 92836-90001.

## 1.4 GLOSSARY

**address** The permanent numerical designation associated with a particular location in main storage.

**byte** The amount of storage necessary to accommodate a single character of information. A byte consists of eight bits.

**CPU** Central processing unit or processor: that part of a computing system containing the circuits that analyze instructions and perform the actual computations.

**character** An item of information represented by a single symbol (e.g., a single digit, letter, punctuation mark); the amount of information that can be stored in a byte of memory.

**compiler** A program that analyzes source programs consisting of high level language statements and produces an operationally equivalent object program in a machine language.

**file** An arbitrary collection of text identified by a unique name that incorporates the text into an overall system library.

**FORTRAN 77** An acronym from FORmula TRANslation, reflecting the major intent of the high level language so named.

**hardware** The physical components comprising a computer system.

**high level language** A programming language whose statements are too complex for a processor to execute directly. FORTRAN 77, COBOL, PL/I, BASIC, ALGOL, and PASCAL are names of some widely used high level languages.

**input** Information submitted to a program for processing. For example, a FORTRAN 77 source program represents input submitted to the FORTRAN 77 compiler for processing.

**instruction** The unit of activity recognized by a processor; a member of a processor's machine language.

**machine language** The collection of instruction types comprising the range of elementary activities that a particular processor is designed to perform.

**main storage** The memory component of a computer system used for storing instructions and data submitted to or produced by a program.

**object program** A sequence of machine language instructions produced by a compiler as the result of processing a source program; output developed by a compiler.

**operating system** An integrated collection of programs that supervise and monitor the ongoing activities in a computing system. Among its other duties, an operating system keeps track of the system's resources and fills users' requests for such resources.

**output** Information transmitted from the processor to the outside world.

**program** A sequence of statements or instructions describing a set of activities for a computer to perform.

**prompt** A signal sent from a program to a user at a terminal indicating that the program requires information from the user before it can proceed.

**reading** Transmission of input to a processor.

**source program** A program written in a high level language and, as such, not directly executable by a computer; input to a compiler for ultimate translation to an object program.

**statement** The unit of expression (i.e., a single conceptual activity) in a high level language.

**word** A unit of main storage usually consisting of two or four bytes.

**word length** The number of bytes contained in a single word of main storage for a particular type of CPU; on many processors the word is the smallest addressable unit of main storage (i.e., each word has its own unique, permanently assigned address).

**writing** Transmission of information from a processor to the outside world.

**PROBLEMS**

1. To help get acquainted with some of the computer terminology, find out what you can about the computing system on which you will be writing FORTRAN programs. For example, see if you can find out the following:
   (a) the model of the HP computer and the year in which it was introduced.
   (b) the size (capacity) of the system's main storage.
   (c) the number and type of each peripheral unit attached to the system.
   (d) some measure of the processor's speed. (For example, how many additions will it perform in one second?
   (e) the cost of the system.
   (f) the number of different instructions there are in your processor's machine language.
2. Become acquainted with the computer industry by obtaining information on any or all of the following:
   (a) the approximate number of professional programmers in the United States and/or in your city.
   (b) the names and headquarter locations of at least three professional organizations in computer-related fields.
   (c) the number of HP installations in your city.

3. Find out the names of at least five high level languages besides FORTRAN that are in general use. Determine some fundamental facts about each language. For example:
   (a) what does the language's name stand for?
   (b) when was the language first introduced?
   (c) what is the major purpose of the language? How does that purpose differ from the one associated with FORTRAN?

4. From Section 1.2.2, it is clear that a high level language compiler is itself a program. Try to find out how large your FORTRAN compiler is (i.e., the number of instructions contained in your compiler).

5. If we study the sample program given in Section 1.2.2, it will become apparent, even before we know the details of the language, that certain types of statements are designed to perform specialized operations. Based on your observations, identify:
   (a) the statement or statements that perform input operations.
   (b) the statement or statements that perform actual computations.
   (c) the statement or statements that perform output operations.

6. Using Example 1.1, follow the step-by-step operations presented in that program and show what the results would look like for each of the following sets of input data:

| | | | |
|---|---|---|---|
| (a) | 100.0 | 20.0 | 100.0 |
| (b) | 54.6 | 0.0 | 120.0 |
| (c) | 0.0 | 30.0 | 60.0 |
| (d) | 220.0 | −2.5 | 20.0 |

7. Assume that Example 1.1 were changed so that it appears as follows:

```
PROGRAM              EX101A
REAL INITIAL_VEL, TIME, ACCEL, DISTANCE
PRINT *, 'SUBMIT TIME, INITIAL_VEL, AND ACCEL. '
READ *, TIME, INITIAL_VEL, ACCEL
DISTANCE = INITIAL_VEL * TIME + 0.5 * ACCEL * TIME * TIME
PRINT *, 'INITIAL VELOCITY IS: ', INITIAL_VEL
PRINT *, 'ACCELERATION IS: ', ACCEL
PRINT *, 'TIME FOR TRAVEL IS: ', TIME
PRINT *, 'DISTANCE COVERED IS: ', DISTANCE
PRINT *, 'NORMAL PROCESSING. RUN COMPLETED. '
STOP
END
```

   Show what the output would look like for each of the input sets given below:

| | | | |
|---|---|---|---|
| (a) | 100.0 | 36.5 | 10.10 |
| (b) | 120.0 | 0.0 | 15.6 |
| (c) | 60.0 | 120.0 | −5.0 |

# 2
# *The Structured Programming Process*

It would be irresponsible to claim that the preparation of computer programs is a science. However, it is just as misleading to insist that this activity is totally an art, with all of the intangibles that an art implies. An enormous amount of experience and study has made it possible to introduce considerable discipline into the programming process. As a result, there has been a rapidly growing acceptance of something called structured programming.

Before this groundswell toward structured programming conjures up any pictures of miraculous cures for all computer-related ailments, it will be helpful to say something about what structured programming is. In general, the term refers to a collection of concepts, techniques, attitudes, and rules of thumb which help to *increase the likelihood of producing a computer program that is correct, clear, simple (for a given set of requirements), easy to analyze, and easy to use.*

These rules and guidelines are quite simple. Consequently, it will be unnecessary to build an elaborate drama around them. We shall introduce the basic ideas of structured programming in this chapter and apply them throughout the rest of the book. Since the approach makes so much sense, its use will develop naturally, without any need to force it. Accordingly, each program we study or write will serve two purposes: It will illustrate some programming technique and/or language feature that offers a convenient way to handle certain problems. At the same time, the overall program will provide yet another example of good structure.

Although we shall be concentrating on the language features and their use in writing good programs, it is important to recognize that this activity is just one part of a more comprehensive process—the design and development of computer applications. Think of a computer application as a situation in which we arrange for a computing system to do a useful piece of work (e.g., keep track of books in a library, prepare a company payroll, or continuously check a patient's heartbeat for abnormal rhythmic patterns). From this point of view, the program is a *product* that enables the computer to do its work. Thus, when we write the program, we *fabricate* that product. Accordingly, a program is like any other manufactured article. We cannot produce it until we establish:

1. What we need the computer to do.
2. Exactly what kind of program it will take to make the computer do what we need.
3. Exactly how we are going to produce that program.

In the next few sections of this chapter we shall sketch the basic characteristics of the steps that precede the writing of a program. Then, with this as background, we can focus on structured programming itself, i.e., the identification and use of effective and reliable "program manufacturing" techniques.

## 2.1 SPECIFICATION OF THE PROBLEM

For many people, writing programs is fun. These otherwise reasonable folk find it enjoyable to compose instructions that a complex computing system eventually will obey to the letter. As a result, there often is an understandable temptation to begin writing a program as soon as we know that a program needs to be written. When this happens, the resulting program may be useless, even though it works. The reason is simple: The useless program solves the wrong problem. Eagerness to get the job done is a good thing, as long as there is a complete understanding of what the job is.

We would not expect a manufacturer to begin production saying, "We don't know what we're making; let's see what it turns out to be when we get finished." Yet this happens surprisingly often with people who write programs.

An even more common occurrence, perhaps not as extreme, is one in which program writing begins as soon as there is some idea of what is needed. The notion behind getting an early start is to complete the definition as the program is being developed. Since the person writing the program often is different from the one setting the requirements, this "headstart" appears to be more "efficient" because everybody is doing something. In such a case a manufacturer might say, "It hurts me when all these expensive machines are sitting idle. We'll start making something as soon as we think we know what it is, correcting it as we go. When we have finished, if luck is with us, the result will be what the customer needs."

Not surprisingly, many projects have ended up costing many times more than they should because all or part of the programming has had to be redone several times to keep up with changing requirements. On top of that, computer applications ordinarily requiring three or four months for completion often have been "streamlined" by this approach so that the development time was "reduced" to a year. Numerous companies in the programming business went broke because they agreed to provide a program for a certain price by a certain date before making sure that they and the customer agreed on exactly what problem the program would solve. An increasing number of organizations using computers are aware of this possibility. They have established formal procedures in which a problem definition must be read and signed by all concerned parties before any subsequent activity can begin.

## 2.2 PROBLEM SOLUTION AND ALGORITHMS

Once the nature of the problem is understood, we have taken a major step in reducing the chances of solving the wrong problem. To help devise an effective solution to the right problem, we must make a very important distinction: *Finding a solution and writing the program for it are different activities.* In general, it is most effective to begin the programming process only when a solution has been worked out and people are convinced that it is a correct one. Another way of saying this is that a problem is solved by an *algorithm*, and a program is a way of expressing an algorithm so that a computer can be used to carry out its intent.

### 2.2.1 What is an Algorithm?

An algorithm is a set of rules for getting something done. Since algorithms exist or can be devised for an infinite variety of activities, there is no automatic connection between an algorithm and a computer. Paddling a canoe across a lake, making lasagne, using a computer to prepare a company payroll, and getting home from the ballgame all involve algorithms.

We can think of any number of problems for which no practical algorithms exist (such as time travel), and problems with many alternative solution algorithms (such as storing books on a shelf). A frequent concern in any human activity is to select the best algorithm from a series of choices. The development of computer applications is no exception.

Existence of a procedure does not necessarily guarantee that it is an algorithm. Booksellers' shelves are filled with procedures for losing weight, quitting smoking, win-

ning in the stock market, and keeping people from taking advantage of you. Each of these, and countless other procedures, represents a set of steps which are to be followed in a certain sequence to achieve a desired result. Since the use of the algorithm may involve time and expense, it is necessary to make sure that the proposed solution actually will solve the problem it is supposed to solve.

### 2.2.2 Properties of Algorithms

As the previous sections illustrate, not every set of rules constitutes an effective algorithm. Consequently, as part of the process of deciding whether a particular procedure is suitable for computerization, we must convince ourselves that the procedure meets certain requirements associated with an algorithm.

First, an algorithm is *finite*. That means that if we follow the rules specified by the algorithm, the procedure eventually will come to an end. Fulfillment of this requirement is sometimes less obvious than it appears. Consider the following example:

**Example 2.1**

**Problem:** The Gaack Brewery needs to adjust its water supply so that it has the proper acidity for Gaack's unique flavor and aroma. As little as a gallon of acidifier per vat of liquid is enough to change the meter reading.

**Solution:** Perform the following steps:

1. Add a gallon of acidifier.
2. Test the acidity. If it is right, stop; if not, go back to Step 1.

This procedure seems simple enough. The cautious addition of one gallon at a time ought to keep us from overshooting. However, the unfortunate brewery is unprotected if they happen to start out with liquid that is already too acidic. If this procedure were to be followed mechanically under such circumstances, the addition of acidifier would go on indefinitely, since each test would produce an improper reading. These algorithmic difficulties are not far-fetched when it comes to programs, and such situations do turn up from time to time, even among experienced programmers. Once in a while we still read a gleeful report about some computing system producing 8179 paychecks, all made out to the same worker, before some alert operator shuts the thing down.

Algorithms also must be *unambiguous*. Despite some films and stories to the contrary, computers are not even a little smart. Anything we want done has to be spelled out in exact and minute detail. There is no room for a computer's judgment because it has no ability to judge. For instance, Example 2.1 (besides being potentially non-terminating) does not define what the "right" acidity is. When we take this into account, our Gaack acidity procedure might look like this (assuming the right acidity to be $6.3 \pm .02$):

1. Define a flag whose value may be either "acidic," "alkaline," or "unused." Set the flag to "unused."
2. Test the acidity level. If it is between 6.28 and 6.32, stop; if it is above 6.32, go to Step 4; if it is below 6.28, go to Step 3.
3. If the flag is "alkaline," then stop; else add one gallon of acidifier, set the flag to "acidic," and go back to Step 2.
4. If the flag is "acidic," then stop; else add one gallon of anti-acid solution, set the flag to "alkaline," and go back to Step 2.

In this version the "right level" has been clearly defined.

A third important algorithmic property is that of *generality*. It is rarely effective to have an algorithm for the solution to a specific instance of a particular problem. Instead, it is preferable to design algorithms so that they can be used without changes for a range of similar problems. Sometimes the generalization process is obvious; in other instances, considerable thought and insight may be required to see how a particular algorithm can be generalized to serve a wider range of purposes without impairing its convenience.

Another look at the Gaack water correction procedure shows that we can use it as long as we are correcting to a level between 6.28 and 6.32. If this value were to change, a new procedure would be needed for each different level. This can be avoided by restating the solution as follows:

1. Define a flag whose values may be either "acidic," "alkaline," or "unused." Set the flag to "unused."
2. Find out what the desired acidity level is for this run. Call that level A.
3. Find out what the allowable deviation is for this run. Call that D. Thus, the acidity we are matching for this run will be A−D.
4. Test the acidity. If it is between A + D and A − D, stop; if it is above A + D, go to Step 5; if it is below A − D, then go to Step 6.
5. If the flag is "alkaline," then stop; else add one gallon of acidifier, set the flag to "acidic," and go back to Step 4.
6. If the flag is "acidic," then stop; else add one gallon of anti-acid solution, set the flag to "alkaline," and go back to Step 4.

Finally, an algorithm must be *precise*, so that its steps can be understood by its users. Telling an expert cook to "bake till done" may be clear enough for that expert. However, if the intended performer is a novice known to be devastating when left alone with an oven, a more detailed specification would be needed to define exactly how we recognize "done." When the algorithm is to be carried out by a computer, the need for detailed clarity becomes all the more crucial. The computer can recognize only certain types of instructions, and any algorithm, when implemented as a program, must be expressed in terms of these instructions.

**Example 2.2**

**Problem:** Now that the basic algorithmic properties have been described, we shall apply them to a type of problem whose solution is commonly implemented on a computer: The Three Musketeers Floor Company ("All Floor One, One Floor All") rents floor polishers and sells accessories at the following current rates:

Floor machines—$3.00 per day.
Cleaning disks—$2.00 each.
Polishing pads—$4.50 each.

Each customer has a five-digit identification number (like 31264 or 00071). Special customers are given i.d. numbers beginning with 9. All such customers receive a 10% discount on their bills. The Musketeers would like to produce a bill for each customer.

The solution is straightforward enough: For each customer, we need the identification (i.e., the customer name and number), number of days the machine was rented, and the number of polishing pads and cleaning discs that were purchased. Once the total bill is computed from the rental and purchase charges, the account number will determine whether or not the discount will be applied. Assuming the required information is available, we can assign names to each data item and summarize the solution:

*Data Names*

| | |
|---|---|
| CUST_NAME | —Customer Name |
| CUST_ID | —Customer Number |
| DAYS | —Number of Rental Days |
| DISKS | —Number of Disks Purchased |
| PADS | —Number of Pads Purchased |
| RENT | —Rental Fee |
| TTL_DISK_COST | —Cost of Disks |
| TTL_PAD_COST | —Cost of Pads |
| TOTAL_COST | —Total Bill |
| ADJ_COST | —Adjusted Bill |

**Solution:** For each customer,

1. Read in CUST_NAME, CUST_ID, DAYS, DISKS, PADS.
2. For record purposes, print the input just read. (This is called an echo.)
3. Compute RENT. (RENT = DAYS × 3)
4. Compute TTL_DISK_COST. (TTL_DISK_COST = DISKS × 2)
5. Compute TTL_PAD_COST. (TTL_PAD_COST = PADS × 4.5)
6. Compute TOTAL_COST. (TOTAL_COST = RENT + TTL_DISK_COST + TTL_PAD_COST)
7. Check CUST_ID. If it is 90000 or higher, apply a 10% discount to TOTAL_COST and assign the result to ADJ_COST; if not, leave TOTAL_COST unchanged and assign its value to ADJ_COST.
8. Print the customer's bill.

All of this is very well as long as the Musketeers' prices never change. (That is unrealistic, even in the best economic times.) Consequently, one step toward generalization that comes to mind immediately is to include the flexibility for varying prices. Instead of building these figures into the procedure as *constant values*, we can redefine them as *variables* to be read in with each run. When we do that, our revised algorithm, along with additional definitions, would appear as follows:

*Additional Data Names*

| | |
|---|---|
| DAY_RATE | —Machine rental rate, dollars per day. |
| DSK_COST | —Price per disk. |
| PAD_COST | —Price per pad. |

**Solution:**

1. Read DAY_RATE, DISK_COST and PAD_COST for this run.
2. For each customer,
   a. Read CUST_NAME, CUST_ID, DAYS, DISKS, and PADS.
   b. Echo the input (same as Step 2 in the previous version).
   c. Compute RENT = DAYS × DAY_RATE.
   d. Compute TTL_DISK_COST = DISKS × DSK_COST.
   e. Compute TTL_PAD_COST = PADS × PAD_COST.
   f. Compute TOTAL_COST = RENT + TTL_DISK_COST + TTL_PAD_COST.
   g. Apply the discount as appropriate (same as Step 7 in the previous version) and determine ADJ_COST.
   h. Print the customer's bill.

## 2.3 ALGORITHMS INTO PROGRAMS

Once a solution has been defined and we are convinced that it will work, we can begin the process of representing the algorithm as a program. The crucial objective in this step is to make sure that the program is a faithful expression of the algorithm. While it is completely unrealistic to suppose that this transformation is free of problems, we can take advantage of some design practices which help keep them to a minimum.

### 2.3.1 The Structured Program

Perhaps the most important single factor in producing an effective program is to make sure that the program is as simple and clear as possible. The design or selection of an algorithm for a particular task does not necessarily mean that only one type of program can be derived from it. Therefore, without some kind of disciplined, well organized approach to the programming process, the development of a program, as well as its reliability, can be rather haphazard. The computing literature is liberally sprinkled with accounts of programming disasters in which the need to introduce a relatively small change in a program caused large expensive programs to be scrapped and rewritten

because nobody (even the authors themselves) could analyze the program in sufficient detail to figure out how the change should be incorporated.

In recent years, extensive studies of program construction and behavior have produced insights and methodologies which make such catastrophes unnecessary. The result is a set of guidelines and techniques for the systematic expression of algorithms as programs. These are referred to collectively as *structured programming.*

The heart of the approach consists of a few simple organizational concepts which serve as the building blocks for all structured programs. An enormous amount of accumulated experience has made it clear that programs consisting of these elements are easier to analyze, test, and modify (when changes are required) than those constructed arbitrarily. Moreover, an important study has proved mathematically that these structured components are sufficient for writing any possible program. Accordingly, a major concern in structured programming is to make sure that a program, no matter how complicated, consists of nothing but these building blocks.

An important aid in designing structured programs is a clear, written description of the program. This is nothing more than a precise document that provides a road map from which the actual program will be prepared. If it contains the proper information, i.e., the specific activities the program is to perform and the sequence in which they are to be performed, it will serve equally well regardless of the language in which the program eventually is written.

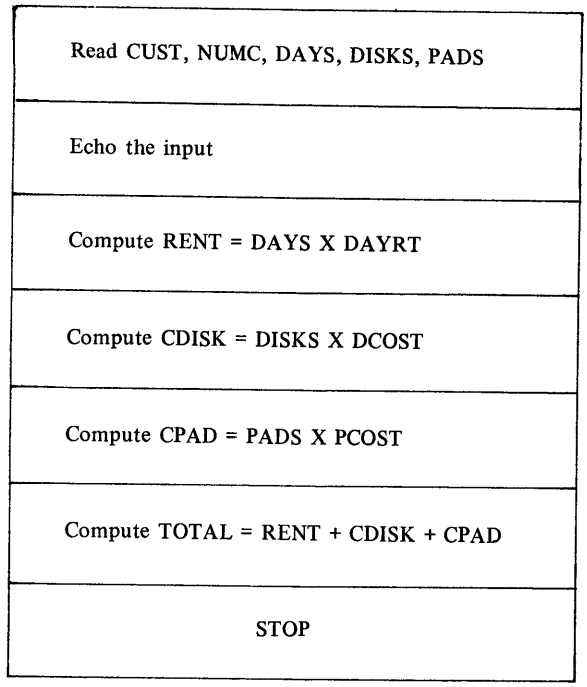
Since the program is to be structured, its description, too, is structured. Each component is shown as a standard structured building block, and the connections among these components give an overall picture of the algorithm's organization. Thus, the program description can be used to check whether a procedure does the required job *before* that procedure is *coded* (i.e., expressed as an actual program). Any required adjustments to the procedure are made at this stage, so that we base the actual coding process on a program description known (or strongly believed) to be correct.

Numerous ways have been devised for preparing program descriptions. We shall be using two methods, each of which is powerful, yet simple. One of these expresses a program description as a *structured flowchart* in which each building block is represented by a particular symbol. One type of structured flowchart is the *N-S diagram*, named for its originators Nassi and Schneiderman. A second method presents the program description in a more narrative form. The flow of events unfolds like a story, with each activity being expressed by an ordinary sentence. These activities are organized into standardized structural components by using specific vocabulary words to bracket the components. This type of representation is called *pseudocode*. The name is very descriptive because, in a sense, we are "writing" the program in story form. What we are not doing, though, is showing in exact detail how a given action is expressed in the particular programming language that will be used.

There is no standard form for pseudocode. Different computer installations have developed or adopted various guidelines for their pseudocodes. Differences among these practices are small enough so that the pseudocode used in this book will serve as an effective basis for any other set with which you will be working in the future.

**2.3.1.1 The Operational Sequence** The most basic type of occurrence in a program consists of a single sequence—a series of events that take place one after the other. The eight steps under Step 2 in the revised Example 2.1 (i.e., the individual customer processing), for instance, form such a sequence. They may be diagrammed as in Figure 2.1(a) or expressed in pseudocode as shown in Figure 2.1(b). Note that the activities themselves are described in simple English—the simpler the better. The *structural identity* of these activities as a sequence is indicated by two pseudocode terms:

1. *seq*—This indicates the beginning of the sequence. It implies that whenever the sequence is used, it starts from this point.




(a)

**Seq**
"Read customer i.d., customer no., no. of rental days, no. of disks used, no. of pads used."
"Echo the input."
"Compute the rental fee, disk costs, pad costs."
"Compute the total rental and purchase amounts."
**Endseq**

(b)

**FIGURE 2.1 (a)** An N-S Representation of a Sequence. **(b)** Pseudocode Representation of a Sequence.

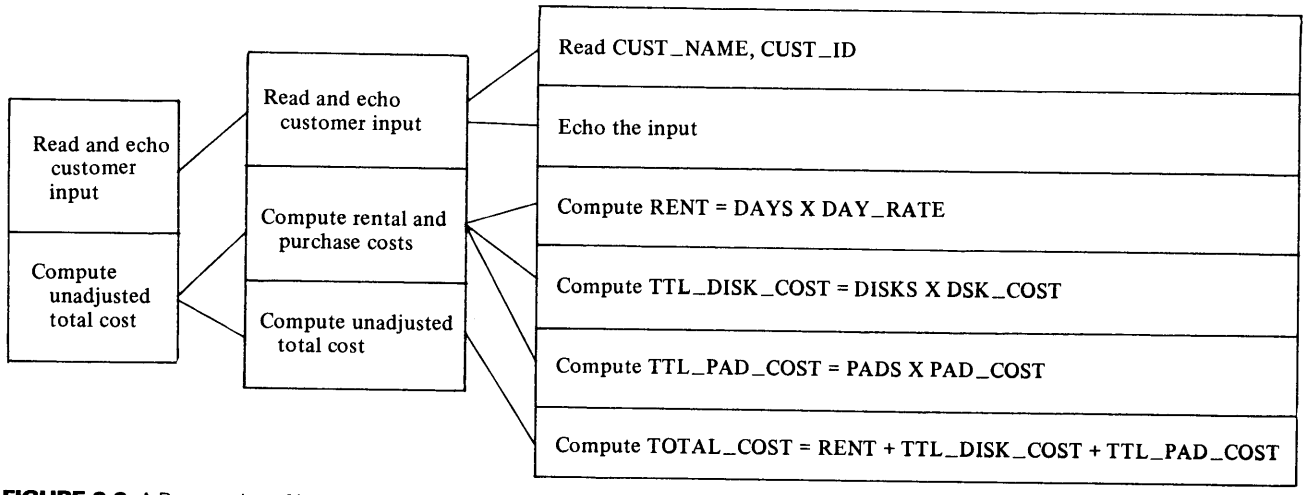



**FIGURE 2.2** A Progression of Increasingly Detailed Sequence Diagrams.

2. *endseq*—This indicates the conclusion of the sequence. Regardless of what happens inside the sequence, it always ends at this point.

If the sequence consists of a single activity, the *seq* and *endseq* may be omitted. More generally, if the sequential nature of the activity is obvious, then *seq* and *endseq* may be unnecessary in such cases as well.

Regardless of whether we use pseudocode or N-S diagrams, note that the idea of a sequence has nothing to do with its length or complexity. Whenever a sequence is used, it is used in its entirety, beginning with its single starting point and concluding with its single end point. In a sense, then, every program is a sequence.

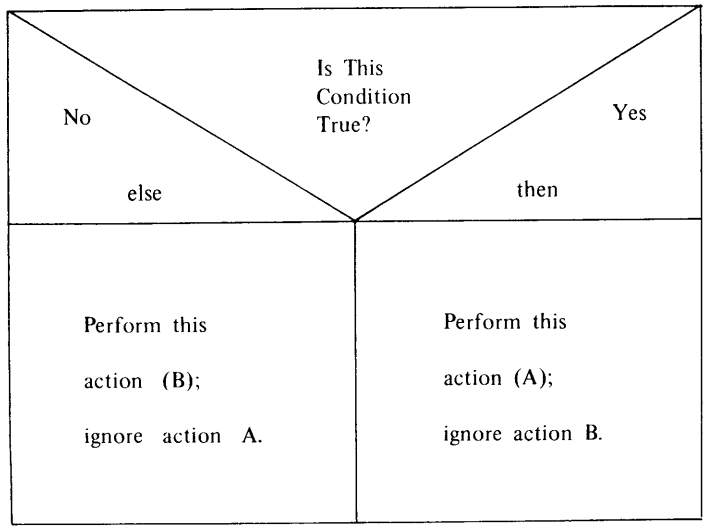
The amount of detail shown in a sequence diagram depends on that diagram's purpose. Figure 2.1, for example, has so much detail that itis actually a paraphrase of the programming steps to be produced from it. Consequently, it can be viewed as the final

version of a progression of diagrams, each one depicting the same sequence in more and more detail. Figure 2.2 illustrates such a progression ending with the level of detail in Figure 2.1. This may be overdone for a sequence as simple as the one in this example, but it does illustrate the general approach. As you gain experience with increasingly complicated problems, you will find it easy to judge the kind of progression needed for a given situation.

**2.3.1.2 Simple Selection (Alternation)** While the operational sequence takes care of many situations, it does not offer the flexibility required by many others. We often are faced with two possible actions where we must select one over the other. Of course, such occasions come up in everything we do. When they occur in a computer-related procedure, the operational sequence simply cannot handle them. However, there is a very effective use of the computer's powerful facilities which allows the construction of a test that selects one activity over another. This process represents a single conceptual activity that forms another basic building block in structured programs. This is called an *IF-THEN-ELSE construct.* The construct consists of a test that produces one of two possible outcomes. Each outcome is associated with a particular action. Once the outcome has been determined, the corresponding action is performed and the alternative action is ignored. The N-S and pseudocode representations of this construct are shown, respectively, in Figures 2.3(a) and 2.3(b). Here again, as in the sequence, we use special pseudocode terms to show the structural components:

1. *if*—This shows the beginning of the construct.
2. *then*—This signals the beginning of the activity to be performed when the test condition is true.
3. *else*—This indicates the beginning of the activity to be performed when the test condition is found to be false.
4. *endif*—This concludes the construct.

The test may be as simple or as complicated as it needs to be, as long as it is designed with "true" or "false" (or "yes" or "no," if that is more convenient) as the only two possible results. Similarly, the action performed in response to each outcome can be anything it needs to be, regardless of the number of steps it takes to achieve that action. The only real limitation is that we must be able to think of it as a single conceptual activity.

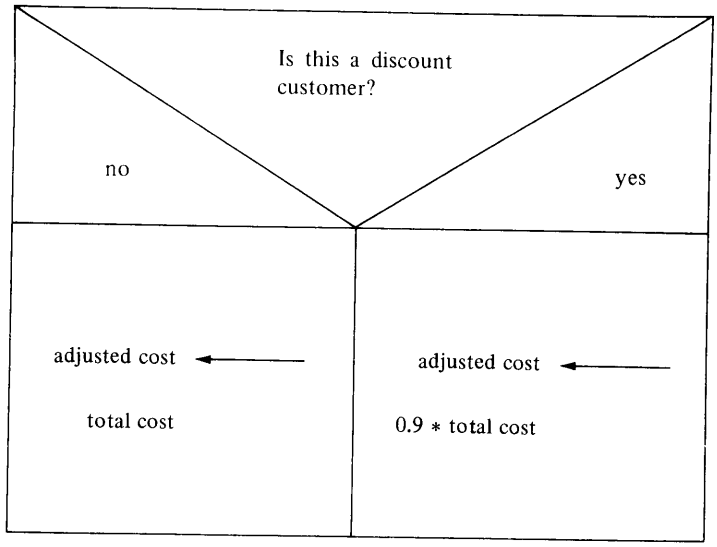



(a)

```
If
        "test condition"
    Then
        "perform this action"
    Else
        "perform this action"
Endif
```

(b)

**FIGURE 2.3 (a)** The IF-THEN-ELSE Construct: N-S Representation. **(b)** The IF—THEN-ELSE Construct: Pseudocode Representation.




(a)

**If**
"this is a discount customer"
**Then**
"deduct 10% from TOTAL"
**Else**
"Total remains unchanged"
**Endif**

(b)

**FIGURE 2.4** **(a)** Step (7) of Example 2.2 as an IF-THEN-ELSE Construct: N-S Representation. **(b)** Step (7) of Example 2.2 as an IF-THEN-ELSE Construct: Pseudocode Representation.

Here are some examples:

1. "Did the bonus money come?"
   "If yes, make reservations at the Elegante Restaurant."
   "If not, unwrap the bologna."
2. "Is the Gravity Indicator Reading above 3.86?"
   "If yes, compute critical thrust using Equation 15b with no correction."
   "If no, compute critical thrust using Equation 27c with Blum's orthogonal correction."

Step 7 in the individual customer processing of revised Example 2.2 is another case in point:

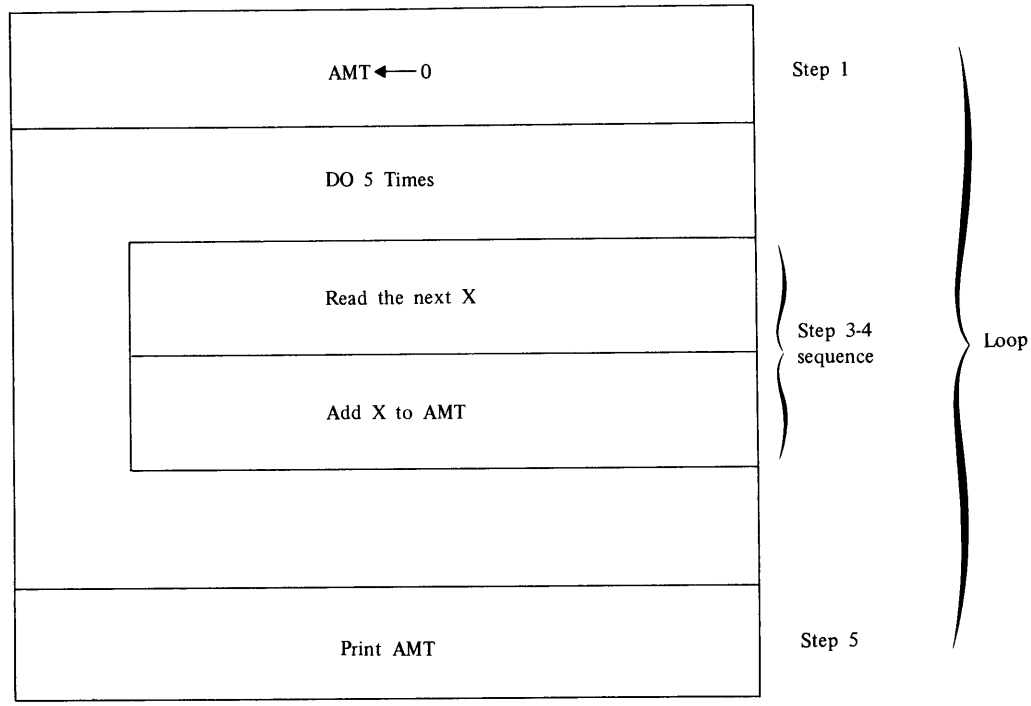
"Does this customer (the one currently being processed) get a discount?"
"If yes, apply the discount to the total charges;"
"If not, set the adjusted amount to the same value as TOTAL."

The representations for this decision structure are seen in Figure 2.4.

**2.3.1.3 Repetitive Action** A very powerful technique, fundamental to all programming, is the construction and use of *loops* for implementing activities which are to be performed repeatedly. The basis of this technique is to make it possible for a given sequence of instructions to be used over and over, therefore avoiding the need for a separate physical copy for each repetition of the activity. A simple example will illustrate:

**Example 2.3** Suppose we wanted to read five values of X and compute their sum (which we shall call AMT). We can specify this process as follows:

1. "Set AMT to an initial value of 0."
2. "Read the first value of X."
3. "Add X to AMT."
4. "Read the second value for X."
5. "Add X to AMT."
6. "Read the third value for X."
7. "Add X to AMT."
8. "Read the fourth value for X."




(a)

**Seq**
"Set AMT to zero."
**Do** 5 times
**Seq**
"Read the next X"
"Add X to AMT"
**Endseq**
"Print AMT"
**Enddo**
**Endseq**

Step 3–4 sequence

Loop

(b)

**Figure 2.5** **(a)** Repetitive Loop for Example 2.3: N-S Representation. **(b)** Repetitive Loop for Example 2.3: Pseudocode Representation.

9. "Add X to AMT."
10. "Read the fifth value for X."
11. "Add X to AMT."
12. "Print the value of AMT."
13. "Stop."

Obviously, this is a completely impractical way to do things. (Imagine having to do this for 200, 2000, or 2,000,000 values of X.) A much more reasonable approach is to set up a loop in which the activities are controlled by a mechanism that makes sure the loop repeats exactly as many times as it is supposed to. With a loop we can restate the previous process as follows:

1. "Set AMT to an initial value of zero."
2. "Perform steps 3 and 4 as a pair exactly five times."
3. "Read the next value of X."
4. "Add X to AMT."
5. "Print the value for AMT."
6. "Stop."

The N-S diagram (Figure 2.5(a)) shows the loop and its controlling mechanism, preceded and followed by

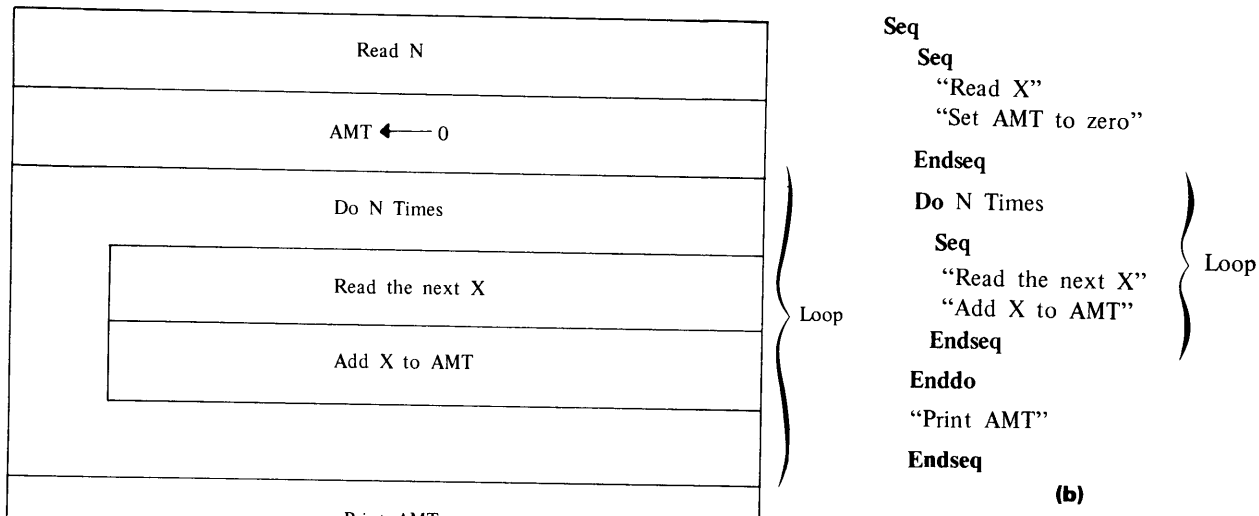



**Figure 2.6** **(a)** A Loop Repeated N Times: Structured N-S Representation. **(b)** A Loop Repeated N Times: Pseudocode Representation.

simple activities (one-step "sequences," to be precise). Pseudocode representation of this construct (Figure 2.5(b)) uses the following special terms:

1. *do*—This shows the beginning of the loop.
2. *enddo*—This shows the end of the loop.

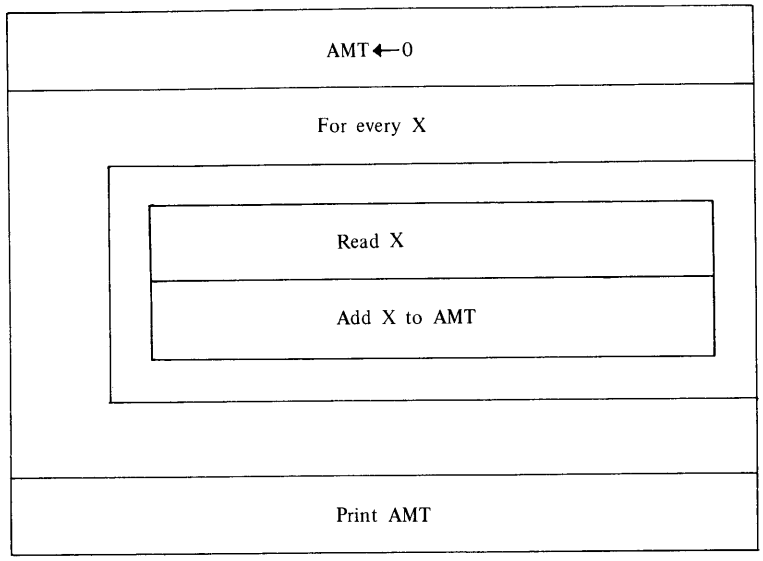
When necessary we shall add information to the pseudocode to describe the way in which the loop is controlled. In this type of loop, the control information simply defines the number of cycles.

The loop does not become any more complicated regardless of the number of times it needs to be repeated. We can generalize the previous loop by reading in N, the number of times we want the loop to repeat, thereby allowing that number to change each time the process is used:

1. "Read N, the number of X values available this run."
2. "Set AMT to an initial value of zero."
3. "Repeat Steps 4 and 5 exactly N times."
4. "Read the next value of X."
5. "Add X to AMT."
6. "Print the value for AMT."
7. "Stop."

The revised N-S diagram and pseudocode are given in Figure 2.6.

We may have different reasons for repeating a loop. We may want to run it until the result hits a given total. The decision to repeat the loop or not to repeat it is governed by some particular condition (other than the number of repetitions) which is tested every time the activity is performed. As long as that condition holds, the loop continues to cycle; as soon as that condition is violated, the loop is bypassed and the process continues just beyond it. We can see this by adjusting Example 2.3 again. This time, we do not know how many values of X there will be. However, we are told that X cannot be zero. Therefore, we can take advantage of that fact by submitting an input value of zero after the last X value and testing for it. Figure 2.7 shows one way of representing this loop. Note that we show that we wish to repeat a certain process for every input value. There is not enough detail to indicate how we shall control the repetition; we merely imply that some kind of control will be needed. The forms shown in Figure 2.8 describe the same loop, but the additional detail reveals how the control will work: Specifically, we shall "prime" the




**(a)**

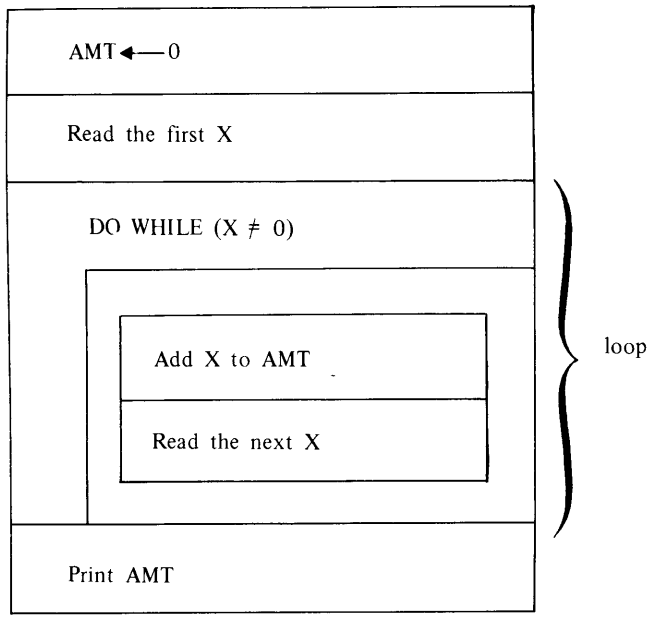
```
Seq
    "Set AMT to 0"
    Do for every X
      Seq
        "Read X"
        "Add X to AMT"
      Endseq
    Enddo
    "Print AMT"
Endseq
```

or

```
"Set AMT to 0"
Do for every X
    "Read X"
    "Add X to AMT"
Enddo
"Print AMT"
```

**(b)**

**FIGURE 2.7** **(a)** N-S Loop with an Unknown Number of Repetitions. **(b)** Pseudocode Representation of a Loop with an Unknown Number of Repetitions.

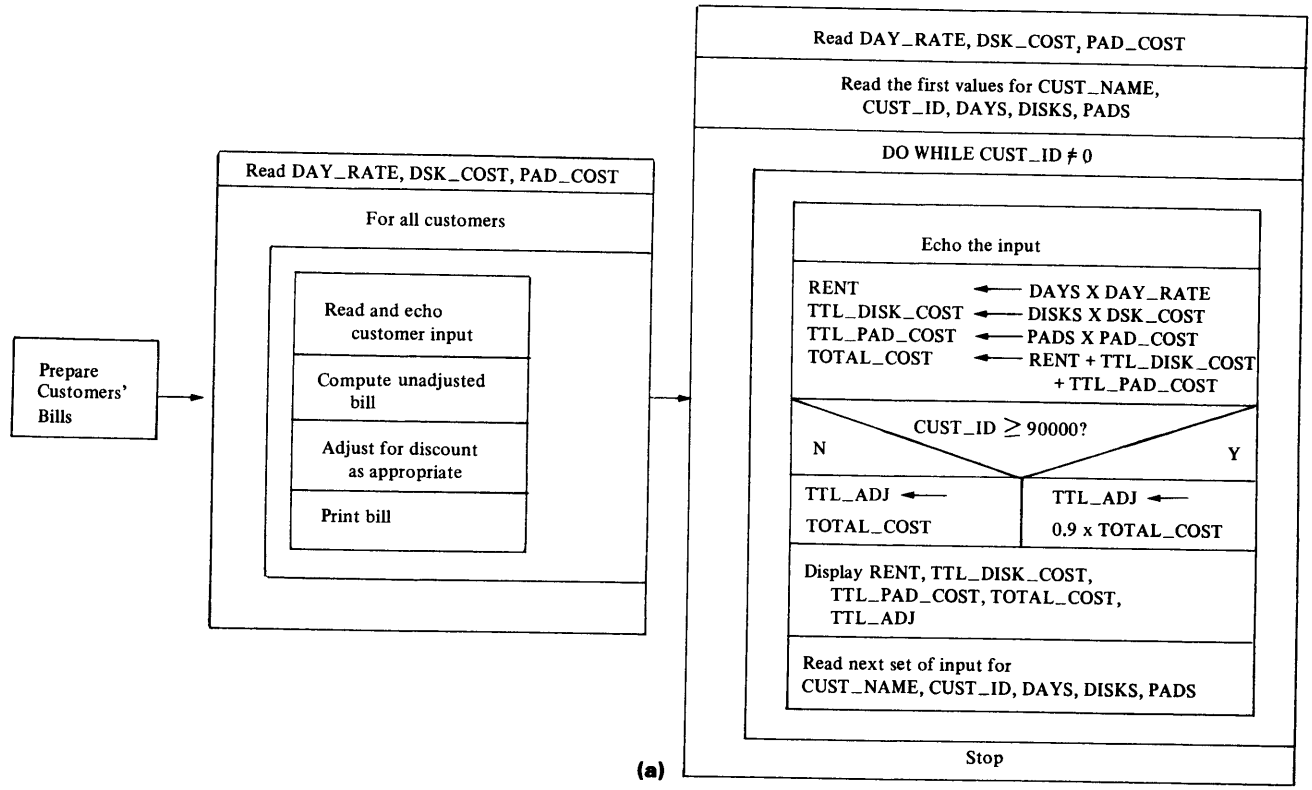



**(a)**

```
"Set AMT to zero"
"Read the first X"
While X ≠ 0
      "Add X to AMT"
      "Read the next X"
Endwhile
"Print AMT"
```

**(b)**

**FIGURE 2.8** **(a)** N-S Diagram for WHILE Construct. **(b)** Pseudocode Representation for WHILE Construct.




```
"Prepare
  Customers'
  Bills"
```

```
"Read DAY_RATE, DSK_COST,
    PAD_COST
Do for all customers
  "Read and echo
      customer input"
  "Compute unadjusted bill"
  "Adjust for discount
    as appropriate"
  "Print bill"
End do
```

```
"Read DAY_RATE, DSK_COST, PAD_COST
"Read the first CUST_NAME, CUST_ID,
    DAYS, DISKS, PADS"
While CUST_ID ≠ 0
  "Echo the input"
  "Set RENT to DAYS × DAY_RATE"
  "Set CDISK to DISKS × DSK_COST"
  "Set CPAD to PADS × PAD_COST"
  "Set TOTAL to RENT + TTL_DISK_COST + TTL_PAD_COST
If
    CUST_ID is equal to or greater than 90000"
    Then
        "set TTL_ADJ to 0.9 × TTL_COST"
    Else
        "set TTL_ADJ to TTL_COST"
  Endif
  "Print RENT, TTL_DISK_COST, TTL_PAD_COST,
    TOTAL_COST, TTL_ADJ"
  "Read next CUST_NAME, CUST_ID, DAYS,
    DISKS, PADS"
Endwhile
"Stop"
```

(b)

**FIGURE 2.9** **(a)** Program Design for Example 2.4: N-S Representation. **(b)** Program Design for Example 2.4.

the additional detail reveals how the control will work: Specifically, we shall "prime" the procedure by reading the first input value outside of the loop (i.e., as a separate action) and then go through a cycle in which testing, processing, and subsequent input are performed over and over. This type of component is called a *WHILE construct.* The pseudocode representation uses two special terms:

1. *while*—This shows the beginning of the loop.

2. *endwhile*—This shows the conclusion of the loop.

Note that the test to determine whether to repeat the loop is placed at the beginning of the loop. This means that it is possible to fail the test even before the loop's activity is attempted for the first time. As a result, the construct is very general. In the situation of Figure 2.7, for example, it takes care of a wide range of situations, including the one in which X's first value is zero and we never get to add anything to AMT.

In general, as an algorithm is analyzed and transformed into a program, it is likely that the level of detail seen in Figure 2.8 would evolve from the more abstract picture given in Figure 2.7.

**Example 2.4** Now we can return to our swashbuckling floor company (Example 2.1) and complete the design by showing the processing of each customer as one cycle in a loop that repeats as long as there are customers to process. To provide a detailed control mechanism, we shall assume that an account number of zero signals the fact that the last customer has been processed. A progression of N-S diagrams (Figure 2.9(a)) shows the procedure starting with a very general description and increasing in detail until there is a clear picture not only of what the program needs to do, but also of how it will be done. The same type of progression is developed in Figure 2.9(b) for the pseudocode representation.

**2.3.1.4 Another Complete FORTRAN 77 Program = Example 2.5** Figure 2.10 shows a FORTRAN program written from the description in Figure 2.9. Now that the fundamental principles of program construction have been introduced, we can look at a complete program in more detail than we did in the previous chapter and begin to build some more familiarity with the structure of the language.

Lines beginning with a C are used to separate parts of the program or to provide comments. These are not part of the actual instructions to the computer. The first four *statements* (the lines beginning with PROGRAM, REAL INTEGER, and CHARACTER) name the program and define the variables used by it: A REAL number can have a fractional part, an INTEGER cannot. A variable described as CHARACTER may consist of letters, numbers, blanks, and other symbols (like parentheses, commas, asterisks, etc.). Such information is called a *character string*. Thus, CUST is defined as a variable whose value may consist of a string of any 20 letters, digits, or other symbols. Note that the customer number, CUST_ID, is declared separately with a statement that starts with INTEGER*4. The other integer variables are declared as INTEGER*2. The 2 or the 4 control the amount of storage used to accommodate the integer values. (A 4 doubles the amount of storage used.) When 2 is used, this saves storage, but it limits the capacity of the variable. (In this example, the largest acceptable customer number would be less than 33000.) Since we are using a 9 in the first digit for testing purposes, this would exceed the capacity of an INTEGER*2 variable. Hence, the larger version is needed for CUST_ID.

This program is designed to be executed *interactively*. This means that its operation will involve a "conversation" between the user and the program. To promote the idea of dialogue, an interactive program sends messages to the user and acts on his or her responses. In a sense, the user is being asked to do something. Such a request is called a *prompt*. The PRINT statement (i.e., the fifth statement in Figure 2.10) does that in this example. When the program is run, the character string enclosed in apostrophes will be displayed on the user's terminal. A string, when specified this way, is called a *literal* string. It is displayed as is (without the apostrophes), blanks and all. After displaying the message, the program waits for the user to supply the requested values through his or her terminal. Once this information is transmitted (this occurs when the user presses the RETURN or ENTER key after typing the values), the program continues.

**FIGURE 2.10** FORTRAN 77 Program for Example 2.4. ▶

```
C ***************************************************************
C                        PROGRAM FOR EXAMPLE 2.4                *
C                                                               *
C    DAY_RATE:           RENTAL RATE, DOLLARS PER DAY           *
C    DSK_COST:           DISK COST, DOLLARS PER DISK            *
C    PAD_COST:           PAD COST, DOLLARS PER PAD              *
C    CUST_NAME:          CUSTOMER NAME, UP TO 20 LETTERS LONG   *
C    CUST_ID:            CUSTOMER I.D. NUMBER, 5 DIGITS LONG    *
C    DAYS:               NUMBER OF RENTAL DAYS                  *
C    DISKS:              NUMBER OF DISKS PURCHASED              *
C    PADS:               NUMBER OF PADS PURCHASED               *
C    RENT:               RENTAL COST                            *
C    TTL_DISK_COST:      COST OF DISKS PURCHASED                *
C    TTL_PAD_COST:       COST OF PADS PURCHASED                 *
C    TOTAL_COST:         TOTAL COST TO THE CUSTOMER             *
C    ADJ_COST:           ADJUSTED COST TO THE CUSTOMER          *
C    RECALL THAT THE ''IMPLICIT NONE'' DECLARATION CAUSES       *
C    FORTRAN TO PRODUCE A WARNING MESSAGE WHENEVER IT           *
C    FINDS A VARIABLE BEING USED WITHOUT A PRIOR EXPLICIT       *
C    DECLARATION. ONE OTHER POINT: NOTE THAT THE                *
C    DECLARATION OF THE REAL VARIABLES IS TOO LONG TO FIT       *
C    ON ONE LINE. CONSEQUENTLY, WE HAVE TO CONTINUE IT ON       *
C    THE NEXT LINE. THE DOLLAR SIGN IN COLUMN 6 OF THAT LINE    *
C    TELLS FORTRAN THAT THE MATERIAL ON THAT LINE IS A          *
C    CONTINUATION FROM THE PREVIOUS LINE.                       *
C ***************************************************************
      PROGRAM          EX204
      IMPLICIT         NONE
      REAL             DAY_RATE, DSK_COST, PAD_COST, RENT,
     *                 TTL_DISK_COST, TTL_CPAD_COST, TOTAL_COST, ADJ_COST
      INTEGER*4        CUST_ID
      INTEGER*2        DAYS, DISKS, PADS
      CHARACTER*20     CUST_NAME

      PRINT *, 'ENTER DAY__RATE, DSK_COST, AND PAD_COST'
      READ *, DAY_RATE, DSK_COST, PAD_COST
      PRINT *, 'ENTER CUST_NAME, CUST_ID, DAYS, DISKS, AND PADS'
      READ *, CUST_NAME, CUST_ID, DAYS, DISKS, PADS
C ***************************************************************
C                        HERE IS OUR DO-WHILE LOOP:             *
C    ENTRY TO THE LOOP IS CONTROLLED BY A TEST (BUILT INTO HP   *
C    FORTRAN 77'S DO WHILE STATEMENT) ON THE CURRENT            *
C    CUSTOMER NUMBER. AS LONG AS THAT NUMBER IS NOT EQUAL       *
C    TO (.NE.) ZERO, THE LOOP IS ENTERED AND ITS ACTIVITY IS    *
C    REPEATED ONCE MORE. THE CONCLUSION OF THE LOOP IS          *
C    MARKED BY THE STATEMENT END DO. AS SOON AS THE             *
C    CUSTOMER NUMBER CUST_ID IS EQUAL TO ZERO, THE              *
C    PROGRAM STOPS.                                             *
C ***************************************************************
      DO WHILE (CUST_ID .NE. 0)
          PRINT *, CUST_NAME, CUST_ID, DAYS, DISKS, PADS
          RENT = DAYS*DAY_RATE
          TTL_DISK_COST = DISKS*DSK_COST
          TTL_PAD_COST = PADS*PAD_COST
          TOTAL_COST = RENT + TTL_DISK_COST + TTL_PAD_COST
          IF (CUST_ID .GE. 90000) THEN
              ADJ_COST = 0.9*TOTAL_COST
          ELSE
              ADJ_COST = TOTAL_COST
          END IF
          PRINT *, RENT, TTL_DISK_COST, TTL_PAD_COST, TOTAL_COST,
         1     ADJ_COST
          PRINT *, 'ENTER CUST_NAME, CUST_ID, DAYS, DISKS, AND PADS'
          READ *, CUST_NAME, CUST_ID, DAYS, DISKS, PADS
          END DO

          PRINT *, 'END OF RUN.'
          STOP
          END
```

In the light of the foregoing discussion, the first two pair of PRINT and READ statements are self-explanatory. After a separating comment, the program specifies a loop beginning with the DO WHILE statement. As long as NUMC is not zero, the test passes and the specified activities are performed in sequence. Another test, part of the sequence, also is straightforward: If the account number is greater than or equal to (that is what the . GE. means in FORTRAN 77) 90000, this means (to us) that the customer is entitled to a discount. Accordingly, we compute ADJ, equal to 90% of the old (unadjusted) TOTAL. Then we go on to write (print) the results. On the other hand, if the test fails (i.e., the account number does not begin with a 9), the program skips the part following the word THEN and goes directly to the ELSE action. In either event, the PRINT statement is executed. The last statement in the loop sends the program right back to the first statement of the loop which checks the new set of input data for a customer number of zero.

### 2.3.2 Additional Structured Elements

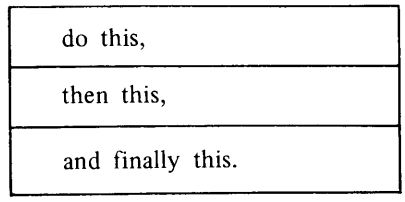
Structured programs make use of other building blocks which we shall discuss later. These are secondary because they can be put together from combinations of the basic constructs already examined. In that sense, they are conveniences rather than essentials, and we shall defer their use till other, more fundamental concepts have been considered.

## 2.4 SUMMARY

*Structured programming* is an approach to the development of computer programs in which there is a systematic progression from a clear statement of the problem to the actual coding of the program. An essential factor in this approach is the explicit delay of coding until an *algorithm* (a sequence of steps for solving the problem) has been defined, charted, and checked for correctness.

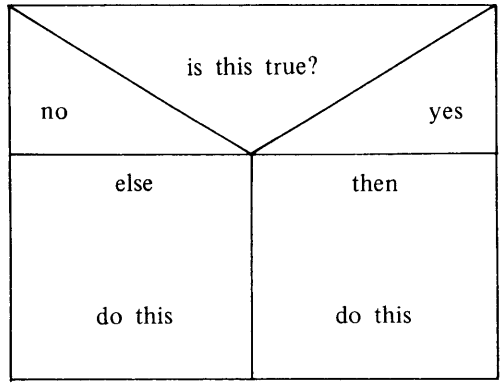
The result of this approach is a *structured program* built from a combination of basic *constructs* shown below. The ones on the left are expressed as *N-S charts* and the corresponding ones on the right are expressed as *pseudocode*, a paraphrase of programming steps in narrative form.

The Sequence:



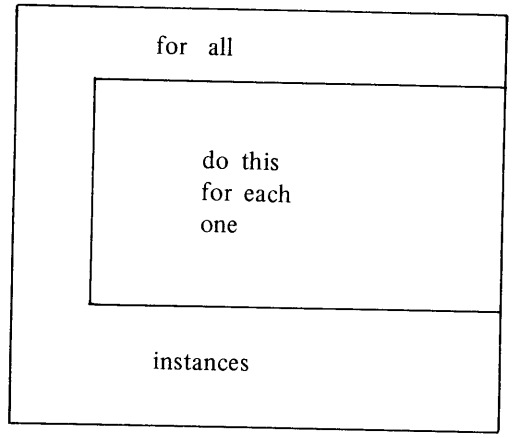

"**Do** this,"
"**Then** this,"
"**And** finally this."

Alternative Selection (IF-THEN-ELSE):




**if**
"This is true"
**then**
"Do this."
**else**
"Do this."
**endif**

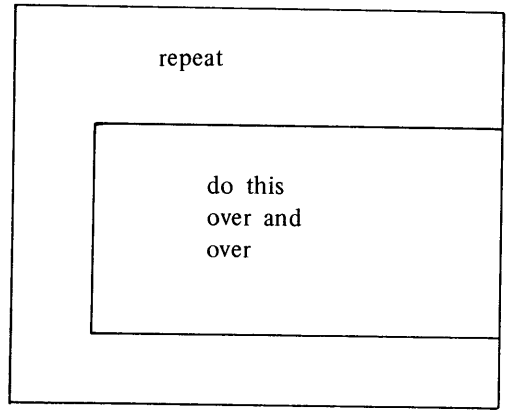
Repetitive Action (loop):




**do** for all instances

"Do this for each one."

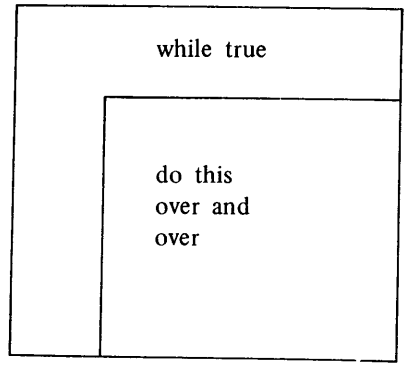
**enddo**




**do** N times

"Do this each time."

N times




**while** this is true

"Do this over and over."

**endwhile**

If a program is structured, it will consist only of these building blocks (or others built directly from these) and is likely to be easier to analyze, develop, and use.

**PROBLEMS**

1. The world of computers is not the only place in which people spend a lot of time and money to solve the wrong problems. There are many famous cases in just about every area of endeavor where exactly the same thing has happened. See if you can think of a few of these, or consult your library for information

about some of the more spectacular fiascos. The following general areas might provide fertile ground for inquiry:

(a) the automobile industry
(b) the movie industry
(c) the television industry
(d) the home photography industry
(e) the high fidelity industry
(f) the beverage industry
(g) the food processing industry
(h) the men's clothing industry
(i) the women's clothing industry
(j) the computer industry

2. Identify at least two different algorithms for solving each of the following problems:

(a) getting home from school or work
(b) preserving peaches
(c) drawing a straight line through a series of points plotted on a graph
(d) legally reducing the amount of income tax you pay
(e) making somebody laugh
(f) reducing the cost of transportation between where you live and where you work or go to school
(g) finding a particular book in a library
(h) finding a book on a particular subject in a library
(i) stopping the hiccups
(j) selecting a ripe cantaloupe

3. Think about each of the following everyday activities. Is the underlying process algorithmic? If it is, list the steps (in proper sequence) that describe the algorithm. If the process is not algorithmic, indicate why it is not:

(a) checking a book out of the library
(b) buying an insurance policy
(c) voting
(d) treating a cold
(e) mailing a letter
(f) buying and replacing a headlight for your automobile
(g) finding and eliminating the squeaky rattle in your automobile
(h) locating an old high school friend for the Big Reunion

The problems in the following group describe various processes or activities. For each one, identify an appropriate construct or constructs that can be used to represent the activity or process and express it either in N-S or pseudocode form:

4. Read three integers INT1, INT2 and INT3 and print them in reverse order.

5. Read three integers INT1, INT2 and INT3 and print them in the order they were read, followed by their sum.

6. A rectangular lot has length LENGTH and width WIDTH, both dimensions expressed in feet to the nearest tenth of a foot. LENGTH and WIDTH are to be read in from a line and used to compute PERIM, the perimeter of the lot, and AREA, the area of the lot. Print the input values followed by the computed results.

7. The rate at which a fluid travels through a pipe can be computed by the formula
WT_FLOW=DENSITY*AREA*VELOCITY
where
WT_FLOW is the rate of flow in kilograms per second
DENSITY is the fluid's density in kilograms per cubic meter
AREA is the pipe's area in square meters
VELOCITY is the fluid velocity in meters per second

Recall that the asterisk (*) is used in FORTRAN to denote multiplication. We are to read DENSITY, AREA, and VELOCITY, each in its proper units. These are to be printed, after which we are to compute WT_FLOW and print that value.

8. We are still pushing fluid through a round pipe as in Problem 5. However, this time the area is not directly available. Instead, input consists of DENSITY (as before), DIAM (the pipe's diameter in meters), and VELOCITY (as before). We are to print the input, after which the area is to be computed and printed. Once that is done, WT_FLOW is computed and printed.

9. Here we are again with the round pipe and the fluid. We still want to compute WT_FLOW, the rate of flow in kilograms per second. However, somebody new came along and was unaware of the ground rules. As a result, the value for the density is keypunched in pounds per cubic foot instead of kilograms per cubic meter, the pipe diameter is recorded in inches instead of meters, and the velocity is keypunched in miles per hour instead of meters per second. Oboy. Undaunted, we still want the same results (i.e., in the units specified in Problem 7). Accordingly, after reading the three values and printing them as *received*, we wish to convert and print the density in the desired units and the velocity, in its proper units. Then, we can go ahead and compute and print WT_FLOW as before. The following information will help in the conversions:

   There are 0.4536 kilograms in a pound
   There are 0.254 meters in an inch.

10. Window glass at the Looky Here Glasse Shoppe costs $6.80 per square foot. (Believe me, this is very special window glass.) Moreover, it is sold only in rectangular shapes. The length (LENGTH) and width (WIDTH), each in inches to the nearest sixteenth of an inch, are to be read in and printed. Then, we are to compute and print the area (AREA) in square feet, the cost of the glass (COST), the tax (TAX), and the total price (TOTAL). Tax in Looky's state is four and one half percent of the cost.

11. Read 36 values of X and print their sum (SUM)_X. (Remember (from Example 2.2) that the sum (SUM_X in this case) must be initialized to zero so that we have a known value to which we can begin adding when the first X is read.)

12. Compute and print the sum of the first 12 positive integers.

13. Read a positive integer INTGR and, for each integer from 1 through INTGR, print the integer (call it NUM), its square (call it NUMSQR), its cube (call it NMCUBE), and its square root (call it ROOTNM).

14. Read a positive integer INTGR and print it. Then, compute and print the sum (INTGR_SUM) of the first INTGR positive integers.

15. Read two positive integers INT1 and INT2 (INT1 < INT2), print them, and then compute and print POSITIVE_SUM, the sum of the integers from INT1 through INT2.

16. Redesign the procedure for Problem 14 so that it will repeat its operations for a succession of INTGR values. Stop the run with a negative value of INTGR.

17. Redesign the procedure for Problem 15 so that it will repeat for any number of sets of INT1 and INT2. Stop the run with a negative value of INT1.

18. Revise the procedure in Problem 6 so that it will read and process 18 sets of LENGTH and WIDTH values.

19. Revise the procedure in Problem 6 so that it will read and process any number of sets of LENGTH and WIDTH values, stopping only when a LENGTH of zero is read.

20. Revise the procedure in Problem 19 so that it uses a counter LOTS to keep track of how many lots it has processed in a given run, a variable named TOTAL_AREA for the total area of the lots, and a variable AVG_AREA for the average area. After the last lot has been processed (use the same signal as in Problem 19), print the final results (i.e., LOTS, TOTAL_AREA, and AVG_AREA).

21. Read two integers INT1 and INT2 and print the smaller one followed by the larger one.

22. Read two integers INT1 and INT2. This time, there is no guarantee that they will have different values. If they are different, print the smaller one followed by the larger one. If they have the same value, print the message "INT1 AND INT2 HAVE THE SAME VALUE", followed by the value.

23. Do Problem 15 with the following change: Instead of guaranteeing that INT1 < INT2, we can only guarantee that INT1 and INT2 will have different positive values.

24. Repeat Problem 15 with the following change: This time we can only guarantee that INT1 and INT2 will be positive.

25. Revise Problem 7 to take the following into account: When pushing a fluid through a small pipe, certain factors act to reduce the amount of the pipe's area that actually is available for flow. The pipe is said to have an *effective area* that is some fraction COEF of the physical area. If EFFECTIVE_AREA is the effective area, its value, then, may be computed as

    EFFECTIVE_AREA = COEF*AREA

    Then, of course, it is the effective area that is used, along with density and velocity, to compute the flow. As a rough rule of thumb, pipes with diameters below 0.0175 meters need correction with a COEF value of 0.794 while pipe diameters at or above that value need no correction. Print the area (AREA or EFFECTIVE_AREA) used to compute the flow.

26. Revise the procedure either in Problem 7, 8 or 25 so that it processes 20 sets of input values.

27. Revise the procedure either in Problem 7, 8 or 25 so that the first input value read and printed is SETS, the number of input sets to be processed during that run. Then it goes ahead and processes that many input sets.

28. Looky Glasse Shoppe has expanded its line so that it now offers frosted glass at $7.25 per square foot in addition to its regular (transparent) type at $6.80. (Prosperity has set in since Problem 10; at these prices, that is no surprise.) Accordingly. we need to modify the procedure in Problem 10 to accommodate the expansion. Input now consists of LENGTH and WIDTH as before, and an indicator named TYPE whose value is 1 for transparent glass and 2 for frosted. Output requirements still are the same as before.

29. Modify Problem 28 so that the process will handle any number of pieces of glass consisting of any mixture of the two types. A LTH value of zero signals the end of the run. After the last set of input has been processed, the revised procedure is to print the number of transparent pieces (NUM_TRANSPARENT), the total transparent square footage (TRANSPARENT_AREA), the total transparent revenue including tax (TRANSP_PRICE), the number of frosty pieces processed (FROST_NUM), their total area (FROSTY_AREA), the frosty revenue (FROST_PRICE), the total number of pieces for the run (TOTAL_PIECES), total area (TOTAL_AREA), and total revenue (TOTAL_PRICE).

30. In Problem 29 there is the implied assumption that all the input data are keypunched correctly. To induce you to think about the kind of algorithms that need to be designed when this assumption cannot be made, assume that LENGTH and WIDTH in Problem 29 always will be correct. Moreover, you can assume that when the type of glass is recorded as 1 or 2, it is the right type for that input set. However, there is no guarantee that the type always will appear as a 1 or 2. That is, this version of the process is to expect and handle cases where the type is specified in error. (For example, somebody might enter a 4 for the type, and there is no type 4.)

31. Design a procedure to process a set of input values as follows: Each input value is an integer NTG. If NTG is odd, add it to SUM_ODD; if it is even, add it to SUM_EVN. Keep reading values of NTG and adding them to the proper sum as long as NTG is positive. As soon as NTG is not positive, print SUM_ODD, SUM_EVN, and NUM_ODD and NUM_EVN, the number of odd and even values, respectively.

32. Here is a FORTRAN 77 program:

```
C ************************************************************************
C                    VARIABLE NAMES AND THEIR MEANINGS:                  *
C      LENGTH:   LENGTH;                WIDTH:    WIDTH;                 *
C      HT:       HEIGHT;                FRONT:    LENGTH*HEIGHT;         *
C      TOP:      LENGTH*WIDTH;          SIDE:     HT*WIDTH;              *
C      VOL:      LENGTH*WDTH*HT;                                         *
C ************************************************************************

      PROGRAM          PROB32
      IMPLICIT         NONE
      REAL             LTH, WDTH, HT, FRONT, TOP, SIDE, VOL
      INTEGER*2        NUMPCS

C ************************************************************************
C      THE ''IMPLICIT NONE'' DECLARATION IMMEDIATELY FOLLOWING THE      *
C      PROGRAM STATEMENT SETS A SPECIAL FLAG IN THE FORTRAN COMPILER     *
C      THAT CAUSES A WARNING TO BE PRODUCED WHENEVER A VARIABLE IS       *
C      NOT EXPLICITLY DECLARED > THIS HELPS KEEP US ALERT.               *
C ************************************************************************
```

```
NUMPCS = 0

PRINT *, 'ENTER VALUES FOR LTH, WDTH, HT'
READ *, LTH, WDTH, HT
DO WHILE (LTH .NE. 0.0)
      NUMPCS = NUMPCS + 1
      PRINT *, LTH, WDTH, HT
      FRONT = LTH*HT
      TOP = LTH*WDTH
      SIDE = WDTH*HT
      VOL = LTH*WDTH*HT
      PRINT *, FRONT, TOP, SIDE
      PRINT *, VOL
      PRINT *, 'ENTER VALUES FOR LTH, WDTH, HT'
      READ *, LTH, WDTH, HT
END DO

PRINT *, 'NO. OF PIECES: ', NUMPCS
PRINT *, 'END OF RUN. '
STOP
END
```

(a) Describe, in three sentences or less, what this program does.
(b) Indicate the constructs used by this program and show where each one begins and ends.
(c) What is NUMPCS?
(d) Indicate which lines are comments and which lines are statements.
(e) How many lines of printout does the program produce for each set of LENGTH, WIDTH and HT read in?
(f) In Example 2.4, there are two separate statements that perform the same READ operation on the same type of data—one outside the loop (before the loop starts) and one inside the loop. The program in this problem has only a single READ statement for the data, and that one is inside the loop. Discuss the *structural* differences between these two types of organization.

# 3
# *Getting Acquainted with FORTRAN 77*

The FORTRAN program shown in the previous chapter touches on a number of powerful features that enable the programmer systematically and conveniently to describe what he or she wants done. In this chapter we expand on those features to present a more comprehensive view of the language, along with an introduction to its major structural rules.

## 3.1 OVERALL ORGANIZATION OF A PROGRAM

Every FORTRAN program, regardless of its size or complexity, consists of a sequence of *statements* with each statement specifying a particular activity. The activity defined in a statement is not limited in any particular way by size restrictions on the statement. Basically, a statement can be up to 20 lines long. (Strictly speaking, a FORTRAN statement cannot exceed 1320 characters in total length; however, that is well beyond the requirements for most statements.) Accordingly, the programmer can concentrate on the problem to be solved rather than the peculiarities of the programming language.

### 3.1.1 The Basic Form of a Program

Figure 3.1 shows the general framework for a simple FORTRAN program. It starts with a PROGRAM statement and concludes with an END statement. Making such a program longer does not make it organizationally less simple; the basic framework is the same regardless of the length.

More elaborate program organizations are built by putting together several of these building blocks. This is illustrated in Figure 3.2. The framework outlined in Figure 3.2 consists of two procedural components. The first one, known as a *main program*, controls the use of the second one, known as a *subprogram*. Figure 3.3 shows this type of construction carried a bit further: This program consists of a main program followed by three subprograms. It is possible to build a large and intricate program by organizing it as a collection of subprograms which are controlled (directly or indirectly) by a single main program. Each subprogram can be designed to provide processing whose details are hidden from the main program. As far as the main program is concerned, the computations and other activities performed by a subprogram can be treated as a single step. Until more familiarity is developed with program construction, we shall concentrate on the basic type of organization shown in Figure 3.1, i.e., the single (main) program. However, if we know the direction in which program organization can expand, it will simplify the actual expansion when we get to it.

### 3.1.2 Major Statement Types

A statement in FORTRAN is similar to a sentence in a natural language: It is a unit of communication expressing something that we think of as a single activity or idea. This is

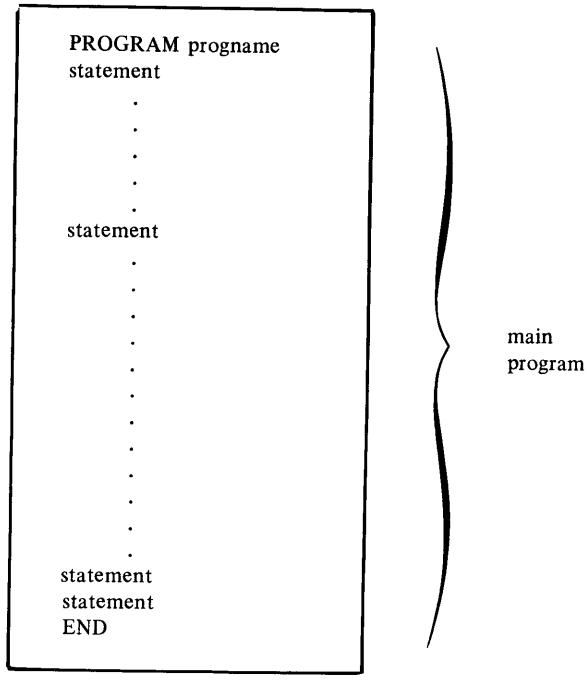



**FIGURE 3.1** General Structure of a Simple FORTRAN Program.

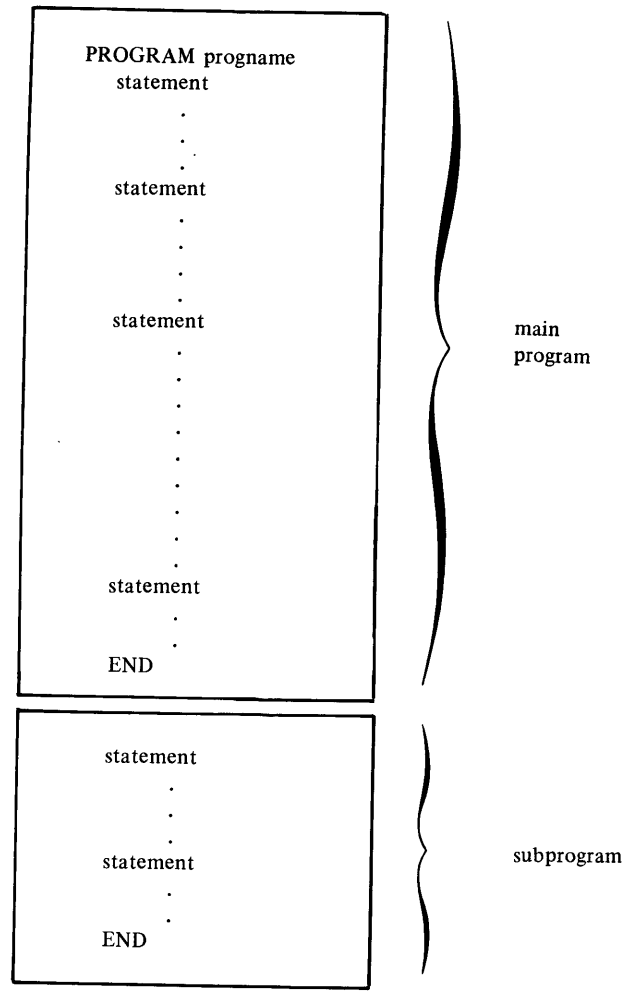



**FIGURE 3.2** A FORTRAN Program Consisting of a Main Program and One Subprogram.

true regardless of the number of separate things the computer has to do to fulfill the directives in a statement. Since it obviously is impossible to anticipate what the steps may be for a particular program, the language is designed to be extremely flexible. As a result, the things we want the computer to do can be expressed easily and conveniently with a surprisingly small number of statement types. This section will introduce each of these briefly and illustrate its use.

**3.1.2.1 PROGRAM and END Statements** A FORTRAN main program begins with a statement that says

PROGRAM *progname*

where *progname* is the name by which that main program is to be identified. When a program is completed and the computer center wants to make it available for general use, that program will be installed in a library. Then, anyone wishing to make reference to that program would do so through its name. In this latter connection, HP FORTRAN 77 accepts a number of additional specifications as part of the PROGRAM statement.

Every FORTRAN main program or subprogram concludes with a statement that says, simply,

END

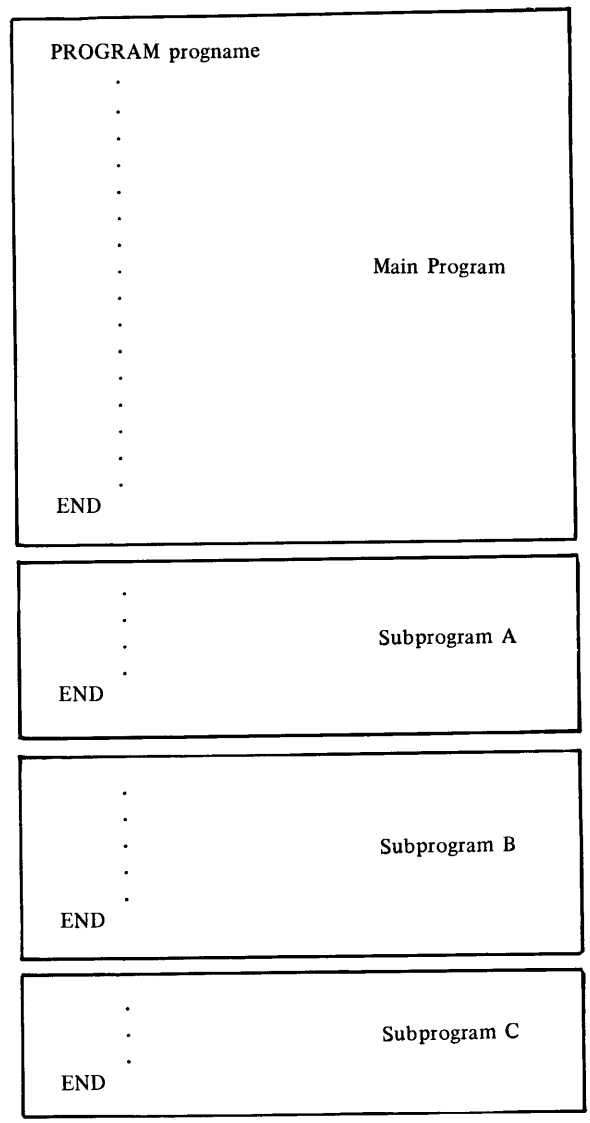



**FIGURE 3.3** A FORTRAN Program Consisting of a Main Program and Three Subprograms.

This tells the FORTRAN compiler (the program that translates FORTRAN statements to machine instructions) that it has reached the last physical statement in that program or subprogram.

**3.1.2.2 Declaration Statements** If a computer is to process information the way our program tells it to, that information must be available in the machine's memory, and its type must be defined. Moreover, there must be a way for the computer to get at the right information when it is needed. This is done by defining a name and a data type for each item of information used by the program. The programmer provides the name and data type, and FORTRAN provides the (invisible) bookkeeping that automatically makes sure that the right name goes with the right location and data type.

FORTRAN handles the definition of these names by means of *declaration statements* (also known as *specification statements*). A couple of examples will shed further light: The statement

```
INTEGER NUMOBS
```

operates as if we were to give the following instruction:

> "Dear FORTRAN: Reserve a place in storage and attach the name NUMOBS to it. I am going to use this location to store a positive or negative whole number whose value may change from time to time throughout the program. Make sure that you have enough information so that every time I use the name NUMOBS you refer to that same location. Thank you."

Another type of declaration looks like this:

```
CHARACTER*8 NXTWRD
```

This reserves enough space (and associates it with the name NXTWRD) to hold any eight characters. No particular set of characters is placed in NXTWRD as a result of this declaration, but the memory space now is available for such placement by other statements in the program.

As we saw in the last chapter, a single declaration statement may be used to define more than one variable. It is a good programming practice to place all of the declarations at the beginning of the program. This provides a clear record of all the variables used in that program. Furthermore, all of the definitions are in one place where we can refer to them conveniently. The next chapter will examine these declarations more fully.

**3.1.2.3 Assignment Statements** The most direct way of providing a value for a variable is by *assignment*, and the type of statement that performs this activity is the *assignment statement*. This statement looks like an equation in algebra but it really is not. To illustrate, let us look at a very simple but complete program:

**Example 3.1**

```
PROGRAM    EX301
IMPLICIT    NONE
INTEGER*2    WIDTH
WIDTH=32
END
```

The first statement defines the *integer variable* WIDTH. The second statement assigns a value of 32 to this variable. Despite its appearance, this statement does not say "WIDTH is equal to 32." Instead, it says: "Place a value of 32 in the location carrying the name WIDTH." This distinction is important, and we shall emphasize it by referring to the familiar symbol = as the *assignment operator.*

The assignment statement can become much more complicated since it is the way to specify any computations that we might want done. To illustrate, we shall lengthen our program a bit:

**Example 3.2**

```
PROGRAM    EX302
IMPLICIT    NONE
INTEGER*2    LENGTH, WIDTH, PERIM
LENGTH=247
WIDTH=32
PERIM=2*(LENGTH+WIDTH)
STOP
END
```

After defining the three variables, values of 247 and 32 are assigned, respectively, to LENGTH and WIDTH. The third assignment statement says, in effect: "Do the indicated arithmetic and assign the resulting value to PERIM." Accordingly, the value currently in LENGTH (247) is copied and added to a copy of WIDTH's current value (32). The sum thus obtained (279) is multiplied by 2 and the resulting product (558) is placed in (i.e., assigned to) PERIM. LENGTH and WIDTH still retain their previous values.

**3.1.2.4 Input/Output Statements** FORTRAN 77 handles all data transmission from the outside world by means of the READ statement. Similarly, delivery of information from the processor to the outside world is specified by means of the WRITE or PRINT statement. The statements used in Chapter 2 (Example 2.4) represent a basic form for specifying reading or writing called *list-directed* input/output. The name stems from the fact that the input or output appears as a list of values clearly separated from each other. In other types of input or output, the appearance of the data is more complicated so that the values must be supplemented by a description. We shall use list-directed input/output for a while.

**Example 3.3** As an example, consider the following program:

```
PROGRAM    EX03
IMPLICIT    NONE
INTEGER*4    LENGTH, WIDTH, PERIM, AREA
CHARACTER*8    NAME
PRINT *, 'ENTER VALUES FOR NAME, LENGTH, AND WIDTH'
READ *, NAME, LENGTH, WIDTH
AREA = LENGTH*WIDTH
PERIM = 2* (LENGTH + WIDTH)
PRINT *, NAME, LENGTH, WIDTH, PERIM, AREA
PRINT *, 'END OF RUN. '
END
```

Note that the READ statement indicates how many data items are to be read and where the values are to be stored. Specifically, it is the same as saying: "Read the next three pieces of data and store them in the variables named NAME, LENGTH, and WIDTH." The statement tells us where the data came from. Specifically, the 1 inside the parentheses serves as a signal to the READ statement to use a standard source of input. That source is defined inside FORTRAN and set at each computer installation. The device selected for this purpose is the one most likely to be used for input at that installation. HP FORTRAN 77 normally assumes your terminal to be the standard input device.

The READ statement in Example 3.3 does not say anything about the appearance of the data. The programmer does not have to supply this information because the list-directed READ statement expects the input data to be in a standard form: The values simply are entered on the terminal keyboard, separated from each other by one or more blanks. Commas also may be used to separate list-directed input values. Whichever form is used, it is up to the programmer to make sure that the order in which the variable names are listed in the READ statement matches that of the data values on the line.

The list-directed PRINT statement operates in the same general way. Output is sent to a predefined device. This is usually a printer or terminal, and we shall assume the latter here. The output values themselves are printed (displayed) in a standard format defined within FORTRAN. ("Print" and "display" will be used interchangeably in discussions of output.) We can begin to see how this works by looking at Figure 3.4, which shows a set of input data for the program in Example 3.3. The three values on the line correspond to the three names listed in the READ statement, so that after the statement is executed (that is, just before AREA is computed), the following will be true:

1. NAME will contain the eight characters OAKHURST. The apostrophes are part of the standard format. They do not get stored.
2. LENGTH now has a value of 424.
3. WIDTH now has a value of 276.
4. Values for AREA and PERIM still are undetermined, since the computations have not been done yet.

```
'OAKHURST'    424    276
```

**FIGURE 3.4** Sample Input for Example 3.3.

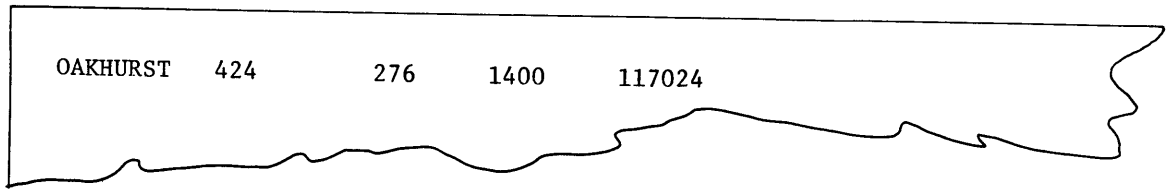



**FIGURE 3.5** Sample Output for Example 3.3 (Produced from Figure 3.4's input).

Exactly the same thing would happen if the input data were to look like this:

```
'OAKHURST', 424, 276
```

After the computations for AREA and PERIM are completed and the PRINT statement is executed, the resulting output will appear as shown in Figure 3.5. The order in which the five items are printed corresponds to that given in the PRINT statement and has nothing to do with anything else in the program. Thus, the programmer can specify any order that suits his or her purpose. The only note of caution is that there must be a value available for each variable name listed in the PRINT statement. Table 3.1 summarizes the values available in the program of Example 3.3 as each of its steps is completed. The program is shown in part (a) of the table, where we have attached a sequential number to each statement for convenience. We see (part (b)) that values for NAME, LENGTH and WIDTH are not known until they are actually read in by statement 5. Similarly, once the program has run, the memory used for the program and data is reassigned for some other purpose and the variables are no longer recognized. Of course, it no longer matters because we have what we want, having printed it in statement 7.

Complete control over input/output processes is exercised through the *formatted* READ, WRITE and PRINT statements. These are described later on, in Chapter 15.

**Table 3.1** Execution of the Program in Example 3.1

| *Statement No.* | *Statement* |
|---|---|
| 1 | PROGRAM EX03 |
| 2 | IMPLICIT NONE |
| 3 | INTEGER*2 LENGTH, WIDTH, PERIM, AREA |
| 4 | CHARACTER*8 NAME |
| 5 | PRINT *, 'ENTER VALUES FOR NAME, LENGTH, WIDTH' |
| 6 | READ *, NAME, LENGTH, WIDTH |
| 7 | AREA = LENGTH * WIDTH |
| 8 | PERIM = 2 * (LENGTH+WIDTH) |
| 9 | PRINT *, NAME, LENGTH, WIDTH, PERIM, AREA |
| 10 | END |

(a)

| *Reference Point* | *Name* | *Length* | *Width* | *Perim* | *Area* |
|---|---|---|---|---|---|
| Just prior to Statement 6 | ? | ? | ? | ? | ? |
| Just prior to Statement 7 | OAKHURST | 424 | 276 | ? | ? |
| Just prior to Statement 8 | OAKHURST | 424 | 276 | ? | 117024 |
| Just prior to Statement 9 | OAKHURST | 424 | 276 | 1400 | 117024 |
| Just prior to Statement 10 | OAKHURST | 424 | 276 | 1400 | 117024 |
| Just after Statement 10 | ? | ? | ? | ? | ? |

(b)

**3.1.2.5 Control Statements** When a program runs, we think of its action as a series of events occurring in a certain sequence, one after the other. In a sense, the program is traveling on a journey toward its conclusion. For a very simple program, like the one in Example 3.3, the journey follows a single pathway until the road ends. Every time such a program runs, it makes the same trip in the same way. When the problem being solved is more complicated, it may be necessary to design a program with numerous pathways having different lengths, leading in different directions, and ending in different places. Under these circumstances, an appropriate journey will be selected each time the program runs. Control statements provide the programmer with a way of defining the selection process and how it is to take place.

In Chapter 2 we dealt with two basic ways to select the path taken through a program. One of these, the IF-THEN-ELSE construction, enables us to choose one of two available pathways depending on some condition that we can test. Once the path is chosen, the program moves forward.

This construction is represented directly in FORTRAN 77 by the *IF-block*:

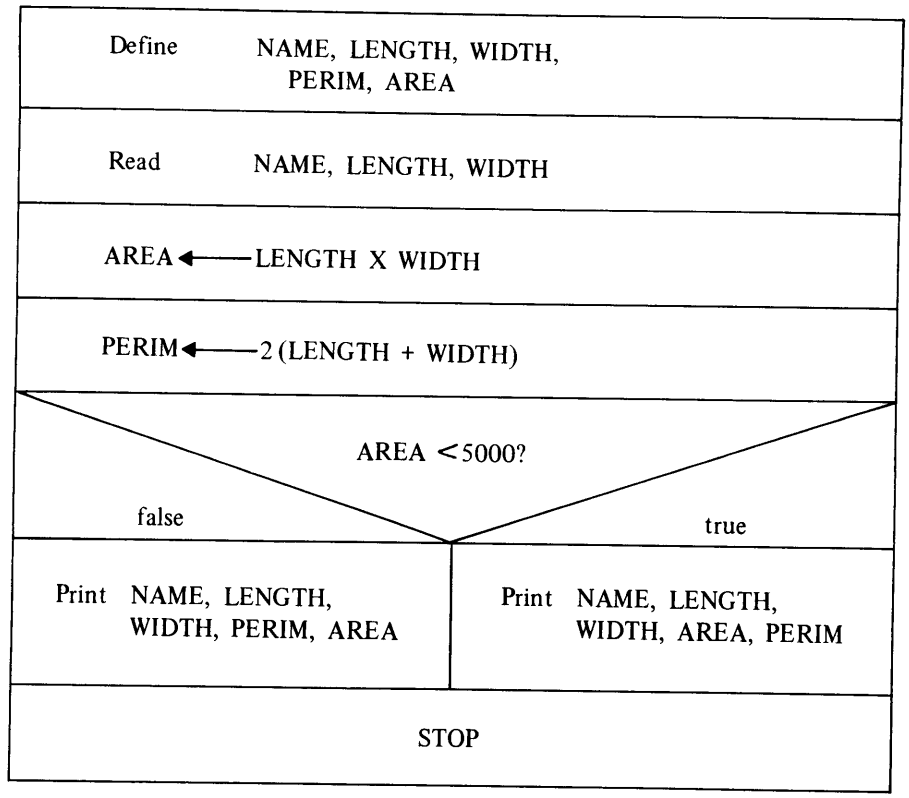
```
IF test condition THEN
    action1
ELSE
    action2
END IF
```

The two alternative paths are described by *action1* and *action2*, respectively. Only one is chosen, depending on the outcome of the test: *action1* is taken when the outcome is "true" (i.e., the answer to the test question is "yes"), and *action2* is followed instead when the outcome is "false."

**Example 3.3A** We can see how this operates in a specific program by changing the processing in Example 3.3. This time, instead of printing the output values in one fixed sequence, we shall provide two choices: if the area is less than 5000, the program is to print the name, length, width, perimeter, and area, in that order. On the other hand, if the area is not less than 5000, the program is to print the name, length, width, area and perimeter, in *that* order. The pseudocode and N-S diagram for this revision are shown in Figure 3.6, and the program itself appears below. Note that the .LT. that appears in the IF statement is the way the "less than" comparison is expressed in FORTRAN. We shall deal more extensively with these types of tests as we go on.

```
PROGRAM              EX03A
IMPLICIT               NONE
INTEGER*4              LENGTH, WIDTH, PERIM, AREA
CHARACTER*8          NAME
PRINT *, 'ENTER VALUES FOR NAME, LENGTH, AND WIDTH'
READ *, NAME, LENGTH, WIDTH
AREA = LENGTH*WIDTH
PERIM = 2* (LENGTH + WIDTH)
IF (AREA .LT. 5000) THEN
    PRINT *, NAME, LENGTH, WIDTH, PERIM, AREA
ELSE
    PRINT *, NAME, LENGTH, WIDTH, AREA, PERIM
END IF
PRINT *, 'END OF RUN. '
END
```

The loop allows the programmer to set up a design in which the program can interrupt its "forward" progress through the statements to double back on itself and repeat part of the journey. Our brief experience with such loops (Examples 2.3 and 2.4) has already illustrated that we can control the action in various ways based on two fundamental techniques:




**(a)**

```
"Define NAME, LENGTH, WIDTH, PERIM, AREA."
"Read NAME, LENGTH, WIDTH."
"Compute AREA = LENGTH x WIDTH."
"Compute PERIM = 2 (LENGTH + WIDTH)"
If "Area is less than 5000" then
   "Print NAME, LENGTH, WIDTH, PERIM, and AREA."
else
   "Print NAME, LENGTH, WIDTH, AREA, and PERIM."
End if
   "Stop."
```

**(b)**

**FIGURE 3.6** **(a)** N-S Representation of Example 3.3A. **(b)** Pseudocode Representation for Example 3.3A.

1. We can set out to repeat a loop a certain number of times. When the specified number of repetitions has taken place, the program breaks out of that pattern and resumes its march toward journey's end.
2. We can set up some kind of signal whose status ("on" or "off") governs the program's conduct. While the signal is on, the program repeats the loop. It continues to do so as long as the signal stays on regardless of the number of repetitions. As soon as the signal goes off, the pattern is broken and the journey's basic direction resumes. It may be that, for a particular run, the signal never gets turned on to begin with, so that the loop is not followed even once. This type of control is exercised by the DO-WHILE construct.

FORTRAN has a variety of features that can be used to construct and control loops. The most basic mechanism involves the DO WHILE loop introduced earlier (see Example 2.4). Another type of control statement enables us to specify loops in which the number of repetitions are counted automatically. (Recall that this mechanism was introduced in Figure 2.6.) When expresed in FORTRAN, these automated loops have general structures such as those shown in Figure 3.7. As the figure indicates, each loop begins with a

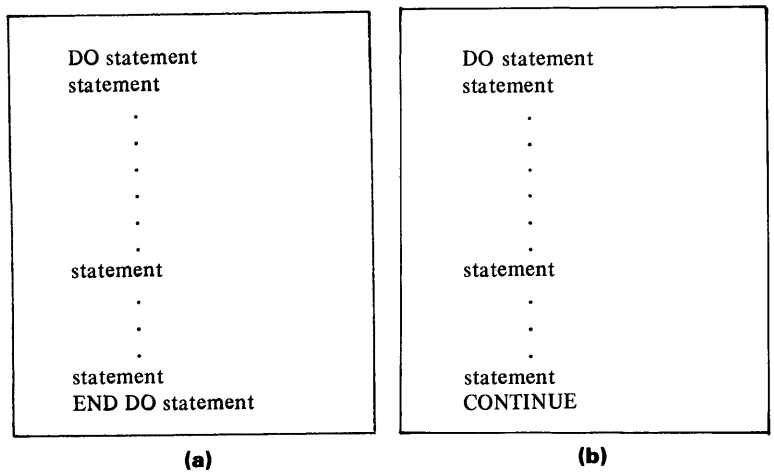



**FIGURE 3.7** Basic Construction of an Automated Loop (Also called a DO loop).

DO statement that defines the way the loop is controlled. In one form (standard in all FORTRAN 77 dialects) the end of the loop is marked with a CONTINUE statement. The CONTINUE provides a convenient bookkeeping device which makes it easy to emphasize the loop as a single activity (Figure 3.7a). An alternative framework available in HP FORTRAN 77 uses the END DO statement as an explicit conclusion for a loop. This structure is shown in Figure 3.7b.

**Example 3.4** The previous example was set up to handle exactly one line of input and then stop. While this is very simple, it is not realistic. In practice, a well designed program is constructed to handle any number of cases in a given run. We shall expand the requirements in Example 3.3 as follows: Instead of processing just one set of NAME, LENGTH and WIDTH values, we would like to read, process and print results for each of several sets. The number of sets (which we shall call NSETS) is likely to be different each time the program is used. However, that number will be known ahead of time and we shall read it in ahead of everything else. Thus, the input for each run now will consist of a single value for NSETS followed by NSETS lines, each containing a value for each of the variables NAME, LENGTH, and WIDTH. Also, the program is to keep track of the total area by adding each newly computed area to a running sum. After the results have been printed for the last set of input, the program is to print the number of input sets and the total area.

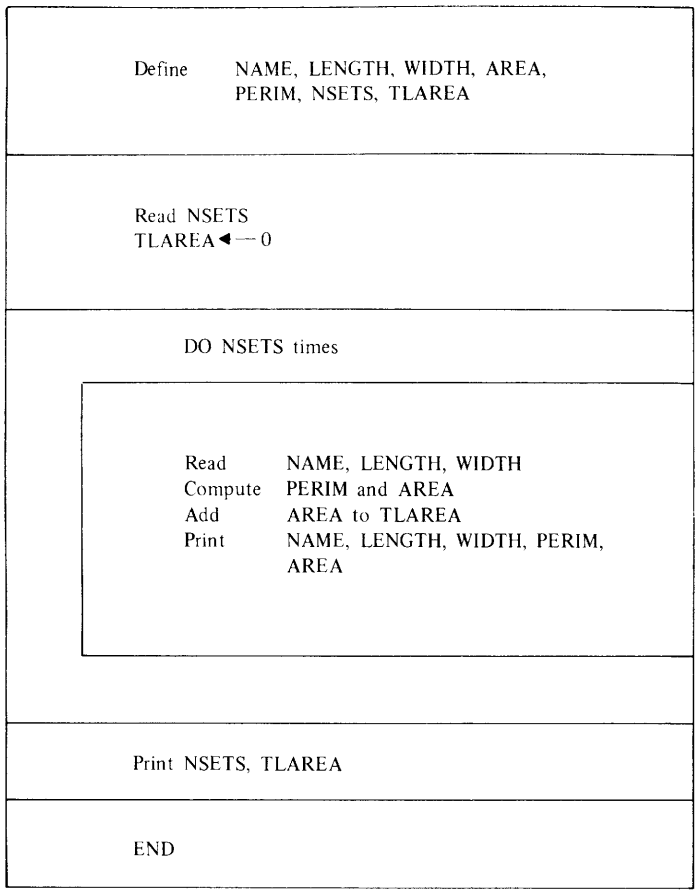
To meet these expanded requirements, we shall define two additional variables: NSETS, into which we shall read the number of sets for a given run, and TLAREA, the variable where the total area will be accumulated. NAME, LENGTH, WIDTH, PERIM, and AREA will be used as before.

The program design (Figures 3.8(a) and 3.8(b)) reflects these conditions. After defining the necessary variables, we get the processing started by setting the stage for the controlled repetition within the loop. For this example, the preparation involves reading the value of NSETS so the program knows how many times it will read and process input sets. In addition, TLAREA has to be set to zero so that the program can add to it as it goes around the loop. This process of stage-setting is called *initialization*. It is an essential part of every program. The specific activities performed during initialization will depend, of course, on the requirements of the particular algorithm.

Back to our program: Having completed the initialization, the program then will repeat the loop NSETS times. Each time through the loop it will:

1. Read a set of NAME, LENGTH and WIDTH values.
2. Compute PERIM and AREA.
3. Bring TLAREA up to date by adding the newly computed AREA to it.
4. Print the results (NAME, WIDTH, LENGTH, AREA, and PERIM).

After NSETS times through this loop, the program will break this pattern and perform its final activity, i.e., display the number of input sets (which simply is NSETS) and TLAREA. These events are represented by the pseudocode and N-S diagram in Figure 3.8 and the program itself is shown in Figure 3.9. We shall focus on the loop to see how it is controlled.




(a)

```
"Define NAME, LENGTH, WIDTH, AREA, PERIM,
      NSETS, TLAREA."
"Read NSETS."
"Initialize TLAREA to zero."
Do NSETS times
      "Read NAME, LENGTH, WIDTH."
      "Compute PERIM and AREA."
      "Add AREA to TLAREA."
      "Print NAME, LENGTH, WIDTH, PERIM, AREA."
Enddo
"Print NSETS, TLAREA."
"Stop."
```

**(b)**

**FIGURE 3.8** **(a)** Structured Diagram for Example 3.4. **(b)** Pseudocode Representation for Example 3.4.

The DO statement, with which the loop begins, sets up all the bookkeeping needed to manage the repetitions. Each component in the statement supplies a piece of information:

1. The number after the word DO defines the extent of the loop. Note that this is the same one (12 in Figure 3.9) as the label attached to the CONTINUE statement at the other end of the loop, thereby telling FORTRAN what to include in each repetition. In effect, this particular example is saying, "Every time this loop is repeated, start right here (at the DO statement) and include everything up through statement number 12."

2. There is a name after the statement number. (This example uses NTIMES.) This defines a variable called an *index* whose job it is to count the number of times the loop is processed. Note that it says

```
NTIMES = 1, NSETS
```

This tells FORTRAN to use a variable named NTIMES and start it off with an initial value of 1. Then, every time the loop is processed, NTIMES is increased automatically by 1. When NTIMES reaches NSETS, the program takes a final trip around the loop. A subsequent attempt to repeat the loop is stopped because another increase of NTIMES brings it beyond NSETS, the *limiting value*.

3. Since the numerical label attached to the CONTINUE statement matches the one specified in the DO statement, the CONTINUE acts as a kind of boundary on the loop. Thus, after each repetition, the program automatically goes back to the beginning of the loop, where the control mechanism is handled. Here, the index (NTIMES in Figure 3.8) is increased by 1 and tested against the limiting value (NSETS in our case). If another repetition is required, the program does it. If not, the pattern is broken and the loop is bypassed, so that the program resumes its basic journey with the statement immediately after the CONTINUE.

We need to look at one other thing in the loop. Each time we go through the loop for each input set, we execute the statement

```
TLAREA = TLAREA + AREA
```

```
C*************************************************************
C                      EXAMPLE 3.4                           *
C*************************************************************
C  THIS PROGRAM READS AND PROCESSES AN ARBITRARY NUMBER OF   *
C  INPUT SETS IN WHICH EACH SET CONTAINS AN ESTATE NAME,     *
C  LENGTH, AND WIDTH. FOR EACH SET READ, THE PROGRAM COM-    *
C  PUTES THE PERIMETER AND AREA, AND IT PRINTS THE NAME,     *
C  LENGTH, WIDTH, PERIMETER, AND AREA. THE FIRST INPUT SET   *
C  IS PRECEDED BY A VALUE (NSETS) THAT INDICATES HOW MANY    *
C  INPUT SETS THERE WILL BE FOR THAT RUN. AFTER THE FINAL    *
C  INPUT SET IS PROCESSED, THE PROGRAM PRINTS NSETS, AS WELL*
C  AS THE TOTAL AREA (TLAREA). NTIMES KEEPS TRACK OF THE     *
C  NUMBER OF CYCLES THROUGH THE LOOP.                        *
C*************************************************************
C
      PROGRAM        EX304
      IMPLICIT         NONE
      INTEGER*2        NSETS,LENGTH,WIDTH,PERIM,AREA,TLAREA,NTIMES
      CHARACTER*8      NAME
C
C*************************************************************
C                          INITIALIZATION                    *
C*************************************************************
      PRINT *, 'ENTER A VLUE FOR NSETS'
      READ *,  NSETS
      TLAREA = 0
C
C*************************************************************
C  THIS IS THE PROCESSING LOOP:                              *
C  ALL THE ACTIVITIES BETWEEN THE "DO" STATEMENT AND THE     *
C  "CONTINUE" STATEMENT WILL BE REPEATED A SPECIFIC NUMBER   *
C  OF TIMES, THAT NUMBER BEING DETERMINED BY THE VALUE IN    *
C  NSETS. NTIMES, THE INDEX, WILL KEEP TRACK OF THE CYCLES   *
C  AUTOMATICALLY.                                            *
C*************************************************************
      DO 12 NTIMES = 1, NSETS
        PRINT *, 'ENTER VALUES FOR NAME, LENGTH, WIDTH'   !PROMPT
        READ *,  NAME, LENGTH, WIDTH
        PERIM = 2 * (LENGTH+WIDTH)
        AREA = LENGTH * WIDTH
        TLAREA = TLAREA+AREA
        PRINT *,  NAME,LENGTH,WIDTH,PERIM,AREA
   12 CONTINUE
C
C*************************************************************
C  PRINT THE SUMMARY RESULTS                                 *
C*************************************************************
      PRINT *,  'NO. OF SETS PROCESSED: ',NSETS
      PRINT *,  'TOTAL AREA: ',TLAREA
C
      END

C *** ALTERNATIVE LOOP ***

      DO  NTIMES = 1, NSETS
         PRINT *, 'ENTER VALUES FOR NAME, LENGTH, AND WIDTH'
         READ *,  NAME, LENGTH, WIDTH
         PERIM = 2 * (LENGTH + WIDTH)
         AREA = LENGTH * WIDTH
         TLAREA = TLAREA + AREA
         PRINT *, NAME, LENGTH, WIDTH, PERIM, AREA
      END DO
```

**FIGURE 3.9** Program for Example 3.4.

```
5
'BONNMOOR'          315     212
'BOGSWAMP'          340     281
'HILLDALE'          280     280
'RAVENTON'          167     226
'COSTALOT'          303     189
```

**FIGURE 3.10** Sample Input for Example 3.4.

```
BONNMOOR            315     212     1054    66780
BOGSWAMP            340     281     1242    95540
HILLDALE            280     280     1120    78400
RAVENTON            167     220      774    36740
COSTALOT            303     189      984    57267
NO. OF SETS PROCESSED:  5
TOTAL AREA:  334727
```

**FIGURE 3.11** Sample Output for Example 3.4.

At first glance this appears to make no sense: How can TLAREA be equal to itself plus something else? Recall, however, that the = sign means something quite different here. It represents the assignment operation. So this statement says, "Take the value currently in TLAREA, add to it the value currently in AREA, and use the result as the new value in TLAREA, replacing what was just there." Once all the input has been processed, we stop going around the loop and print the final information.

Now that we have analyzed the operation of the program, we can examine its behavior for a specific run. The input for such a run, shown in Figure 3.10, produces the results shown in Figure 3.11.

Although we can construct a wide range of very powerful control mechanisms with the FORTRAN features discussed so far, it should be recognized that our expressive abilities will be increased by additional language facilities. We shall introduce these later, as skills build and expand.

## 3.2 PREPARATION OF FORTRAN STATEMENTS

By now we have seen enough FORTRAN statements to develop an overall impression about their appearance. In this section we shall become more specific about these forms by describing the rules that govern their construction.

### 3.2.1 Format of a Statement

FORTRAN statements are organized as *lines* of information. A line is an actual line entered through a terminal. Each line has 80 positions, so that it can hold as many as 80 characters.

A FORTRAN statement occupies at least one line, with certain parts of a statement being limited to particular areas on the line. To make the writing of these statements more convenient, many people use *coding forms* such as the one shown in Figure 3.12. Each line on the form corresponds to a line. Note that certain boundaries are marked with heavy lines. We shall discuss each of these, relating it to a particular part of a statement.

IBM

FORTRAN Coding Form

GX28-7327-6 U/M 050**
Printed in U.S.A.

PROGRAM EXAMPLE 3.4 (with Comments removed) | PUNCHING INSTRUCTIONS | GRAPHIC | PAGE OF

PROGRAMMER S.V. POLLACK | DATE 6/13/82 | PUNCH | CARD ELECTRO NUMBER*

| COMM. | STATEMENT NUMBER | CONT. | FORTRAN STATEMENT | IDENTIFICATION SEQUENCE |
|---|---|---|---|---|
| C | | | EXAMPLE 3.4 | 00000010 |
| | | | PROGRAM EX304 | 00000020 |
| | | | IMPLICIT NONE | 00000030 |
| | | | INTEGER*2 NSETS, LENGTH, WIDTH, PERIM, AREA, TLAREA, NTIMES | 00000040 |
| | | | CHARACTER*8 NAME | 00000050 |
| | | | | 00000060 |
| | | | WRITE (1,*) 'ENTER A VALUE FOR NSETS' | 00000070 |
| | | | READ (1,*) NSETS | 00000080 |
| | | | TLAREA = 0 | 00000090 |
| | | | | 00000100 |
| | | | DO 12 NTIMES = 1, NSETS | 00000110 |
| | | | WRITE (1,*) 'ENTER VALUES FOR NAME, LENGTH, WIDTH' !PROMPT | 00000120 |
| | | | READ (1,*) NAME, LENGTH, WIDTH | 00000130 |
| | | | PERIM = 2 * (LENGTH+WIDTH) | 00000140 |
| | | | AREA = LENGTH * WIDTH | 00000150 |
| | | | TLAREA = TLAREA+AREA | 00000160 |
| | | | WRITE (1,*) NAME, LENGTH, WIDTH, PERIM, AREA | 00000170 |
| | 12 | | CONTINUE | 00000180 |
| | | | | 00000190 |
| | | | WRITE (1,*) 'NO. OF SETS PROCESSED: ', NSETS | 00000200 |
| | | | WRITE (1,*) 'TOTAL AREA: ', TLAREA | 00000210 |
| | | | | 00000220 |
| | | | END | 00000230 |
| | | | | 00000240 |

*A standard card form, IBM electro 888157, is available for punching statements from this form

**FIGURE 3.12** Coding form Showing Statement Boundaries.

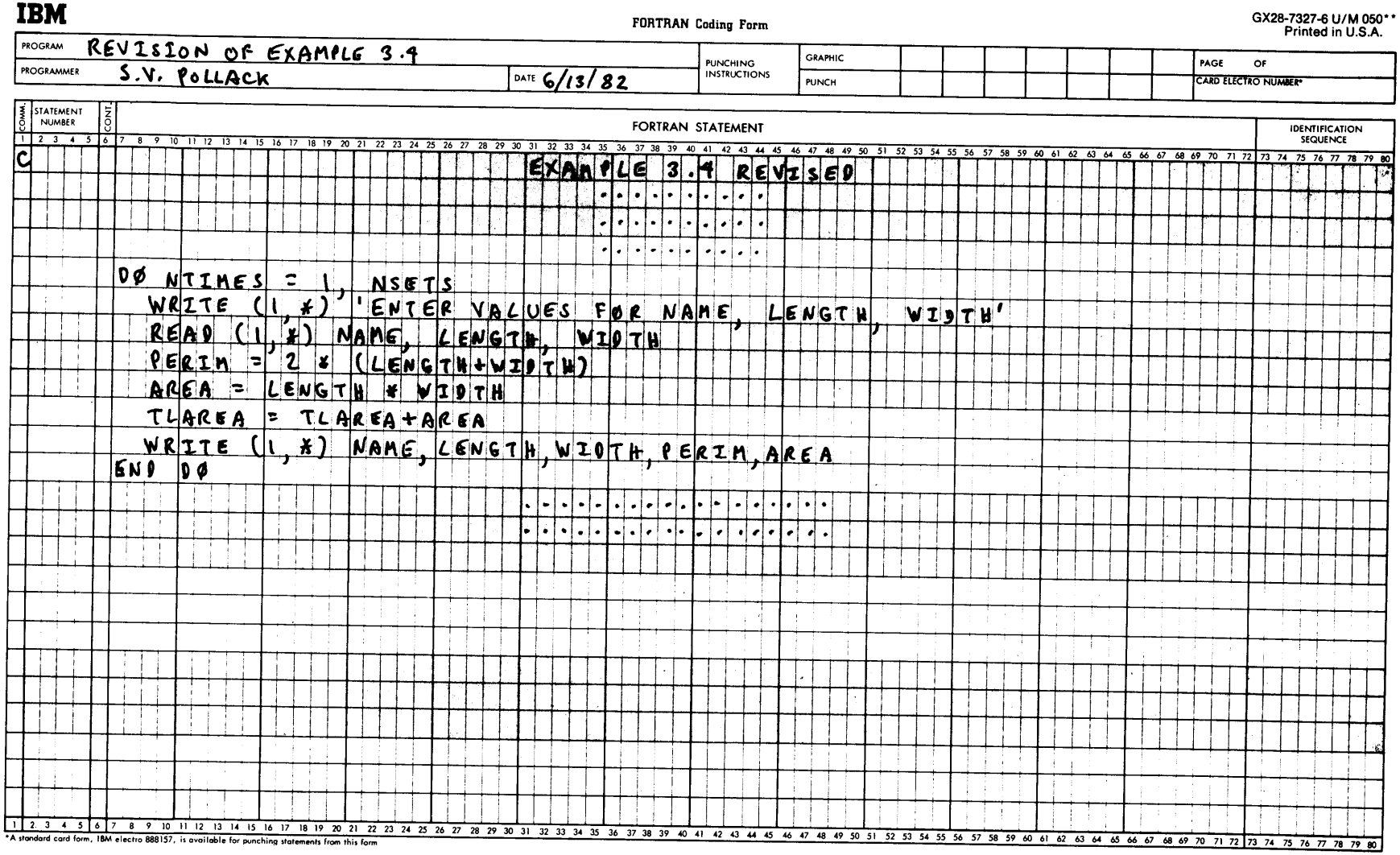



**FIGURE 3.13** Example 3.4 Coding with Loop Indented Coding Form Showing Indented Loop.

**3.2.1.1 Statement Labels** In Example 3.4 (Figure 3.9) we attached the number 12 to the CONTINUE statement as a statement label, thereby giving that statement a "name" by which other statements in the program can refer to it. Most types of statements can be labeled. (Exceptions will be pointed out as we get to them.) Only certain statements must be labeled. In general, we shall find that few statements in a program need labels. Note that if we were to rewrite the loop in Example 3.4 using HP FORTRAN 77's DO. . . END DO form, there would be no need for the label. This is shown in Figure 3.13.

When a label is used, it must appear somewhere in the first five positions. If the statement is unlabeled, those positions remain blank. No more than one label may be attached to a statement, and that label is an unsigned integer consisting of one to five digits. A particular label can be used as such only once in a program. Note that even though a label consists only of digits, FORTRAN does not do any arithmetic with it; it is merely a name. Consequently, a statement near the beginning of a program may have a label of, say, 125 and a statement further on may be labeled 16, or 290, or any other number from 1 through 99999, as long as the number is unique.

When a label is less than five digits long, it may be positioned anywhere in positions 1–5. (This is what was meant earlier by "somewhere in the first 5 positions." The unused positions must be blank and there should be no blanks in the middle of the label. (FORTRAN allows blanks in the middle of statement labels (e.g., `12 5` or `4 1 4`), but this is poor practice which will not be followed here.) Thus, `125bb`, `bb125`, and `b125b` all are treated by FORTRAN as the same label. The symbol `b` is used here (and throughout the text) to denote a single blank. Table 3.2 shows some examples of legally and illegally constructed statement labels.

**Table 3.2** Construction of Statement Labels in FORTRAN

| *Label* | *Remarks* |
|---|---|
| `12021` | OK |
| `bbbb6` | OK |
| `b34bb` | OK |
| `bbbJ4` | Illegal; non-numeric character |
| `32bb4` | Illegal; contains embedded blanks |
| `b8888b` | OK |
| `7888889` | Illegal; too long |
| `43/63` | Illegal; non-numeric character |

**3.2.1.2 Sequence Numbers** Positions 73–80 cannot be used as part of a statement. FORTRAN ignores anything that may be entered there. Instead, these positions are available for *sequence numbering*. That is, the programmer can enter a number on each line without interfering with the program itself.

The use of sequence numbers is illustrated on the coding form in Figure 3.12. Note that the numbers are 10 apart: Instead of numbering the lines 00000001, 00000002, etc., they are sequenced as 00000010, 00000020, etc. This is done so that if a program has to be changed by inserting new lines somewhere in the middle, the sequence can still be preserved. For instance, if we were to need a line between lines 00000070 and 00000080, we could number that line 00000075.

**3.2.1.3 The Body of a Statement** Having tied up positions 1–5 and 73–80, we are left with positions 6–72. As will be seen in the next section, position 6 is reserved for another specific purpose, so that the actual statement itself (the *body*) is written in positions 7–72. Specifically, the body of a statement cannot begin before position 7. However, it may start anywhere after position 7 (up to position 72). This flexibility is useful because it allows us to emphasize the structure of a program by indenting certain portions of it relative to others. We shall make frequent use of indentation in examples throughout the book.

**3.2.1.4 Continuation of Statements on Additional Lines** Every FORTRAN statement must begin on a new line. However, it does not have to end on that same line. Longer statements may extend onto additional (*continuation*) lines. This is the reason position 6 is reserved: It is used to indicate whether that line starts a new statement or continues a previous one. If a line is the first (or only) one for a statement, position 6 is left blank. Programmers may use a zero in position 6 instead of the blank. FORTRAN recognizes a continuation line as such because it has something other than a blank or zero in position 6. A statement may have as many continuation lines as necessary. The formatting rules still apply: Only positions 7–72 may be used on each line for the body of the statement. Positions 1–5 on continuation lines are irrelevant, because a statement can have only one label.

When long statements are used, some people like to number the continuation lines systematically. For instance, many programmers will place a 0 in position 6 of the first line (leaving it blank if it is the only line for that statement), 1 in position 6 of the first continuation line, and so on. Letters in alphabetical order can be used if the statement is so long that continuation lines are needed beyond 9. There will be few occasions where we shall need long statements here. Consequently, we shall adopt the practice of using 1, 2, etc. as continuation signals.

**3.2.1.5 Comments in Programs** We have already used comments in prior examples, so that their use (and usefulness) has been established. Consequently, the structural rules will be given here so that the picture is complete.

Any line starting with a C or * in position 1 is treated by FORTRAN as a comment. This means that it is not treated as a statement. When the original program is printed, the comment is included as part of the display, but that is as far as it goes. FORTRAN never actually analyzes it; once it sees the C in position 1, it loses interest. A single line is either all statement or all comment, never both. There is no limit to the number of comment lines a program may contain, nor is there any restriction as to where in the program such comments may be placed. In addition HP FORTRAN 77 allows comments to appear on the same lines as program statements. Such comments start with an exclamation point (!) and may appear anywhere after the statement as long as they appear prior to position 72. An example is seen in Figure 3.9.

**3.2.1.6 A Brief Treatise on Blanks** In writing programs thus far, we have not paid particular attention to the use of blanks. As it turns out, we do not have to. In most cases, FORTRAN obtains enough information from the symbols in a statement to be able to separate them from one another. Consequently, we generally use blanks to increase visual clarity. For example, the following two assignment statements

```
Y=A* (B-C) - (D-E-F) * (3.8/E)
```

and

```
Y = A * (B-C) - (D-E-F) * (3.8/E)
```

produce identical results. FORTRAN ignores all the blanks it does not need. Accordingly, it is possible to place any number of blanks between any symbols in a statement and we should do so wherever it makes things easier to read. We should not break up a symbol (like a variable name) by inserting blanks in the middle of it. (This is legal, but it is potentially confusing.)

### 3.2.2 The FORTRAN Character Set

HP FORTRAN 77 recognizes a set of characters consisting of upper and lower case letters (A–Z and a–z), digits (0–9), and special characters. These are defined in Table 3.3. Note that the letters (*alphabetic characters*) together with the ten digits (*numeric characters*) form the group of *alphanumeric* or *alphameric characters.*

Some care needs to be exercised when writing FORTRAN programs on coding sheets (especially if a person other than the programmer will be keypunching from the handwritten code). It is relatively easy to confuse certain characters. To avoid this, we shall use the following conventions: The final letter in the alphabet, when handwritten, will appear as to distinguish it from a carelessly written 2. Similarly the letter O ("oh") will be distinct from zero (0) and the letter I will look different from the number one (1).

### 3.2.3 Directives to the Compiler

When somebody writes a FORTRAN program, that person is not communicating directly with the computer. Rather, his or her dealings are with a program (the FORTRAN compiler). Consequently, the information submitted to the compiler need not be restricted to specifications describing the intended program's structure and processing characteristics. There is also an opportunity to guide the behavior of the compiler itself as it processes the FORTRAN statements. The HP FORTRAN 77 compiler is designed to take advantage of this opportunity by including facilities for accepting a number of *directives* that are not part of the programming language. This section introduces the general concept and describes several directives that are of general use. The material will serve as background for using the other, more specialized directives described in the FORTRAN 77 Reference Manual supplied with your system.

**3.2.3.1 Use of Directives** A directive starts on a new line with a dollar sign ($) in position 1. Directives may appear anywhere in a FORTRAN program where their inclusion makes sense.

**3.2.3.2 The $LIST Directive** One of the basic supporting services provided by any compiler is the production of a listing showing the program statements submitted by the programmer. Depending on the particular compiler, this may range from a literal replica of the source program "as received" to an elaborate display reflecting a considerable amount of analysis, editing, and formatting. One of the features in the HP FORTRAN 77 compiler makes it possible to suppress all or part of the listing altogether. This is done simply by including a line in the FORTRAN program as follows:

```
$LIST OFF
```

As a result, subsequent lines will be processed, but they will not appear on the listing. Suppression continues until the compiler encounters a directive that says.

```
$LIST ON
```

The $LIST directive may appear in a program as many times as desired.

**3.2.3.3 The $PAGE Directive** When the compiler prints a program listing, it uses an internal counter to control the number of lines on a page. There are numerous occasions wherein the legibility of the listing can be enhanced by starting certain sections on separate pages. The compiler will start a new page (regardless of the number of lines printed on the current page) in response to the directive

```
$PAGE
```

An arbitrary number of $PAGE directives may appear throughout the program. Of course, this directive will be ignored if it appears in a part of the program for which listing has been suppressed by the $LIST OFF directive.

**3.2.3.4 The $TITLE Directive** As another aid to program legibility, the HP FORTRAN 77 compiler enables the programmer to include a descriptive line and have it appear at the

**Table 3.3** Characters Available for Use in HP FORTRAN 77

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | a | N | n | 0 | 1 | blank | = |
| B | b | O | o | 2 | 3 | + | - |
| C | c | P | p | 4 | 5 | * | / |
| D | d | Q | q | 6 | 7 | ( | ) |
| E | e | R | r | 8 | 9 | . | , |
| F | f | S | s | | | : | ! |
| G | g | T | t | | | $ | _(underscore) |
| H | h | U | u | | | ' | " |
| I | i | V | v | | | @ | |
| J | j | W | w | | | | |
| K | k | X | x | | | | |
| L | l | Y | y | | | | |
| M | m | Z | z | | | | |

top of each listed page. This information, not part of the program, is specified by the directive

$TITLE descriptive information

The descriptive information may be up to 46 characters long, beginning with the first nonblank character after $TITLE. In a typical instance, the programmer places such information immediately prior to the PROGRAM statement. As a result, the information will appear at the top of each page of the listing until the compiler runs across another $TITLE directive, in which case a new page is started immediately with the new title at the top. When a new title appears while $LIST OFF is in effect, the title information is changed immediately, but (of course) it does not appear until the first page after $LIST ON is specified.

## 3.3 SUMMARY

The basic unit of expression in a program is the *statement* which, like a sentence, has its length determined by the activity being described. FORTRAN statements are conveniently categorized as follows:

1. Program Boundaries—The PROGRAM and END statements begin and end a FORTRAN PROGRAM.
2. Specification—This type of statement defines variables and describes information to be used in a program.
3. Assignment—These statements define actual computations: An expression value is produced by performing specified operations, and that value, (i.e., the result) is placed in (assigned to) a designated variable, replacing the previous contents.
4. Input/Output—These statements (READ, WRITE, PRINT) move data between the central processor and the outside world.
5. Control—These capabilities, represented so far by the IF, DO, DO WHILE, END DO, and CONTINUE statements, enable the programmer to set up alternative pathways and automatic loops in a program.

Each FORTRAN statement corresponds to one or more lines. Positions 1–5 are reserved for a numeric label, which is optional for most statements. The body of the statement appears anywhere in positions 7–72. The remaining positions (73–80) are available for *sequencing* and are ignored by FORTRAN. Statements needing more than one line are continued in positions 7–72 of additional lines. Such *continuation lines* are marked in position 6 by any character (letter, digit, or special character) other than blank or zero.

**PROBLEMS**

1. Write the appropriate FORTRAN statement (or statements) for each of the following:

   (a) Start a program named DSCOMP
   (b) Start a program named FIX31
   (c) Declare an integer variable named BACKLG
   (d) Declare an integer variable named INSTS
   (e) Declare a character variable named ADVERB having a length of 10
   (f) Declare two integer variables named FORCE and MOVE42
   (g) Declare three integer variables named COUNTS, JMAX, and CLASSES
   (h) Declare three character variables named WORDS, SGNTS, and NOUNS, each with a length of 12
   (i) Declare three character variables named INSTR, OUTSTR, and BLDSTR having respective lengths of 4, 3, and 21 characters
   (j) Declare integer variables VALX and VALZ, and a character variable NUMVAL with length 4
   (k) Declare integer variables WDMEAS and PHRMIX, and character variables SNGS, DSCRS and OMNIS, with all of the character variables having a common length of 11
   (l) Declare character variables named INTPHR and PHRINV with lengths 5 and 7, respectively, integer variables BRKPT, SCQUOT, BCR11, and NXTDAY, and character variables INDEX and ALTO9, both with length 17

2. Write the appropriate FORTRAN statement (or statements) for each of the following:

   (a) Assign a value of 104 to variable VALX
   (b) Assign a value of −26 to RATE
   (c) Assign an integer value of zero to variables YCOUNT and ZCOUNT
   (d) Assign a value of 16 to variables ADJ, ADJX2Y, and ADJX2Z
   (e) Increase the value in variable ONHAND by 247
   (f) Add 117 to AMT's current value and store the result back in AMT
   (g) Increase AMT's value by 117
   (h) Add 98 to SPEC's value and store the result in REGLR
   (i) Replace the value of REGLR with the sum of 98 and SPEC's current value
   (j) Increase REGLR's value by the su of SPEC and 98
   (k) Replace YVAL with double its current value
   (l) Assign twice YVAL's current value to DBLV
   (m) Double YVAL's current value and assign it to YVAL and NEWVAL
   (n) Replace each of the values in variables NEWV1 and NEWV2 with twice the sum of its respective current value and 86

3. Assume that the following statements

```
INTEGER*2     ADVAL, COMPT, PRTSEC, RSLT
ADVAL = 2
COMPT = 4
PRTSEC = 36
```

   precede the statement or sequence of statements in each of the following problems. For each set, then, specify the values in each of the four variables as a result of the indicated processing. (The processing in each problem is to be considered completely independent):

   (a) RSLT = ADVAL + COMPT + PRTSEC
   (b) ADVAL = ADVAL + COMPT + PRTSEC
   (c) COMPT = 2 * COMPT + ADVAL − PRTSEC
   (d) COMPT = 2 * (COMPT + ADVAL − PRTSEC)
   (e) RSLT = PRTSEC * (ADVAL + COMPT)
       ADVAL + ADVAL + RSLT
   (f) RSLT = ADVAL * (COMPT − PRTSEC)
       ADVAL = ADVAL − RSLT
       RSLT = COMPT * ADVAL + PRTSEC

4. Write the appropriate FORTRAN statement (or statements) for each of the following:

   (a) Read the next value into variable CRDNO
   (b) Read the next three values into variables ORDNUM, AMT, and COST
   (c) Display the values from variables ORDNUM, AMT, and COST in that order
   (d) Display ORDNUM's value twice, followed by COST and AMT
   (e) Read values into PRINUM, BINLOC, and ONHAND and then display them in the same order in which they were read

(f) Display variables PRTNUM, BINLOC, and ONHAND, each on a separate line
(g) Print variables PRTNUM and ONHAND on the same line, followed by BINLOC and ONHAND on the next line
(h) Read values into TEMP, HUMDTY, VEL, and CURRNT and print them in reverse order on four separate lines
(i) Read values into variables MASS and DIFFUS and display them on the same line. Then read values into variables TRANS, BLOCK, and WT and print them on three separate lines in the order in which they were read
(j) Ask the user to submit input values for AMT and COST.
(k) Ask the user to submit input values for ORDNUM, AMT, and COST, each on a separate line.

5. Assume the following declarations

```
INTEGER*2     MUMOBJ, MINFOR, STR, BKFRC, ELAST
```

and the following input line:

```
43267    -462    254    887    -3
```

Indicate the values in the five variables declared above when each of the following READ statements is applied to this line:

(a) READ *, STR, BKFRC, ELAST, NUMOBJ, MINFOR
(b) READ *, STR, ELAST, BKFRC, NUMOBJ, MINFOR
(c) READ *, ELAST, STR, BKFRC, MINFOR
(d) READ *, ELAST, STR, STR, MINFOR, NUMOBJ
(e) READ *, ELAST, BKFRC, ELAST, NUMOBJ

6. Assuming the following declarations

```
INTEGER*2       MOLWT, MELTPT, DEN
CHARACTER*8     CHMNAM, TRDNAM
```

and the following input line,

```
'CARBAZID', 153, 202, 'MELOGOOP', 88
```

indicate the values in the five variables that result when each of the following READ statements is applied to the input line:

(a) READ *, CHMNAM, MOLWT, MELTPT, TRDNAM, DEN
(b) READ *, CHMNAM, MELTPT, MOLWT, TRDNAM, DEN
(c) READ *, TRDNAM, DEN, MELTPT, CHMNAM, MOLWT
(d) READ *, TRDNAM, MELTPT, DEN, TRDNAM, DEN

7. Show the output resulting from each of the following sequences of statements:

(a)
```
INTEGER*2     NUMX, NUMY, NUMZ
NUMX = 24
NUMY = 17
NUMZ = NUMX+NUMY
PRINT *, NUMX, NUMY, NUMZ
```

(b)
```
INTEGER*2     NUMX, NUMY, NUMZ
NUMX = 14
NUMX = -6
NUMZ = 3 * (NUMX+NUMY)
PRINT *, NUMX, NUMY
PRINT *, NUMZ
```

(c)
```
INTEGER*2     SET1, SET2, SET3, SET4
SET1 = 36
SET2 = 2*SET1+3
SET3 = SET2-15
SET4 = SET3-SET1
PRINT *, SET1
PRINT *, 'SET2: ', SET2
PRINT *, 'SET3 HAS A VALUE OF ', SET3
PRINT *, 'SET4 = ', SET4
```
Use the following input line:
```
18    12    5
```

(d)
```
INTEGER*2     XVAL, YVAL, ZVAL, RSLT
READ *, XVAL, YVAL, ZVAL
RSLT = XVAL+YVAL-2*ZVAL
PRINT *, 'INPUT VALUES: '
PRINT *, 'XVAL: ', XVAL
PRINT *, 'YVAL: ', YVAL
PRINT *, 'ZVAL: ', ZVAL
PRINT *, 'FINAL RESULT: '
PRINT *, 'RSLT: ', RSLT
```

(e)
```
      INTEGER*2     LTH, WDTH, HT, VOL
      VOL = 0
      READ * LTH, WDTH, HT
      VOL = LTH*WDTH, HT
      PRINT *, 'LTH = ', LTH, '
     1 WDTH = ', WDTH, 'HT = ', HT
      PRINT *, 'VOLUME = ', VOL
```

Use the following input line:

```
20    11    3
```

(f)
```
      INTEGER*2
      INGR1, INGR2, INGR3, EXPR
          CHARACTER*7
      LABEL
      EXPR = 0
      READ *, LABEL, INGR3, INGR1
      EXPR = EXPR+4* (INGR1+INGR3)
      PRINT *, 'NAME = ', LABEL
      PRINT *, 'EXPR IS: ', EXPR
      PRINT *, INGR1, INGR2, INGR3
```

Use the following input line (be extra careful here):

```
'BUILDER', 32, 14, 6
```

8. Consider a program that begins as follows:

```
PROGRAM          C3P08
IMPLICIT           NONE
INTEGER*2          PRT1, PRT2, PRT3, TTLPRT
TTLPRT = 0
PRT1 = 74
PRT2 = 18
PRT3 = 3 * (PRT1 - 2*PRT2)
```

Assume that each of the sequences given below follows the last assignment shown above. For each one (taken independently) specify what the printout will show:

(a)
```
IF (PRT1 . EQ. PRT2) TTLPRT = TTLPRT+PRT1
PRINT *, PRT1, PRT2, PRT3, TTLPRT
```

(b)
```
IF (PRT2 . LT. PRT3) TTLPRT = PRT1-PRT2
PRINT *, PRT1, PRT2, PRT3, TTLPRT
```

(NOTE: . LT. means "is less than")

(c)
```
IF (PRT1+PRT2 . LT. PRT3) PRT3 = PRT3+PRT1
PRINT *, PRT1, PRT2, PRT3, TTLPRT
```

(d)
```
IF (PRT1+PRT2 . LT. PRT3-PRT2) THEN
    PRT3 = PRT3 - PRT1
ELSE
    PRT2 = PRT3 - PRT
END IF
PRINT *, PRT1, PRT2, PRT3, TTLPRT
```

9. Consider the following sequence of statements:

```
INTEGER*2          ALL, TRIPS, PROD
ALL = 0
PROD = 1
DO TRIPS=1, 6
    ALL = ALL + 1
    PROD = PROD * ALL
END DO
PRINT *, ALL, PROD
```

(a) How many times is the loop processed?
(b) Show what is printed.

10. Consider the following sequence of statements:

```
INTEGER*2          BSUM, CYCLES, NBL, NDR, REPS
BSUM = 0
NBL = 4
NDR = 3*NBL
CYCLES = NDR- (NBL+3)
DO 8 REPS = 1, CYCLES
    BSUM = BSUM + 6 - NBL
    NDR = NDR - 1
8 CONTINUE
CYCLES = CYCLES - 2
PRINT *, CYCLES, NBL, , NDR, , BSUM
```

(a) How many times will the loop process?
(b) Show what is printed.

11. In the following sequence of statements,

```
INTEGER*2          TSQ, LIMIT, NVAL
TSQ = 4
LIMIT = 28
DO WHILE (TSQ . LE. LIMIT)
    TSQ = TSQ + 2
    NVAL = NVAL + 1
END DO
PRINT *, NVAL, TSQ, LIMIT
```

(a) What does NVAL indicate?
(b) How many times does the loop process?
(c) What does the printout show?
(NOTE: . LE. means "is less than or equal to")

12. In the following sequence of statements,

```
INTEGER*2          MVMT, TRNS, BASE
MVMT = 24
BASE = 0
TRANS = MVMT * BASE
PRINT *, MVMT, BASE, TRANS
DO WHILE (TRANS . GE. MVMT)
    BASE = BASE+3
    TRANS = TRANS - (MVMT-BASE)
    MVMT = MVMT + 1
END DO
PRINT *, MVMT, BASE, TRANS
```

(a) How many times is the loop processed?
(b) Show what each of the PRINT statements produces.
(c) Modify the program fragment shown above by introducing an integer variable NUMTR that is used to keep track of the number of times the loop is processed. The final value is to be printed, on a separate line, after the last PRINT statement given above. (NOTE: . GE. means "greater than or equal to")

13. In the following sequence of statements,

```
INTEGER*2          LSUM, RSUM, PIVOT, TRIPS, TOP
LSUM = 0
RSUM = 0
PIVOT = 3
TOP = 14 + 3*PIVOT
DO TRIPS = 1, TOP
    IF (PIVOT . EQ. 3) THEN
        PIVOT = 1
    ELSE
        PIVOT = 3
    END IF
    IF (PIVOT . EQ. 3) THEN
        LSUM = LSUM + 5
    ELSE
        RSUM = RSUM + 2
    END IF
END DO
PRINT *, LSUM, RSUM
```

(a) How many times is the loop processed?
(b) How many values were used to compute LSUM?
(c) How many values were used to compute RSUM?
(d) What is PIVOT's value just prior to the PRINT statement?
(e) What does the display show?
(f) (more challenging): Alternative actions specified in an IF-block (see Section 3.1.2.5) may be more than one statement long. Knowing that, rewrite the loop given above so that the same processing is achieved with only a single IF-THEN-ELSE construct.

14. Modify the program shown in Example 3.4 so that it will include the decision-making capability of Example 3.3A.

Each problem in the next group presents a FORTRAN program in which one or more statements are absent. In their place there is a pseudocode or N-S representation of the intended processing. Prepare the appropriate FORTRAN 77 statements in each case.

15.
```
PROGRAM                 C3P15
IMPLICIT                    NONE
INTEGER*2               K1, K2, K3, K4, K5, K6
READ *. K1, K2, K3, K4
seq
"COMPUTE K5 = K1K2 - K3K4."
"COMPUTE K6 = K1K3 - K2K4. "
endseq
PRINT *, K1, K2, K3, K4
PRINT *, 'K5 = ', K5
PRINT *, 'K6 = ', K6
STOP
END
```

16.
```
PROGRAM                 C3P16
IMPLICIT                    NONE
INTEGER*2                   K1, K2, K3, K4, K5, K6
do 5 times:
  "Request values for K1, K2, K3, and K4."
  READ *, K1, K2, K3, K4
  K5 = (K1+K3) * (K2+K4)
  K6 = (K1+K4) * (K2+K3)
  PRINT *, K1, K2, K3, K4
  PRINT *, 'K5 = ', K5
  PRINT *, 'K6 = '. K6
end do

STOP
END
```

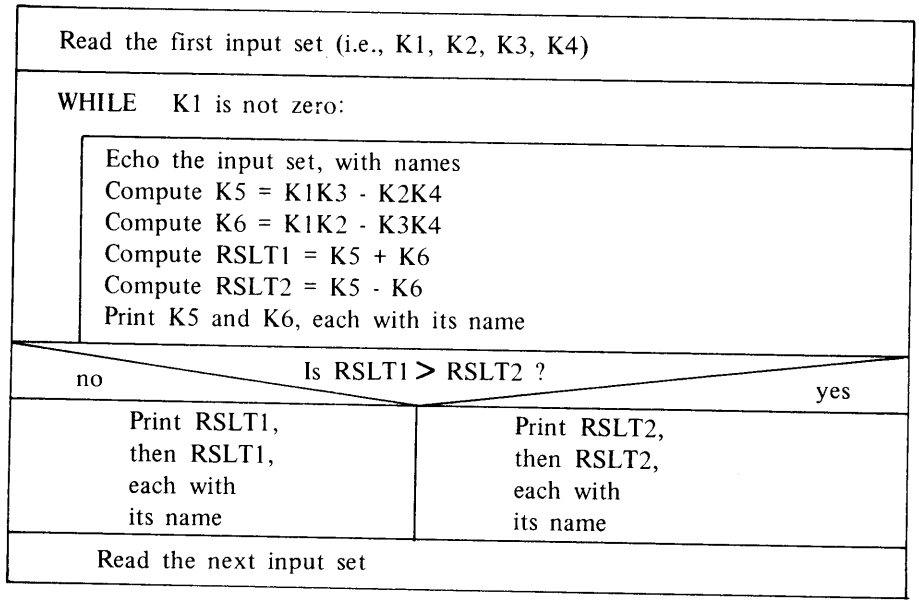
Declare any additional variables you might need.

17.
```
PROGRAM                 C3P17
IMPLICIT                    NONE
INTEGER*2                   K1, K2, K3, K4, K5, K6, NTIMES
READ *, NTIMES
   DO ntimes times
      "Read K1, K2, K3, and K4."
      "Echo K1, K2, K3, and K4 on one line."
      "Compute K5 = K1K2 - K3K4."
      "Compute K6 = K1K4 - K2K3."
      "Print K5 and K6 on separate lines,
           each with its name."
      enddo
STOP
END
```

Make sure you declare any additional variables you may need.

18.
```
PROGRAM                 C3P18
IMPLICIT                    NONE
INTEGER*2               K1, K2, K3, K4, K5, K6, BIGK, SMALLK
"Request values for K1, K2, K3, and K4."
READ *, K1, K2, K3, K4

while K1 is not zero
    "Compute K5 = K1K2 - K3K4."
    "Compute K6 = K1K4 - K2K3."
    "Print K5 and K6 on separate
    lines, each with its name."
    IF
    "K5 > K6."
    THEN
        "BIGK = K5 and SMALLK = K6."
    ELSE
        "BIGK = K6 and SMALLK = K5."
    ENDIF
    "Print BIGK and SMALLK on one line,
    with their respective names."
    "Read the next set of input values."
ENDWHILE

STOP
END
```

19.
```
PROGRAM                 C3P19

IMPLICIT                    NONE

INTEGER*2               K1, K2, K3, K4, K5, K6, RSLT1, RSLT2
```




```
STOP
END
```

20. Using the following set of input lines:

```
                                    6    10    21    30        first line
                              24    16    8     2             second line
                        3     7     5     12                  third line
                  8     6     10    7                         fourth line
            0     3     18    22                              fifth line
```

(a) Show the printout for the program in Problem 15.
(b) Show the printout for the program in Problem 16.
(c) Show the printout for the program in Problem 18.
(d) Show the printout for the program in Problem 19.

21. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 4 in Chapter 2.

22. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 5, Chapter 2. Define any additional variables you may need, and include appropriate comments as part of your program.

23. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 11, Chapter 2. Assume that all of the X's are integers.

24. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 12, Chapter 2. Define all the variables required to do this, and include appropriate comments in your program.

25. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 13, Chapter 2. *Omit the computation and printout of the last variable given in that problem* (*i.e.*, ROOTNM). Define any additional variables you may need.

26. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 14, Chapter 2. Define any additional variables you may need.

27. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 15, Chapter 2.

28. Modify the program in Problem 27 so that it meets the specifications given in Problem 17, Chapter 2. Run the program using the following input data:

```
                                    6     8               first line
                              22    30                    second line
                        36    37                          third line
                  -17   31                                fourth line
```

29. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 21, Chapter 2.

30. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 22, Chapter 2.

31. Prepare a complete FORTRAN program from the diagram or pseudocode prepared for Problem 24, Chapter 2.

32. Here is a FORTRAN 77 program:

```
C*************************************************************
C                        PROGRAM C3P32                        *
C*************************************************************
C  P3P COMPANY SELLS THREE PRODUCTS. CUSTOMERS SEND OR PHONE*
C  IN ORDERS AND THESE ARE RECORDED SO THAT EACH LINE CON-   *
C  TAINS A CUSTOMER NUMBER CUST_NUM, THE NUMBER OF TYPE 1    *
C  PARTS ORDERED (NUM_PART1), THE NUMBER OF TYPE 2 PARTS     *
C  ORDERED (NUM_PART2), AND THE NUMBER OF TYPE 3 PARTS OR-   *
C  DERED (NUM_PART3). THE PRICE FOR EACH TYPE (PRICE1,       *
C  PRICE2, AND PRICE3, RESPECTIVELY), ARE READ IN AHEAD OF   *
C  THE FIRST DATA LINE. A CUSTOMER NUMBER OF ZERO STOPS THE  *
C  RUN. FOR EACH CUSTOMER ORDER, THE PROGRAM COMPUTES A      *
   TOTAL PRICE (BILL) AND DISPLAYS THIS AMOUNT, ALONG WITH   *
C  THE CUSTOMER NUMBER AND THE INPUT VALUES. AT THE END OF   *
C  THE RUN, THE PROGRAM DISPLAYS TTL_BILL, THE SUM OF ALL    *
C  THE INDIVIDUAL BILL AMOUNTS.                              *
C*************************************************************

       PROGRAM        C3P32
       IMPLICIT         NONE
       INTEGER*2        NUM_PART1,NUM_PAR2T,NUM_PART3,PRICE1,PRICE2,
                        PRICE3,BILL,TOTAL_BILL,CUST_NUM

       READ *, NC,PRT1,PRT2,PRT33

C*************************************************************
C  HERE IS OUR DO-WHILE CONSTRUCT THAT WILL PROCESS EACH     *
C  SET OF INPUT IN TURN, AS LONG AS THE CUSTOMER NUMBER      *
C  IS NOT ZERO.                                              *
C*************************************************************
       DO WHILE (CUST_NUM .NE. 0)
          BILL = NUM_PART1*PRICE1 + NUM_PART2PRICE2 +
          NUM_PART 3*PRICE3
          TOTAL_BILL = TOTAL_BILL + BILL
       PRINT *,     CUST_NUM,NUM_PART1,NUM_PART2,NUM_PART3,BILL
       END DO
C
C*************************************************************
C  THE END OF THE RUN HAS BEEN REACHED AND WE ARE READY TO*
C  PRINT OUR TOTAL FIGURE.                                   *
C*************************************************************
       PRINT *, 'TOTAL PRICE FOR ALL ORDERS THIS RUN: ',TOTAL
       STOP
       END
```

This is not a perfect program; no indeed. Help us out and list the things that are wrong with it.

# 4
# *Data*

Every computer program, regardless of the application for which it is intended, operates on data in some way. Consequently, it is important to pay the same kind of serious attention to the definition and specification of data as is paid to the description of the operations themselves. This chapter discusses the facilities and aids that FORTRAN provides for defining data items and making them available for use throughout the program.

## 4.1 TYPES OF CONSTANTS AND THEIR REPRESENTATION

In the examples examined so far, we have used numbers and strings of characters without paying too much attention to their exact form and organization. The fact that we have been able to ignore these details (up to now, at least) underscores the convenience of the FORTRAN language with regard to our ability to specify data "naturally." However, we cannot take full advantage of this convenience in any but the simplest situations unless we become familiar with the full scope of these data features. The first step in this process is to learn about the types of data that FORTRAN recognizes and how they are specified in a program.

### 4.1.1 Numerical Constants

When we write a simple assignment statement such as

```
B = 12.4
```

the 12.4 is a *numerical constant.* As such, we can use it in arithmetic operations and compare it with other numbers. FORTRAN recognizes several forms for numerical constants, each with its own convenient advantages.

**4.1.1.1 Integer Constants** FORTRAN accepts signed or unsigned whole numbers as legitimate integer constants. As is true in other contexts, an unsigned integer is treated as being positive. Thus, values of 314, 0, −772, 30982, and +30982 are all acceptable integer constants, with the last two examples representing the same value. A decimal point cannot be part of an integer constant, so that −208, for example is legal while −208.0 and −208. are not.

**4.1.1.2 Octal Constants** For many kinds of computer-related work, it is useful to be able to express integers in octal form (i.e., an integer to the base 8 using only the digits 0 through 7). HP FORTRAN 77 provides two forms for this purpose.

The B-form consists of an optional sign, the digits, and the letter B. Thus, `30B`, `-417B`, and `+675B` are examples of legal octal constants in B-form while `+287B` is not (8 is not a legal octal digit). This form may be used like any ordinary integer constant.

The O-form does not allow a sign. It consists of the letter O followed by a string of octal digits enclosed in apostrophes. For instance, `O'30'`, `O'2767'`, and `O'1777'` are

legitimate octal constants while `O'-317'` and `O'829'` are not. (The first one has a sign, and the second includes digits other than those in the octal system.) The O-form may appear only in a DATA statement (Section 4.3).

Octal constants in the O-form are included in the military extension (MIL-STD-1753) to the FORTRAN 77 standard, but neither form of octal constant is part of the regular language standard.

**4.1.1.3 Hexadecimal Constants** Besides octal values, computer-related work often involves the use of values expressed in hexadecimal form. Accordingly the MIL-STD-1753 extension includes this form, and HP FORTRAN 77 supports this feature.

A hexadecimal constant is specified by writing the letter Z followed by a string of hexadecimal digits (0 through 9 and A through F) enclosed in apostrophes. The digits must be unsigned. For example, `Z'01B'`, `Z'8F88'`, `Z'C1C2'`, and `Z'D3D5D8'` all are acceptable hexadecimal constants. As is true with the O-form of octal constants, hexadecimal constants may appear only in DATA statements.

**4.1.1.4 Real Constants** A *real constant* in FORTRAN is any number that has (or can have) a fractional portion. In general, real numbers are used for most computations, while the major role of integers is that of counting and keeping track of data and events in a program as it progresses through its run. These differences in usage make it convenient to enable the programmer to express real constants in various forms.

**4.1.1.4.1 The basic form for real constants.** As we have seen earlier, a number with a fractional portion can be represented in conventional form (for example, 22.4, −178.39, 0.00613) without any problem. As is true with integer constants, an unsigned real constant is treated as a positive number. Since there is a difference between the way real numbers and integers are stored inside the processor's memory, we must make sure that a processor does not mistake a real constant for an integer. For this reason, a real constant must be shown with a decimal point, even if it does not happen to have a fractional part. The decimal point indicates that the constant *can* have a fractional portion. For instance, the integer 0 when stored in the machine appears quite differently from 0. or 0.0, both of which are real constants. Similarly, the values −2., −2.0, and −2.00 all are representations of the same real constant and will have identical appearances inside the machine. The value −2, on the other hand, is an integer and it is stored quite differently.

**4.1.1.4.2 Scientific forms for real constants.** It often happens that very large or very small numbers are required for a particular computational process. In such cases it may be inconvenient to express these numbers in the basic (conventional) form discussed in the previous section. For example, numbers like 624000000000 or 0.00000000917 are all too easily copied with too many zeros or not enough of them. To make it more convenient for such numbers to be specified, FORTRAN recognizes another form known as *exponential* or *scientific notation.* In this method of expression, the number is written in two parts: the digits themselves (usually as a fraction), and a power of 10. This power is selected such that when the first component is multiplied by it, the result is the number we wish to express.

To illustrate, let us write the number 602400.0 in scientific notation. To do this, we must recognize the fact that the digits 6024 represent the *significant digits* in this value in that they describe the number. The two zeros to the right of the 4 merely indicate how big the number is. Consequently, expression in scientific notation, whatever the appearance of the final outcome may be, must preserve the significant digits. Thus, once the appearance of the first component is defined, the value of the second component would be set to make the entire constant come out correctly. Looking at our original value 602400.0, let us suppose that our first component in the scientific representation of 602400.0 is to be

0.6024. The second component, then, must provide the answer to the question: "What is the power of 10 which, when multiplied by 0.6024, will give a value of 602400.0?" Since the answer is 6 (that is, we need to multiply 0.6024 by 10 to the sixth power to set the proper magnitude), we can conclude that 602400.0 can be represented in scientific notation as 0.6024 × 10 to the sixth power. One way of writing this in FORTRAN is

```
0.6024*10**6
```

Recall that the asterisk is used to denote multiplication. The double asterisk is used to indicate a number being raised to a power. Obviously, this is not the only way to represent this number in scientific notation. Had we chosen another way of writing the first component, the second component (the power of 10) would have to be adjusted to keep the overall value correct. For instance, the same value (602400.0) could be written with its significant digits as 6.024. Then, the second component would need to be `10 ** 5` to provide the correct multiplier. Thus, as far as FORTRAN is concerned, there is no one official way for us to express numbers in scientific notation. The forms shown below, i.e.,

```
0.06024*10**7     0.6024*10**6     6.024*10**5
60.24*10**4       602.4*10**3      6024.*10**2
60240.*10**1      602400.*10**0    6024000.*10**-1
```

all are acceptable scientific representations of the same value, i.e., 602400.0. To avoid confusion, it is a good idea to settle on one standard form for expressing the first component of a number in scientific notation. We shall use a zero in front of the decimal point so that our example (602400.0) usually will appear as

```
0.6024*10**6
```

Since it is always a power of *ten* that is being used to adjust the first component of a real constant in scientific notation, FORTRAN permits a more convenient way to write such constants: Using our previous example, the number `0.6024*10**6`, can be expressed with perfect safety as 0.6024E+06, 0.6024E+6, or even .6024E6. In this form, the entire value is written without blanks. (The blanks are not illegal; we prefer not to use them.) Some additional examples are shown in Table 4.1. (Note that when we express a number with an explicit exponent, e.g., `0.6024*10**6`, FORTRAN has to go through some time-consuming calculations to produce the desired value.)

**Table 4.1** Representation of Numbers in Scientific Notation

| *Value* | *Scientific Notation* | *FORTRAN* |
|---|---|---|
| 3 | 3.0*10**0 | -3E0 |
| | | 3.E0 |
| | | 3E+0 |
| | | 3E+00 |
| | | 3.0E+0 |
| | | +3.0E+00 |
| | | 3.00E+00 |
| -.0627 | -6.27*10**-2 | -6.27E-2 |
| | | -6.27E-02 |
| | | -6.270E-02 |
| 3964.27 | 3.96427*10**3 | 3.96427E3 |
| | | +3.96427E+3 |
| | | 3.96427E+03 |
| 265000000000 | 2.65*10**11 | 2.65E11 |
| | | +2.65E+11 |

**4.1.1.4.3 Floating point numbers.** The ability to express real numerical constants in two different forms (scientific notation as well as the basic form) is a convenience built into the FORTRAN language. As far as the machine itself is concerned, the representation of a real number in storage follows a set of standards built into the circuit designs. Any differences between these standards and the forms available to the programmer have to be taken care of by FORTRAN . At this stage, it is not crucial to know exactly how real numbers are stored. However, it is helpful to be aware of the fact that the basic representation is conceptually similar to scientific notation. That is, each real constant is stored as two components which, when multiplied together, produce the proper magnitude. When numbers are represented internally this way, we refer to them as *single precision floating point* numbers. These numbers, which occupy 32 bits of storage in HP computer systems, are declared as `REAL*4`. A `REAL*8` declaration reserves 64 bits, and an addition declaration, `REAL*6`, available in HP1000 FORTRAN 77, reserves 48 bits for a floating point number.

**4.1.1.4.4 Double precision numerical constants.** When a computation is done by hand, there is no serious concern about the number of digits involved in the calculation. For example, if somebody wants to add 0.617398422 and 0.210450386 together, it is just a matter of writing down the numbers and going through the operations. While this is a tedious process, with possibilities for errors, there is no question that it (eventually) could be done for values with any number of digits.

This is not the case when it comes to computers. Each unit of a processor's main storage has a fixed size and, consequently, there is a definite limit on the numerical value that can be accommodated in each storage cell. For the HP processor, this limit is about seven digits. This means that a number like 6274943 (i.e., 0.6274943E+06) or 309.178 (i.e., 0.309278E+03) can be represented by a pair of 16-bit words in the HP1000 or a single 32-bit word in the HP9000 processor. However, a number like 627497.84 (0.62749784E+06) or 0.000087463989 (0.87463989E−04) is beyond the capacity of these representations; there simply is no room for all of it. (Note that we had no difficulty expressing it by hand.) The *magnitude* of the number has nothing to do with it. A very "reasonable" value, say, 1.2749786, causes the same kind of difficulty. The rightmost digit falls beyond the fixed capacity and so it is lost. Since the capacity is fixed, the machines always use all of it. Thus, a real constant such as 23.61, which does not require the full capacity of the 32 bits, still uses it, so that in HP systems it would be represented as 2.361000E+01.

While a representational capacity of seven digits might seem adequate, there are many applications for which this turns out to be a severe (and intolerable) limitation. This has turned out to be a common enough problem so that FORTRAN includes facilities for recognizing another form of real constant known as a *double precision* constant. Having discussed "ordinary" real numbers as single precision floating point constants, we see that the meaning of this new term is obvious: by showing a real constant as double precision, we can force HP FORTRAN 77 to provide 64 bits of storage (i.e., four words in the HP1000 or two words in the HP9000) for that value. The extension thus obtained more than doubles the capacity, making it possible to accommodate about 16 digits of precision.

Double precision constants are written in scientific notation with a D replacing the E. Thus, a real constant expressed as 0.64728E+03 will be stored in single precision floating point form, while the same value, expressed as 0.64728D+03, will be stored in double precision floating point form.

FORTRAN treats single precision as the "normal" way of representing real constants. Consequently, places intended to hold double precision values have to be declared as such. FORTRAN provides separate forms for single and double precision declarations, as we shall see in Section 4.2

**4.1.1.5 Complex Numbers** Many computational applications, primarily in mathematics and engineering, involve the processing of *complex numbers*, i.e., numbers which include an imaginary component. Accordingly, FORTRAN has special facilities for expressing and processing such values. However, since these applications tend to be rather specialized, we shall defer the discussion of complex numbers to the latter part of the text. They are mentioned here only for completeness.

### 4.1.2 Character Constants

Previous use of character constants already has given us an informal idea of their appearance, so that it is necessary only to reinforce these concepts here: a character constant is enclosed in apostrophes, and everything within those apostrophes counts as part of the character constant, including blanks. Thus, for example, the constant `'LOOK HERE'` represents a string of nine characters. If the characters we want to represent include an apostrophe, the constant shows two apostrophes. Thus, the constant `'CAN''T'`, for instance, represents the five characters CAN'T. Note that in the case where we want to represent an apostrophe, the extra one does not count as part of the character constant. This is the one type of exception with regard to the length. Quotation marks (`' '`) count as a single character. Additional examples are shown in Table 4.2.

**Table 4.2** Character Constants in FORTRAN

| *Value* | *Representation as a FORTRAN Constant* | *Length* |
|---|---|---|
| BOOKKEEPING | `'BOOKKEEPING'` | 11 |
| BIG JULIE | `'BIG JULIE'` | 9 |
| BIG  JULIE | `'BIG  JULIE'` | 10 |
| 'TWAS O'ER THE HILLS | `'''TWAS O''ER THE HILLS'` | 20 |
| HE SAID, "I'M HERE!" | `'HE SAID, "I''M HERE!"'` | 20 |

### 4.1.3 Hollerith Constants

As an alternative to character constants, HP FORTRAN 77 also recognizes Hollerith constants. In this form, we specify the character string's length (including blanks) followed by the letter H, followed by the string itself. (Surrounding apostrophes are not used.) An apostrophe in a Hollerith constant is treated as part of the string. Thus, `3HFAR`, `7H32BZ()`, and `5H$32.T` are examples of acceptable Hollerith constants.

Availability of the more convenient character constants makes the use of Hollerith constants obsolete. Accordingly, we mention them here and nowhere else.

### 4.1.4 Logical Constants

A wide variety of computer applications are constructed around (or are helped by) the use of conceptual switches that the program turns on or off to signal certain events or conditions that arise as the program works its way through its activities. FORTRAN makes it convenient to set up these switches, manipulate them, and test their status at any time. Each switch is treated as a *logical* quantity. When a switch is on, its logical status is said to be *true*; when it is off, its status is said to be *false*. Another way to say this is that FORTRAN has two *logical constants*: "true," represented in FORTRAN by `.TRUE.`, and "false," represented in FORTRAN by `.FALSE.`. (The period must be present on both sides of the `TRUE` and `FALSE`.) Just how these logical switches and constants are used will be covered later. The intent here merely is to acquaint you with their existence.

### 4.1.5 Names for Constants = the PARAMETER Statement

There are many times when it is useful to refer to a constant value by a familiar name. For example, the constant 3.14159 is used frequently in a wide variety of computations. However, in *describing* those computations, it is handy to refer to that constant as "pi" instead of having to write the number out each time we want to use it. Such named constants are called *parameters* in FORTRAN 77, which provides a convenient way to define them, i.e., by means of the PARAMETER statement. This statement is constructed as follows:

```
PARAMETER    (NAME = CONSTANT)
```

where "name" is the symbolic name to be used, and "constant" is the value to be associated with the name. (Rules for constructing names appear in section 4.2.1.) For instance, the statements

```
REAL*4          PI
PARAMETER       (PI = 3.14159)
```

set up such an association between the name PI and the real constant 3.14159. Note that the name PI still must appear in a regular declaration. Then, throughout that program, any time the programmer uses that name, FORTRAN will "know" to use the value 3.141599. Thus, assuming RADIUS and AREA are real variables and there is a value for RADIUS, the statement

```
AREA = PI * RADIUS ** 2
```

will be accepted by FORTRAN, and the desired result will be produced.

Several parameters may be defined in a single statement (after suitable declarations). For instance, the statements

```
REAL*4          PI, GRAV
INTEGER*2       NUM
CHARACTER       SIGNAL*4
PARAMETER       (PI=3.14159, GRAV=32.164, NUM=4, SIGNAL='FINI')
```

sets up the real parameters PI and GRAV, the integer parameter NUM, and the character parameter 'FINI'. Thus, a PARAMETER statement placed at the beginning of a program provides a convenient record of such definitions, and it enables the programmer to express the various computations in a more meaningful way. It also is possible to assign a name to a constant expression. For instance, the sequence

```
PARAMETER       (PI = 3.14159, REF_RADIUS = 1.017)
PARAMETER       (REF_AREA = PI * REF_RADIUS ** 2)
```

is perfectly valid.

## 4.2 VARIABLES AND THEIR DECLARATION

When we wish to store data in a particular place and we want to refer to that place by giving it a name, we inform FORTRAN of our intent by setting up a declaration for a *variable*. As seen in previous examples, such declarations are placed at the beginning of a program. In this section we shall add to the list of those already used while providing some more details.

### 4.2.1 Variable Names

FORTRAN has simple rules that govern the construction of variable names:

1. A variable name must begin with one of 26 upper or lower case letters.

**Table 4.3** Variable Names in HP FORTRAN 77

| *Value* | *Status* |
|---|---|
| BIG3 | OK |
| big3 | OK |
| Big_3 | OK |
| J73E | OK |
| net_Value | OK |
| VELOC41 | OK |
| 3w | illegal starting character |
| net value | name contains ineligible character (blank) |
| $PAY | illegal starting character |
| NET-VALUE | name contains ineligible character (-) |
| W34c10 | OK |
| APPARENT_EXIT_VELOCITY | OK, but FORTRAN will use only APPARENT_EXIT_VE |

2. The name may have up to 15 additional characters, and these may be either letters, numerical digits, or break characters (_) in any combination. (HP FORTRAN 77 will accept names of any length, but it will use only the first 16 characters. Consequently, names like DEPARTMENTAL_AVG_GROSSPAY and DEPARTMENTAL_AVG_NETPAY are seen by the compiler as being identical. Standard FORTRAN 77 limits the length of variable names to six characters and does not accept the break character.)
3. These rules also apply to named constants.

Table 4.3 gives some examples of legal and illegal variable names. Regardless of the type of variable being defined, its name must follow these rules. In addition, the programmer must make sure that the names are unique. That is, when a particular variable name is defined, there must be only one variable in that program with that name.

Since a name can be up to 16 characters long, the programmer has ample flexibility to use names that carry some meaning. For instance, if a program is going to compute a numerical value that represents a velocity, it is not particularly helpful to store it in a place called X. With no more effort, the programmer can use a name like VELOCITY.

### 4.2.2 Declaration of Variables

A fundamental aspect of good programming practice is to keep careful records of all the information used in a program and all the variables in which the information is kept. FORTRAN's declaration statements are clear and simple, making it particularly convenient to develop such records. When we declare a variable, we define two things about it:

1. We give it a name.
2. We describe the type of information to be stored in that variable.

Since all of a program's declarations are placed at the beginning of that program, we know exactly where to look to answer any questions about the variables.

#### 4.2.2.1 The INTEGER Declaration

The statement

```
INTEGER     name
```

reserves storage under the indicated name and makes note of the fact that an integer value will be stored there. It should be understood that the declaration, as shown, does

not cause any value to be stored. Thus, when we write

```
INTEGER     UNITNO
```

we are directing FORTRAN to set up the bookkeeping that will enable us to place integer values, one at a time, in a variable named UNITNO. At this point, we do not know what there is in UNITNO. Whatever may be in there is meaningless because our program did not put it there.

Several integer variables may be defined in a single declaration. For example, the statement

```
INTEGER     UNITNO, EMPLOYEE_NUM, NUM_OF_PARTS, H2PCS
```

reserves storage for four integer variables. (We do not know what there is in any of them.)

An integer variable in HP FORTRAN 77 normally occupies 16 bits of storage, thereby accommodating a value that may range from −32768 to +32767. This capacity may be specified explicitly by an alternative form for the INTEGER declaration:

```
INTEGER*2     name
```

An integer variable in HP9000 FORTRAN 77 normally occupies 32 bits of storage, thereby accommodating a value between −2147483648 to 2147483647. It is possible to decrease this range to match that of the normal HP1000 FORTRAN 77 integer variable by using the INTEGER*2 declaration. The variable thus declared for the HP9000 is called a *short integer*. Conversely, the range of an HP1000 FORTRAN 77 integer variable can be increased to match that of the normal HP9000 integer variable by using the INTEGER*4 declaration. In response, the compiler will allocate 32 bits of storage. To avoid unnecessary concern about machine-dependent allocations, we shall use explicit length declarations throughout the text.

**4.2.2.2 The REAL Declaration** Variables designed to hold single precision real values are declared with the statement

```
REAL   name      or      REAL*4   name
```

A real variable in HP FORTRAN 77 is stored in 32 bits (i.e., two words in the HP1000 and one word in the HP9000. We shall use the REAL*4 form.) It can receive a positive value in the range 1.469368E−39 to 1.70141E+38, a negative value in the range −1.70141E+38 to −1.469368E−39, and zero. Except for the type of value that can be stored in a real variable, the REAL declaration operates exactly like the INTEGER declaration.

**4.2.2.3 The DOUBLE PRECISION Declaration** Double precision real values are stored in variables declared with the statement

```
DOUBLE PRECISION   name      or      REAL*8   name
```

This motivates the HP FORTRAN 77 compiler to allocate 64 bits of storage (i.e., four words in the HP1000 and two words in the HP9000). This corresponds to a capacity (in the HP1000) of positive values ranging from 1.46936793852785938D−39 to 1.70141183460469227D+38, negative values ranging from −1.70141183460469232D+38 to −1.46936793852785946D−39, and zero. The corresponding capacity in the HP9000 is about 2.2D−308 to 1.8D+308, −1.8D+308 to −2.2D−308, and zero, with about 16 digits of precision. Here again, as is true for the INTEGER and REAL declarations, it is possible to declare several variables with one DOUBLE PRECISION statement. (We shall use the REAL*8 form to avoid machine-dependent complications.)

**4.2.2.4 The CHARACTER Declaration** Strings of characters are stored in variables declared as follows:

```
CHARACTER*length      name
```

The *length part of the declaration tells FORTRAN how long a string (i.e., how many characters) the programmer expects to store in that variable, and FORTRAN reserves the appropriate amount of storage. For example, if we write

```
CHARACTER*8      TITLE2, PARTNAME
```

FORTRAN will reserve sufficient storage for two variables TITLE2 and PARTNAME, each with a capacity of eight characters. The amount of storage required to hold one character is called a *byte*, so another way of saying this is that FORTRAN reserves two eight-byte areas of storage in response to the declaration given above. Variables of unequal length can be declared in the same statement. For instance, the statement

```
CHARACTER*8      TITLE2, PARTNAME, MODEL*6
```

reserves storage for two eight-character variables (TITLE2 and PARTNAME) and a six-character variable named MODEL. A more complicated example, i.e.,

```
 CHARACTER*7     SET_WORD, LAST_WORD, SHORT_WORD*6,
1                LONG_WORD*10, STANDARD_WORD
```

reserves storage for five character strings: SET_WORD, LAST_WORD, and STANDARD_WORD, each of which is seven characters long, SHORT_WORD with a length of six, and LONG_WORD with a length of ten. As is true with any other declaration, no actual values are stored.

**4.2.2.5 The LOGICAL Declaration** Logical variables are declared by the statements

```
LOGICAL NAME    or    LOGICAL*2 name    or
LOGICAL*4 name
```

LOGICAL*2 reserves 16 bits of storage, and LOGICAL*4 reserves 32 bits of storage. In either case, storage thus reserved will contain a variable whose value, when assigned, will be either .TRUE. or .FALSE.. (The LOGICAL declaration, without an explicit length indicator, reserves 16 bits in the HP1000 or 32 bits in the HP9000. Hence, we shall use the explicit length indicator with our logical declarations.) The extended storage allocation for logical variables does not change their behavior. It is included as a convenience in setting up certain common storage organizations, to be discussed later.

**4.2.2.6 A Note for Old Times' Sake** We started examining the declaration statements with the idea that a systematic record of all variables is a basic part of a well-written program. Even though this sounds like an obvious notion, its importance was not fully realized until well after the initial versions of FORTRAN were produced and well established. Consequently, there are features based on the idea that if we use a variable without explicitly defining it, FORTRAN can (and should) determine from the usage that this is a new variable which requires storage and appropriate bookkeeping support. These features make it possible to assign a value to an undeclared variable or to read a value into an undeclared variable. In either case, FORTRAN was designed to provide a declaration if the programmer did not do so. In order to make it possible for FORTRAN to produce such an "automatic declaration," certain assumptions had to be made about the type of variable to be declared. In earlier times, variables in FORTRAN were either INTEGER or REAL, so that the following rules were defined:

1. Use of an undeclared variable whose name began with the letters I, J, K, L, M, or N would force FORTRAN to reserve storage under that name for an integer variable.

2. Use of an undeclared variable whose name began with any other letter (i.e., A through H or O through Z) would force FORTRAN to reserve storage under that name for a real variable.

These naming rules, called *defaults*, could be overridden, of course, by an explicit declaration. Moreover, an additional feature (the IMPLICIT declaration) could be used to change the defaults so that an automatic real or integer declaration would apply to first letters other than the ones used by the built-in rules.

These features, and the rules supporting them, still exist in FORTRAN 77 so that programs written under earlier versions of the language can be accepted by the new standard without change. In fact, they have been expanded so that the programmer can associate certain letters with double precision, character, or logical variables as well. In this book we shall avoid the use of undeclared variables. To help emphasize the need to declare all variables, we shall include the statement

```
IMPLICIT      NONE
```

in our programs. As pointed out earlier, this forces FORTRAN to produce a warning message calling our attention to implicitly declared variables. (The tendency in newer programming languages is to design them so that undeclared variables are illegal.) The IMPLICIT feature is described in Appendix C for reference purposes, should you find it necessary to read an old FORTRAN program.

### 4.2.3 Alternate Names for Variables — The EQUIVALENCE Statement

There are many situations where it is convenient to refer to a particular variable by more than one name. For example, the value stored in that variable might mean one thing during part of a program, and then it might take on a different meaning in another part of the program. FORTRAN makes it possible to provide variables with such aliases by use of the EQUIVALENCE statement. We shall introduce it here and explore its various uses later on, as opportunities arise to take advantage of it.

Different names are associated with the same variable by using an EQUIVALENCE statement in conjunction with a normal declaration. For instance, suppose we define variables named NUMOBS and COUNTS with the ordinary statement

```
INTEGER*2     NUMOBS, COUNTS
```

Without any further specifications, FORTRAN will reserve a separate storage location for each variable. Now, if we follow the previous statement with

```
EQUIVALENCE     (NUMOBS, COUNTS)
```

only one location will be reserved, and we shall be able to refer to it using either name. this capability is not limited to two names. For example, the statements

```
REAL*4      SIZE, WEIGHT, BULK, MASS, LBS, KILOS
EQUIVALENCE     (SIZE, BULK, MASS)
```

will produce reservations for four separate real variables: Three of these will have single names (i.e., WEIGHT, LBS, and KILOS) while the fourth will be associated with any of three names (SIZE, BULK, and MASS). More than one such relationship can be set up in a single statement. Using the same declaration as above, we can write

```
REAL*4      SIZE, WEIGHT, BULK, MASS, LBS, KILOS
EQUIVALENCE     (SIZE, BULK, MASS) , (LBS, KILOS)
```

in which case there will be three distinct variables: One will have the exclusive name WEIGHT, one will have three names (SIZE, BULK, MASS), any one of which can be used, and one will have the two names LBS and KILOS.

It is possible to list variables of different types in the same EQUIVALENCE group. For instance, the sequence

```
REAL*4          RVAR
INTEGER*2       WHOLENUM
EQUIVALENCE     (RVAR, WHOLENUM)
```

is legal, resulting in the use of the same storage for either variable. When the program refers to RVAR, the value in that location is treated as a real number; a reference to WHOLENUM compels the program to view the contents of that same location as an integer. This feature must be used carefully since variables of different types do not necessarily occupy the same amounts of storage. HP FORTRAN 77 uses the following rule for EQUIVALENCE lists:

Each data item in an EQUIVALENCE list has the same first storage unit.

In the example just given, the declaration for RVAR reserves 32 bits of storage while WHOLENUM, not being declared as a double integer, occupies only 16 bits. If we apply the rule stated above, the relationship will be as follows:

```
    RVAR
WHOLENUM   not used for   WHOLENUM
```

These considerations will become more essential when we set up aliases for arrays.

## 4.3 INITIALIZATION OF VARIABLES

When we discussed the declaration of variables, it was emphasized that we did not (and could not) know anything about the contents of a newly declared variable until we placed a value in it. Because of this fact, it is a good idea to place a known value in each variable before we actually put that variable to its intended use. By doing that, we make sure that we have complete control over the variable from the beginning. If the value changes, we know that it changed because we know what the original value was. For the same reason, we know when a value does *not* change. This process of explicitly providing a starting value is known as *initialization*. We saw an example of initialization in Example 3.3 of the previous chapter, where we set the total area TLAREA to zero before we began processing the first set of input data. This gave us a base value to which we could add the individual areas as they were computed.

Of course, the specific reason for initializing a particular variable will depend on the type of variable and its usage. Similarly, there is no standard initial value that is assigned. Many numerical variables are initialized to zero, but that is not true for all numerical variables. Along the same lines, it often is appropriate to initialize a character variable so that it contains all blanks, but that need not always be the case. As our skill with FORTRAN develops, and we tackle more ambitious programming problems, initialization will play an increasing role.

FORTRAN recognizes the importance of initialization, and it provides convenient methods for handling it. The most systematic way of defining initial values is by means of the DATA statement. This statement, placed at the beginning of the program along with the other declarations, has the form

```
DATA  name/value/
```

FORTRAN will produce the value (a constant) given between the slashes, and it will store that value in the variable indicated by the name. For example, the statements

```
INTEGER*2  WIDTH
DATA  WIDTH/267/
```

will reserve storage for an integer variable named WIDTH, and an initial value of 267 will be placed in that variable.

Several variables may be initialized in a single DATA statement. To illustrate, let us consider the following sequence of statements:

```
REAL*4          INCOME, ADJUST
INTEGER*2       CASES, SWITCHSET
INTEGER*4       MEMORY_IMAGE
CHARACTER*6 WORD
DATA    INCOME/0./, ADJUST/0./, CASES/0/,
1 SWITCHSET/O'177777'/, MEMORY_IMAGE/Z'0F0FF0F0'/, WORD/'ZZZZZZ'/
```

As a result of this sequence, INCOME, ADJUST, and CASES will each contain an initial value of zero, WORD will contain six Z's, SWITCHSET will contain an integer equivalent to the octal value O'177777', and MEMORY_IMAGE will contain an integer equivalent to the hexadecimal value Z'0F0FF0F0'. Note that INCOME and ADJUST, having been declared as real variables, were initialized with real constants (0. in this instance); CASES, being an integer, was initialized with an integer value. FORTRAN also permits the use of several DATA statements. Thus, the version

```
DATA    INCOME/0./, ADJUST/0./, CASES/0/
DATA    SWITCHSET/O'177777'/, MEMORYIMAGE/Z'0F0FF0F0'/, WORD/'ZZZZZZ'/
```

may appear less cluttered than the previous version.

Another way of writing the DATA statement enables the programmer to place all the names together, followed by their respective initial values. Using the same variables as above and assuming the same declarations, we can rewrite the DATA statement as follows:

```
 DATA    INCOME, ADJUST, CASES, SWITCHSET, MEMORY_IMAGE, WORD
1    /0.,0.,0,O'177777',Z'0F0FF0F0','ZZZZZZ'/
```

The result is exactly the same as before: INCOME and ADJUST are initialized to 0, CASES is initialized to 0, and WORD is initialized to ZZZZZZ.

If several variables are to be initialized to the same value, a more concise form of the DATA statement may be used. In the previous example, two variables (INCOME and ADJUST) were initialized to the same real value. We can specify this fact by writing

```
DATA    INCOME, ADJUST, CASES, SWITCHSET, MEMORY_IMAGE, WORD
        /2*0.,0,O'177777',Z'0F0FF0F0','ZZZZZZ'/
```

The 2* in front of the zero is a *repetition factor* that indicates how many variables are to be initialized to that value.

In order for the DATA statement to work properly, there must be a match between the number of items (variables) in the list and the number of initial values between the slashes. Since a DATA statement may be as long as any other FORTRAN statement, there is capacity for a long list. Sometimes it is easier for the programmer to keep track of this matching process by breaking the list into several smaller ones. This also can be done in the DATA statement. Using our previous example, we can rewrite the DATA statement as follows:

```
 DATA    INCOME, ADJUST/2*0./, CASES/0/,
1        SWITCHSET, MEMORY_IMAGE/O'177777', Z'0F0FF0F0'/,
2        WORD/'ZZZZZZ'/
```

Additional features are available for more elaborate initialization processes. We shall examine these later, specifically when it comes to setting up entire tables of values.

## 4.4 SUMMARY

The basic types of data used in HP FORTRAN 77 include:

1. Integers or double integers (positive or negative numbers without fractions) where 348, −90, O'3751', −274B, and Z'C60B' are examples of integer constants.

2. Real numbers (also called single precision floating point numbers) (positive or negative numbers with possible fractions) where −7.7, 348., 4.796E−5, and 0.089 are examples of real constants.
3. Double precision numbers (positive or negative numbers with extended capacity for fractions) where 0.87456D3, 627.40572841, and −23.358790083000D−4 are examples of double precision constants.
4. Character strings (sequences of letters, numerical digits and special symbols in any combination) where 'X23J7', 'PARCELS7' and '−003 %J' are examples of character constants.
5. Logical values (either .TRUE. or .FALSE.).

FORTRAN variables are declared at the beginning of a program by specifying a variable name (selected by the programmer) and the type of data the variable will contain. Declarations for the type of data listed above take the following form:

INTEGER name (equivalent to INTEGER*1 for the HP1000 and INTEGER*4 for the HP9000)
REAL name (equivalent to REAL*4)
REAL*6 name (for the HP1000)
REAL*8 name or DOUBLE PRECISION name
CHARACTER*length name or CHARACTER name*length
LOGICAL name (equivalent to LOGICAL*2 for the HP1000 and LOGICAL*4 for the HP9000)

Variables declared as indicated above may be given initial values by means of the DATA statement using one of several basic forms:

```
DATA name/value/,name/value/,.....,name/value/
```

or

```
DATA name,name,name,...,name/value,value,value.,,,.value/
```

or

```
DATA name,name,...,name/value,value,...,value/,name/value/
```

**PROBLEMS**

1. Indicate which of the following is a legal FORTRAN integer constant. If it is not, give the reason:

| | | |
|---|---|---|
| (a) 22 | (e) +4142 | (i) 3127B |
| (b) −761 | (f) 6 21 | (j) O'−21457' |
| (c) 20. | (g) −7. | (k) O'394' |
| (d) −2,46 | (h) 0646 | (l) Z62AE |

2. Indicate whether each of the following is a legal FORTRAN real constant. If it is not, give the reason:

| | | |
|---|---|---|
| (a) −23.6 | (h) 3.27*10**4 | (o) −2.0E−02 |
| (b) −23 | (i) −2.0*10**2 | (p) .7*10**2 |
| (c) 202 | (j) 4*10**3 | (q) +.23E+02 |
| (d) 3,200,000.0 | (k) 228.E2 | (r) +0.2E+ |
| (e) −0 | (l) −2.2E−01 | (s) −0.1E−01 |
| (f) 81.61 | (m) 0.E0 | (t) .0E+03 |
| (g) −81. | (n) 7.1E7.1 | (u) 212.12E−12 |

3. Convert each of the following numbers to scientific notation. Express your results in standard form (i.e., with zero to the left of the decimal point):

| | | |
|---|---|---|
| (a) 44.2 | (e) −313.313 | (i) .0000000786 |
| (b) −17 | (f) 46,324,000 | (j) 6756.44 |
| (c) 2.00084 | (g) −1.00092 | (k) 845904.1 |
| (d) 0.0 | (h) −93200000000000 | (l) 0.000000000000004174 |

4. Convert each of the following numbers to conventional notation:

| | | |
|---|---|---|
| (a) 6.3*10**2 | (e) −244.65*10**−8 | (i) 0.00000E0 |
| (b) 2.11E2 | (f) −.000786E+04 | (j) 0.868E−06 |
| (c) 0.868E6 | (g) 211.0E0 | (k) 89675.4*10**4 |
| (d) −2.E8 | (h) 61673.8E5 | (l) 0.00021100E+06 |

5. Arrange each set of numerical constants so that they are in ascending order (i.e., smallest one first):

| | | |
|---|---|---|
| (a) 22.7 | −38.14 | 3.64*10**1 |
| (b) −14.61 | −0.2416E01 | 0.21621E−1 |
| (c) 3.E3 | 03.0E−1 | 2.17E0 |
| (d) 4.87*10**2 | .6644E03 | .06628E04 |
| (e) −3.71E−1 | −37.28E−3 | −.2898E−1 |

6. Write each of the following in FORTRAN floating point notation using the standard form from Section 4.1.1.2.2 with six decimal places:

| | |
|---|---|
| (a) 327.16 | (e) 887.321*10**−8 |
| (b) −3 | (f) 1000.0*10**−3 |
| (c) 2.2*10**6 | (g) .00007482 * 10 ** 2 |
| (d) .000714 | (h) −891674*10**4 |

7. Express each of the following as a FORTRAN double precision floating point constant using 14 decimal places:

| | |
|---|---|
| (a) −862.186 | (f) 1421.1421 |
| (b) 8562478.06 | (g) 11.7E8 |
| (c) 26.OE−4 | (h) −1.28435947 |
| (d) −.00007241551515 | (i) 7.9E17 |
| (e) 1.E0 | (j) −38.388E−12 |

8. Express each of the following strings as a character constant and indicate its length. A blank is represented by b:

| | |
|---|---|
| (a) BELL | (d) RIGHT-LEFT |
| (b) AbCHICKEN | (e) 'TWASN'TbTHERE |
| (c) bbCENTERbbLEFTb | (f) ''IbCAN'TbLOSE,''bSHEbSAID. |

9. Write PARAMETER statements for each of the following:

| | *Variable* | *Type* | *Value* |
|---|---|---|---|
| (a) | V | real | 17.2 |
| (b) | BASE | integer | 2 |
| | MINVAL | real | −0.8 |
| (c) | FILLER | character | bbbbbb |
| | WVAL | real | −8.84 |
| | LNGVAL | double precision | 12.8486729 |
| | YVAL | real | 360000000 |
| | WD1 | character | GG'6'ST |
| (d) | FACT | logical | true |
| (e) | BKGRD | character | $$$$$ |
| | ZERO | integer | 0 |
| | NOTHING | real | 0.0 |
| | INITL | integer | 1 |
| | OHNO | logical | false |
| | HITEST | real | 999.99 |
| | BLANKS | character | bbb.bbb |

10. Write FORTRAN declarations for each of the following:
    (a) Real variables XUP, XDOWN, XSIDE, and integer variable WIDTH.
    (b) Real variables YRT, YLFT, character variables WVERB (length 10), WNOUN (length 8) and WADVB (length 10).

(c) Double precision variables BIGM and BIGT, real variable WRL, and integers NUMX, NUMY, and NUMT.
(d) Character variables STRP, STRN and STRY (all with length 7), integer variables TX and TY, double precision variable VLONG, logical variables SW1 and SW2, and real variables FILLA and FILLB.
(e) Logical variable FLIP, double logical variable FLIP_2, character variable WD (length 9), real variables XTHEOR and XACTL, integer variable COUNT, double integer variable BIGVAL, and character variables PRFX and SFX, each with length 4.

11. For each of the following sequences of statements, specify how many different locations FORTRAN will reserve in the particular HP system with which you are working. Give the name or names associated with each location:

```
(a) REAL          HT, WT, LGTH, ALT
(b) REAL          HT, WT, LGTH, ALT
    EQUIVALENCE   (HT, ALT)
(c) INTEGER*2     COUNT, AMT, TOTAL
    EQUIVALENCE   (COUNT, AMT, TOTAL)
(d) REAL          BEV, DRK, QUF, LIQ
    INTEGER*2     QTS, VOL, CAP
    CHARACTER*8   DSCR, NAME, APPL
    EQUIVALENCE   (QTS, CAP), (DRK, BEV), (DSCR, APPL)
(e) REAL*8        LNGVAL, EXTNS, BILDUP
    REAL          XWAY, YWAY, ZWAY, HORIZ, VERT, FRBACK
    INTEGER       TALLY, TOTAL, SUMALL, PRTSUM
    CHARACTER*7   ENDNGS, PRFXES, SFIXES, WORDM
    LOGICAL       TURN, PIVOT, HINGE, SWL, SW2
    EQUIVALENCE   (TURN, HINGE), (SW1, SW2), (XWAY, HORIZ)
    EQUIVALENCE   (LNGVAL, BILDUP), (XWAY, ZWAY)
```

12. Write FORTRAN statements for each of the following:

(a) Declare real variables AVAL, BVAL and CVAL with all three referring to the same location.
(b) Declare real variables XVAL, HVAL, YVAL and VVAL with XVAL and HVAL referring to the same location, and YVAL and VVAL referring to another location.
(c) Declare real variables TMIN, TMED and TMAX, integer variables CTMIN, CTMED and CTMAX, and character variables WDMIN, WMDED and WDMAX (all with length 7). TMED and TMAX refer to the same location, CTMED and CTMIN refer to the same location, and all three character variable names refer to the same location.
(d) Reserve a location for a real variable whose name could be either BWAY, VWAY, XWAY or ZWAY; an integer variable named JMAX, another integer variable named either JM or OVRNUM, and a logical variable named SW1, HINGE, POINT, FLIP, or MAYBE.

13. For each of the following sequences of statements, show what the resulting output would be. Assume that real values are printed in standard floating point form with six decimal places:

```
(a) REAL*4          S, A, VO, T
        VO = 25.0
        A = 10.0
        T = 2.0
        S = VO*T + 0.5*A*T**2
        PRINT *, VO, A, T, S
(b) REAL*4          S, A, VO, T
        V0 = 0.5E2
        A = 1.2E1
        T = 2.E-01
        S = VO*T - 0.5*A*T*T
        PRINT *, VO, A, T, S
(c) REAL*4          VO, T, S
        PARAMETER (ACCEL = 5.0)
        DATA  VO/10.0/
        T = 2.0
        S = T * (VO + 0.5*ACCEL*T)
        PRINT *, V0, T, S
```

(d)
```
REAL*4              V0, T, S
    PARAMETER (GRAV = 32.164)
    DATA   V0/0.0/, T/100.0/
    S = T * (V0 + 0.5*GRAV*T)
    PRINT *, V0, T, S
```

(e)
```
REAL*4              CIRCUM, AREA, VOL
INTEGER*2           RADIUS, LOOPS
PARAMETER           (PI = 3.14159)
RADIUS = 1
DO 14 LOOPS = 1, 4
   CIRCUM = 2 * PI * RADIUS
   AREA = PI * RADIUS ** 2
   VOLUME = 1.333333 * PI * RADIUS ** 3
   PRINT *, RADIUS, CIRCUM, AREA, VOLUME
   RADIUS = 2 * RADIUS
14 CONTINUE
```

(f)
```
REAL*4              BIGX, BIGY, SMLX, SMLY, XMAX, YMAX, XMIN, YMIN, Z
INTEGER*2           POWER, EXPO, NUM, COUNT
PARAMETER           (BASIS=2, COEF=0.955, ADDER=6.6)
EQUIVALENCE         (BIGX, XMAX), (SMLX, XMIN)
    DATA   BIGX, BIGY, SMLX, SMLY/1000.0, 2000.0, 2*1.0/
    YMAX = 5000.0
    YMIN = 2 * XMIN
    PRINT *, BIGX, BIGY, XMAX, YMAX
    PRINT *, SMLX, SMLY, XMIN, YMIN
    Z = 3 * (BIGX - SMLX) + 2 * (XMAX - XMIN)
    PRINT *, Z
    Z = Z + COEFF* (SMLX+SMLY) ** BASIS + ADDER
    PRINT *, BASIS, ADDER, Z
    BIGX = 0.5*BIGX
    XMIN = XMIN+SMLX
    PRINT *, XMAX, SMLX
```

14. Write a FORTRAN program using the diagram or pseudocode developed for Problem 8 in Chapter 2. Run your program using the following set of input values:

    DEN = 11.50.0 ksm/cubic meter
    DIAM = 0.03175 meters
    VEL = 24.384 meters per second

15. Write a FORTRAN program from the N-S diagram or pseudocode developed for Problem 9 in chapter 2. Run your program using the following input values:

    density = 44.95 lbs./cubic foot
    diameter = 0.875 inches
    velocity = 114.5 miles per hour

16. Modify the program in Problem 15 so that it will process any number of input sets. Use a density value of zero to terminate the run.

17. Write a FORTRAN program based on the diagram or pseudocode developed for Problem 10 in Chapter 2. Run your program with the following input data:

    LENGTH = 12.375
    WIDTH = 8.4375

18. Modify the program in Problem 17 so that it will process any number of input sets. Use a length (LTH) or zero to terminate the run. After the set of results has been printed, the program is to print summary information showing the number of input sets processed, the total area ordered, the total price (without tax), and the total price (with tax).

19. Modify the program in Problem 18 so that the price per square foot may be changed for each run (where a run consists of any number of input sets).

20. Modify the program in Problem 18 so that the price per square foot may be changed any number of times within a single run. That is, set up your algorithm design so that the user can specify a certain price per square foot. Then, the program will use that price for every input order until a new price is specified. The design is to be completely flexible, so that a new price may appear after only one order at the previous price, or it may appear after several hundred orders at the previous price. (*HINT:* Design your data flow so that there is a special line (that your program can recognize and distinguish from an order) that gives a new price.)

21. Write a FORTRAN program to implement the procedure designed for Problem 20 in Chapter 2. Run your program using the following input values:

| *LENGTH* | *WIDTH* |
|---|---|
| 400.0 | 100.0 |
| 410.0 | 122.0 |
| 414.5 | 114.6 |
| 200.0 | 414.8 |
| 100.5 | 56.2 |
| 0.0 | 0.0 |

22. Write a FORTRAN program to implement the procedure designed for Problem 25 in Chapter 2.

23. Write a FORTRAN program to implement the procedure designed for Problem 29 of Chapter 2.

24. Write a FORTRAN program that implements the procedure designed for Problem 30 in Chapter 2.

25. Design and write a FORTRAN program that computes the equivalent weight in kilograms for 1 pound, 2 pounds, and so on, up to and including 25 pounds. The printout is to show a weight on each line, similar to the form illustrated below:

| *WT. LBS* | *WT. GMS* | *WT. KGMS* |
|---|---|---|
| 1 | 0.453600E+03 | 0.453600E+00 |
| 2 | 0.907200E+03 | 0.907200E+00 |
| . | .... | .... |
| * | etc. | |

# 5
# *Computations*

FORTRAN's greatest strength lies in the ease with which we can specify computations. The name of the language (recall that it is an abbreviation for FORmula TRANslation) emphasizes a major objective: To enable programmers to write down a formula (simple or complicated as it may be) and have the computer set up and do the necessary arithmetic (or algebra, or calculus, or whatever) automatically. In this chapter we shall examine FORTRAN's computational features and confirm our suspicions that they are really convenient and powerful.

## 5.1 THE ASSIGNMENT STATEMENT

Just about everything we want to do in the way of computation is specified by means of the assignment statement. Since we have used these statements in earlier examples, we need only review the general form before examining the computational features in more detail:

variable = expression

By now we should be used to the fact that although an assignment statement looks like an equation, it really is not. It is a directive to do the following work:

1. Perform the computations described by the expression to the right of the = sign. Successful performance will produce a single result, i.e., some numerical, character, or logical value.
2. Place that value in (assign that value to) the variable named on the left of the = sign. This will replace whatever was in that memory location before.

What is meant by successful performance? In order for the calculations to be done properly, they must be specified *correctly* and *clearly*. The first of these requirements makes it necessary for the programmer to write the expression using those ingredients that FORTRAN is designed to recognize. The second requirement is fulfilled by making sure that the expression says what the programmer wants to say.

These requirements can be met by learning and following a few simple rules. In most cases the rules are "natural." They "force" us to do what we would have done anyway. However, we need to understand the rules so that we can make them work for us under all circumstances.

### 5.1.1 Construction of an Expression

An arithmetic expression in FORTRAN is not terribly different from an arithmetic expression written down for calculation by hand. There are certain ways in which we have to change the physical appearance to get around some of the computer's limitations, but these changes are minor, and we can get used to them quickly. Specifically, we must recognize that the computer can handle only one line at a time; it does not have our ability to look at something described in several lines and take it all in at once. For example, we

have no problem at all in reading and understanding something like

$$Y = \frac{A}{B} + \frac{C}{D}$$

To us, this is a simple formula; to a computer, it is hopeless. As a result, we must present the expression so that FORTRAN can "look" at it as a single line. For the little example given above, we would write

```
Y = A/B + C/D
```

A second type of change in the form of an expression is due to the fact that the computer does not know the rules of algebra. (As we pointed out in the first chapter, it does not "know" *anything*.) Consequently, it cannot recognize the shortcuts and conveniences that we have learned. For instance, if we decide to use variables named A, B, and C, we have no trouble understanding

$$Y = \frac{AB}{C}$$

Obviously, this means, "Y is equal to A times B divided by C." Furthermore, knowing what we know about the assignment operation in FORTRAN, it ought to be possible (one might think) to rewrite the abovementioned formula as an assignment statement, namely,

```
Y = AB/C
```

which ought to mean, "Produce the value A times B divided by C, and assign that value to Y, replacing whatever was there before." However, it does not work out that way. FORTRAN operates under the assumption that we mean to divide some variable named AB by a variable C and store the result in Y. Consequently, we have to be explicit about everything we want to do. That is why every multiplication must be denoted by an asterisk (*). Table 5.1 shows these rules applied to some additional examples.

**5.1.1.1 Operators and Operands** Every arithmetic expression consists of some combination of two types of terms: *operators* and *operands*. Operators specify what kind of computation is to be done, and operands indicate which information is to be used in doing the work. For instance, suppose we have declared the real variables VELOC, ACCEL, and TIME. Then, if we wrote the expression

```
VELOC * TIME + 0.5 * ACCEL * TIME ** 2
```

that expression would consist of five operators (three multiplications, an addition, and an *exponentiation*, as the ** operator is called), and six operands: The four variables

**Table 5.1** Simple Arithmetic Expressions in FORTRAN
Variables are A, B, C, D, E, F

| *Conventional Form* | *FORTRAN Form* |
|---|---|
| $A + \frac{B}{C}$ | `A + B/C` |
| $\frac{AB}{C} = \frac{D^2E^2}{F}$ | `A*B/C - D**2*E**2/F` |
| $\frac{A^{2.5}B}{C^2} + \frac{C}{\frac{D}{E}}$ | `A**2.5*B/C**2+C/D/E` |
| $\frac{A^3B^{.7}C^{.2}}{D^{.4}}$ | `A**3*B**.7*C**.2/D**.4` |

(VELOC, ACCEL, and TIME, the latter appearing twice), and the two constants 0.5 and 2. If we want to store the result of these computations in a real variable named DIST, for example, we would convert the expression into an assignment statement by including an assignment operator and a destination variable (i.e., a place to store the result):

```
DIST = VELOC * TIME + 0.5 * ACCEL * TIME ** 2
```

**5.1.1.2 Types of Arithmetic Operators** As indicated earlier, FORTRAN recognizes five arithmetic operators. Three of them are always *binary* operators. In order to use a binary operator correctly, it must appear between two operands. These three operators are multiplication (×), division (/), and exponentiation (**). If we use the names *opnd1* and *opnd2* to indicate two operands, then we can summarize these three basic activities as follows:

*opnd1 * opnd2: opnd1* is multiplied by *opnd2*
*opnd1/opnd2: opnd1 is divided by opnd2.*
If we wanted to divide *opnd2* by *opnd1* we would have to write *opnd2/opnd1.*
*opnd1 ** opnd2: opnd1* is raised to the power indicated by *opnd2.*

Each of the other two operators (+ and −) may have one of two somewhat different meanings, depending on how it appears in an expression. When used as binary operators (i.e., between two operands), + and − mean what one would expect:

*opnd1 + opnd2: opnd1* is added to *opnd2*
*opnd1 − opnd2: opnd2* is subtracted from *opnd1.*

The + and − operators also may appear as *unary* operators. An activity performed by a unary operator uses only one operand, and the operator is placed in front of (to the left of) that operand.

When − is used as a unary operator the activity is called *negation.* It amounts to multiplication by −1:

*−opnd1:* negate *opnd1*

For instance, the assignment statement

```
Y = -X
```

contains a simple expression consisting of the single operand X and the unary negation operator −. It says, "Negate the value in X and store the result in Y." If X happens to contain 17.2, this statement produces a value of −17.2 in Y.

When + is used as a unary operator, it refers to an *identity*:

*+opnd1:* same as *opnd1*

For example,

```
Y = +4.27
```

uses + as a unary operator which "acts on" the single operand 4.27. (Use of the unary + is rather limited, and we are unlikely to encounter it elsewhere in the book.)

**5.1.1.3 Simple and Complicated Arithmetic Operands** In their final form (that is, when FORTRAN gets through with them), arithmetic operands are numbers. When, for instance, we write an expression like

```
VELOC * TIME
```

we are directing the computer (through FORTRAN) to multiply together two numbers to be found in the designated variables (i.e., in VELOC and TIME). In this case, the directive is simple: Each of the operands taking part in the multiplication comes directly from a variable. When we describe a more elaborate set of computations, the operands for a

particular operator may themselves be results of other computations described as part of the same expression.

For example, let us go back to an earlier statement:

```
Y = A/B + C/D
```

and look at the + operator. Here, it represents a binary operator (addition) since it clearly is placed between two operands. But what are these operands? Which two numbers will the computer be adding together? The answer is what we would expect: One of the two operands is the number produced by dividing A by B, and the other is the number produced when C is divided by D. Thus, there is no single form for an operand in a FORTRAN expression. It may be as simple as a constant or variable, or it may itself be an arithmetic expression. In our example, A/B and C/D both are arithmetic expressions whose results, when computed, will serve as operands for the addition.

The basic point of this discussion is that we need to know exactly how FORTRAN does its computations so that when we write an expression in which computed results become operands for further computations, we can be sure we get what we want. In the next section we shall look at the use of parentheses as a way of providing such computational guarantees. Then, in a later section, we shall examine FORTRAN's internal rules for computations so we can see in detail exactly why FORTRAN computes the way it does.

**5.1.1.4 Parentheses in Arithmetic Expressions** As seen in the previous section, we can write expressions consisting of smaller expressions. When this is done, the physical appearance does not always give a clear picture of what FORTRAN will do. We do not have to look for an exotic series of computations to see this: A relatively simple bit of arithmetic will make the point clear.

**Example 5.1** The president of Plotz Cookware, Ltd. ("A Plotz Pot Will Keep It Hot") has a strong-minded wife who has a brother with a degree in employee testing. As a result Plotz is heavily dedicated to systematic testing of its workers. Each person to be tested is brought into a separate little room containing five numbered chairs, and is given a written test. While he or she is taking the test, a Specially Trained Observer records the time required to complete the test (TSTIME) as well as a Fidgibility Index (FGNDEX) based on the amount the person moves around in the chair. Later on, the test score (TSCOR) is used together with TSTIME, FGNDEX, and the chair number (CHNUMB) in which the person sat to compute an overall Employee Adjustment Rating (EARTNG) using the formula

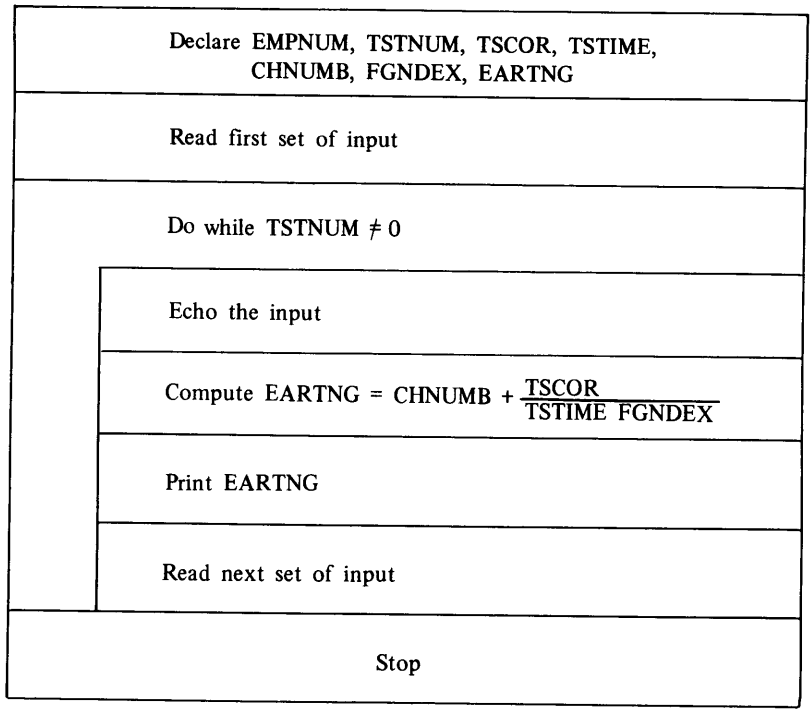
$$\text{EARTNG} = \text{CHNUMB} + \frac{\text{TSCOR}}{\text{TSTIME}\ (\text{FGNDEX})}$$

Now, with business expanding rapidly, a FORTRAN program is needed to automate the computations for the many people being tested.

The flow of events is simple enough: Each set of test results will be read from a separate card which also will contain the person's name (EMPNAM) and the test number (TSTNUM). For this example, we shall assume that we do not know how many sets of test results will be submitted for any given run. Consequently, we shall use *dummy input* to stop the run. This dummy input consists of a set of data that we know beforehand to be "illegal" because we define it that way. Then, by placing that set at the end of the regular data, we can test for it in our program. As long as we do not run into the dummy data, the program "knows" to continue processing. As soon as the dummy data have been read, the program is designed to stop. It does not matter what kind of value we use for this purpose, as long as we make sure that this value cannot possibly occur in the actual data. For this example, as was the case in several earlier ones, we shall say that no test ever will have a test number of zero, so that we can use that value in the dummy data.

When it is convenient to use this technique, it is almost always possible to find an impossible value. For instance, if one of the variables in a particular program represents a weight, a negative value (say −1) would serve perfectly well to signal the end of the data. (Later on we shall learn other ways of providing such signals.)

With this in mind, we can see that our program structure is nothing more than a loop that repeatedly keeps reading data and testing for the dummy value. And, as long as it is not found, the program computes EARTNG and prints the results. This is seen in the flow diagram and pseudocode in Figure 5.1.




(a)

"Define variables EMPNUM, TSTNUM, TSCOR, TSTIME, CHNUMB, FGNDEX, EARTNG."
"Read first set of input."
**while** TSTNUM is not equal to zero:
"Echo the input."
"compute EARTNG = TSCOR / TSTIME*FGNDEX"
"Print EARTNG."
"Read the next input set."
**endwhile**
"Stop."

**(b)**

**Figure 5.1** **(a)** Structured Flowchart for Example 5.1. **(b)** Pseudocode for Example 5.1.

The heart of the program is the assignment statement in which the required value is computed and stored in EARTNG. Referring back to our formula, a first glance might persuade us to write the statement as

```
EARTNG = CHNUMB + TSCOR/TSTIME*FGNDEX
```

This might appear to be reasonable, especially since it does not look terribly different from the formula. However, because of the order in which FORTRAN performs these calculations, the result will not be the one we want. With the expression written this way, FORTRAN will do the division (TSCOR/TSTIME) before it does the multiplication. Consequently, the operand that would be multiplied by FGNDEX will be the value TSCOR/TSTIME and not TSTIME by itself. In other words, the expression, as written, carries the intent of a different formula, namely,

$$\text{EARTNG} = \text{CHNUMB} + \frac{\text{TSCOR}}{\text{TSTIME}} \text{FNGDEX}$$

We can force FORTRAN to do the computations properly (for us) by using parentheses in the same way we would in a conventional formula. By rewriting the statement so that it says

```
EARTNG = CHNUMB + TSCOR/ (TSTIME*FGNDEX)
```

we have "isolated" TSTIME*FGNDEX as a separate subexpression. In a sense, we are telling FORTRAN to do the computations inside the parentheses first, so that there is a single result there. That number can then be used as the operand by which TSCOR will be divided. Having made that simple adjustment, we can now write the entire program (Figure 5.2).

There is only one rule regarding the use of parentheses in expressions: Parentheses come in pairs. We can use as many parentheses as we need to, as long as there is a right one to match every left one. When such a match is provided, the expression is said to contain *balanced* parentheses. Even when parentheses are not needed, their use will not upset anything, as long as they are balanced. For example, the expression

A + B + C

is perfectly fine without parentheses. There is no doubt as to what the computations will be, and parentheses obviously are not needed. However, we could put in unnecessary

```
*****************************************************************
*                          EXAMPLE 5.1                          *
*****************************************************************
*  THIS PROGRAM ILLUSTRATES THE USE OR PARENTHESES TO MAKE CLEAR *
*  THE EXACT SEQUENCE OF ARITHMETIC OPERATIONS NEEDED BY A GIVEN *
*  PROBLEM. AT THE SAME TIME, IT ILLUSTRATES THE USE OF AN ECHO  *
*  TO PROVIDE AN IMMEDIATE RECORD OF THE INPUT IN EXACTLY THE    *
*  FORM IN WHICH IT WAS RECEIVED.                               *
*****************************************************************

      PROGRAM          EX501
      IMPLICIT         NONE
      REAL*4           TSCOR,TSTIME,FGNDEX,EARTNG
      INTEGER*2        TSTNUM,CHNUMB
      CHARACTER*25     EMPNAM

*****************************************************************
*  INITIALIZE BY READING THE FIRST INPUT SET.                   *
*****************************************************************

      PRINT *, 'ENTER THE FIRST SET OF INPUT VALUES.'
      READ *,  EMPNAM,TSTNUM,TSCOR,TSTIME,CHNUMB,FGNDEX

*****************************************************************
*  HERE IS THE WHILE CONSTRUCTION:                              *
*  NOTE THE SPECIAL FORM OF THE READ STATEMENT INSIDE THE LOOP:  *
*  THE 'END=99' SPECIFIES THAT IF THE PROGRAM RUNS OUT OF DATA   *
*  WHILE TRYING TO PERFORM THIS INPUT OPERATION, IT IS TO PROCEED*
*  IMMEDIATELY TO THE STATEMENT WITH THE ATTACHED LABEL 99.      *
*****************************************************************
      DO WHILE (TSTNUM .NE.0)
         PRINT *, EMPNUM,TSTNUM,TSTIME,CHNUMB,FGNDEX
         EARTNG = CHNUMB + TSCOR/(TSTIME*FGNDEX)
         PRINT *, EARTNG
         PRINT *, 'ENTER THE NEXT SET OF INPUT VALUES.'
         READ (*,END=99)  EMPNAM,TSTNUM,TSCOR,TSTIME,CHNUMB,FGNDEX
      END DO

   99 PRINT *, 'RUN COMPLETED.'
      STOP
      END
```

**FIGURE 5.2** Program for Example 5.1.

parentheses, so that the expression reads

$$(A + B + C)$$
$$\text{or } (A + B) + C$$
$$\text{or } A + (B + C)$$
$$\text{or even } ((A + B) + (C))$$

without affecting the computations. Note, however, that –X**2 is quite different from (–X) **2. Table 5.2 shows some additional examples illustrating the use of parentheses.

Whenever there is any question with regard to how FORTRAN will interpret a particular expression (or part of an expression), any doubts can be removed by using

**Table 5.2** Use of Parentheses in FORTRAN Arithmetic Expression
Variables are A, B, C, D, E, F, G

| *Traditional Form* | *FORTRAN Form* |
|---|---|
| $\frac{A+B}{C}$ | (A+B) /C |
| $\frac{A+B}{C-D}$ | (A+B) / (C–D) |
| $A+\frac{B}{C-D}$ | A+B/ (C–D) |
| $A-\frac{B}{(C-D)^{2.5}}$ | A–B/ (C–D) **2. 5 |
| $(A-B)^2(C+2D)$ | (A–B) **2* (C+2*D) |
| $\frac{(A+B)(A-C)(D+E)^{0.8}}{G-F}$ | (A+B) * (A–C) * (D+E) **. 8/ (G–F) |
| $\frac{B(A-C^2+DE)^3}{(A+BC)^{-2}}$ | B* (A–C**2+D*E) **3/ (A+B*C) ** (–2) |

parentheses. Since there is no penalty in using parentheses that are not needed, it is a good practice to include them so that the expression can easily be read and interpreted by human.

### 5.1.2 Representation of Computed Results

Regardless of the length or complexity of an assignment statement, it directs FORTRAN to perform a sequence of two basic activities:

1. Evaluation of the expression on the right of the = sign to produce a single result.
2. Storage of that result in the destination specified by the variable on the left side of the = sign.

**5.1.2.1 Real Expressions** Once the computed result is available, it would seem to be a simple matter to place that value wherever it is supposed to go. Under many conditions, this is as simple as it sounds. However, we must note the fact that the destination is a *variable name.* As such, it has been declared earlier in the program, so that the name is associated with a certain type of data. For instance, if we had part of a program that said

```
REAL*4      COST, TAX
INTEGER*2   PRICE
*     *     *     *     *
*     *     *     *     *
TAX = 0. 045 * COST
```

there would be no question as to what happens: 0.045, a real constant, is multiplied by another real value, i.e., the one stored in the variable COST. The result, which is also a real number, is stored in TAX, a place set up to hold a real value.

Now, suppose we add another statement to this sequence: After the calculation of TAX, let us write

```
PRICE = COST + TAX
```

Again, the computation is clear enough: TAX and COST, both of them real, are

added together to produce a sum that is real. Now, the computer is ready to place this result in PRICE. But PRICE, having been declared INTEGER, cannot accept a real value. Consequently, FORTRAN introduces another step that converts the sum of COST and TAX to an integer and stores the converted value in PRICE. Since an integer has no fractional part, the conversion process must deal with it somehow. What it does is simple: It just throws the fraction away. (This is called *truncation.*) For example, if COST is 150.80 and TAX is 6.90, the sum (157.70) will be processed by ignoring the .70 and converting the whole number (157.) to integer form (157). Then, that value is stored in PRICE. (Sad rumor has it that *somewhere* not far from Carson City there is an enormous pile of discarded fractions drying and withering under the relentless Nevada sun. And this mountain is growing daily faster than ever.)

We can generalize from this example to a rule that governs the behavior of the assignment process.

> After a number is computed by evaluating an arithmetic expression, FORTRAN will convert that value (if necessary) so that its type matches that declared for its destination.

**5.1.2.2 Integer Expressions** Let us apply this rule to another situation, the reverse of the previous one: We shall specify some computations involving integers, with the result to be stored in a real variable.

```
REAL*4      TOTAL
INTEGER*2   SBTTL1, SBTTL2
SBTTL1 = 83
SBTTL2 = 19
TOTAL = SBTTL1 + 3*SBTTL2
```

Three times the value in SBTTL2 (57) is added to SBTTL1 (83) producing an integer result of 140 which now needs to be stored in the real variable TOTAL. Consequently, FORTRAN will convert it to the real value 140., and *that* value will be assigned to TOTAL. Table 5.3 gives some additional examples in which this rule is applied.

In looking at the illustrations in this table, note in particular the behavior of example (d): The expression calls for the division of two integer values, i.e., 23 divided by 8. The result, then, will be converted to real since its final destination is the real variable RATIO. Even though RATIO is perfectly capable of accommodating a number with a fraction, the value that ends up there (a 2.) has no fraction. Unfair, you say? But true nonetheless.

**Table 5.3** Conversion Rules in FORTRAN Expression

```
A = 12, B = 6., C = 3., D = 4., E = 2., F = 0., RATIO = 0.
I1 = 10, J = 2, K = 0
D1 = 4.D0, D2 = 24.D0, D3 = 6.0D0, D4 = 0.0D0
```

| *Assignment Statement* | *Result* |
|---|---|
| (a) K = J*I1 | A value of 20 is stored in K. |
| (b) K = D*E/B | A value of 1 is stored in k. |
| (c) K = D1 + D2 | A value of 28 is stored in K. |
| (d) RATIO = 23/(11 − J) | A value of 2, is stored in RATIO. |
| (e) F = C/A | A value of 0.25 is stored in F. |
| (f) F = D1 − D2 | A value of 20 is stored in F. |
| (g) D4 = D1 − D2 | A value of −2.0D + 1 is stored in D4. |
| (h) D4 = 23/(I1 − J) | A value of 2.0D0 is stored in D4. |
| (i) D4 = C/D | A value of 7.5D − 1 is stored in D4. |

Inside the computer, integers and real arithmetic are two separate and different processes, each involving its own electronic logic. Thus, when an arithmetic operation (like addition) is performed on two integers, the result automatically is an integer. Real operations produce real results. Accordingly, the type of arithmetic can have a considerable effect on the results when it comes to division. Since division of one integer by another can produce results with fractions, it is quite "normal" for such divisions to produce results that are incorrect. For example, division of 6 by 8 produces 0 while division of 6. by 8. produces the expected value of 0.750.

Sometimes there are situations where this behavior is useful and programmers will specify integer division intentionally. However, this can be used to advantage only after the programmer is familiar enough with computer arithmetic to know exactly what is going on. In general, it is a good idea to do almost all computing with real values, restricting the use of integer arithmetic to simple counting and adding/subtracting.

### 5.1.3 Rules for Conversion in Assignment Operations

When FORTRAN performs conversions between a computed result and its representation in a specified destination, there is no guarantee that the correct result always will be produced. Consequently, the programmer needs to know how FORTRAN will behave in a particular set of circumstances so that he or she can know when intervention is necessary. A detailed summary of these rules for HP FORTRAN 77 is given in Table 5.4.

Now that we are aware of this automatic conversion process and how it can affect the final result, we are ready to look a little more closely at the way FORTRAN handles the actual computations when it evaluates an expression. This is treated in the next section.

**Table 5.4** HP FORTRAN 77's Rules for Conversion in Assignment Operations
`Integer*2` = Integer for HP1000, Short Integer for HP9000;
`Integer*4` = Double Integer for HP1000, Integer for HP9000;
`Real*4` = Real; `Real*8` = Double Precision

| *Type of Destination* | *Type of Computed Value* | *Conversion Rule* |
|---|---|---|
| `Integer*2` | `Integer*2` | No conversion |
| `Integer*` | `Integer*4` | Truncate to `integer*2`; assigned value probably is incorrect. |
| `Integer*2` | `Real*4` | Truncate to `integer*2` |
| `Integer*2` | `Real*8` | Truncate, convert to `integer*2` |
| `Integer*4` | `Integer*4` | No conversion |
| `Integer*4` | `Integer*2` | Expand to `Integer*4` |
| `Integer*4` | `Real*4` | Truncate and convert to `integer*4` |
| `Integer*4` | `Real*8` | Truncate and convert to `integer*4` |
| `Real*4` | `Real*4` | No conversion |
| `Real*4` | `Integer*2` | Convert to `real*4` |
| `Real*4` | `Integer*4` | Convert to `real*4`; converted value probably is incorrect since the precision available in an `integer*4` value exceeds that available in a `real*4` |
| `Real*4` | `Real*8` | Round to `real*4` |
| `Real*8` | `Real*8` | No conversion |
| `Real*8` | `Integer*2` | Convert to `Real*8` |
| `Real*8` | `Integer*4` | Convert to `Real*8` |
| `Real*8` | `Real*4` | Expand to `Real*8` |

## 5.2 HOW COMPUTATIONS ARE PERFORMED

Back in the first chapter it was pointed out that most computers are built to perform simple arithmetic operations, one at a time. Since we can write an arithmetic expression in FORTRAN that is just about as long and complicated as we care to make it, there must be some definite rules built into FORTRAN that regulate the way it handles such expressions. The rules are never violated. If they were, it would be impossible to know what will happen from one time to the next. Consequently, by learning these rules, which are standardized for all FORTRAN 77 systems, we can control the computations and avoid surprises.

### 5.2.1 Arithmetic Involving Different Types of Data

The previous section brought to light the need for FORTRAN to do some automatic data conversion in order to send the right type of numerical value for its intended destination. As part of that discussion, we took note of the fact that, in a computer, integer and real arithmetic are two different processes. Each of these requires all of the data to be in the right form (all integers for integer arithmetic, all real values for real arithmetic). Consequently, when we write a FORTRAN expression, for example, in which we specify a real value to be added to an integer, it is impossible for the computer to do this. Again, FORTRAN must step in and make everything nice. Making everything nice consists of converting one of the values so that its type matches that of the other. But which one? That is where the rules come in.

**5.2.1.1 Conversion Rules for +, −, *, /** Table 5.5 shows the conversions that automatically take place when +, −, *, or / is specified for two different types of operands. We can see how these rules work by applying them to a few simple situations. To set things up, we shall make the following declarations:

```
REAL*4      R1, R2
INTEGER*2   I1, I2
INTEGER*4   K1, K2
REAL*8      D2
```

**Table 5.5** Conversion of Operands for +, −, *, /
`Integer*2` = Integer for HP1000, Short Integer for HP9000;
`Integer*4` = Double Integer for HP1000, Integer for HP9000;
`Real*4` = Real; `Real*8` = Double Precision

| *If the two operands are* | *FORTRAN will take the following action:* |
|---|---|
| `Real*4` and `Real*4` (†) | None |
| `Integer*2` and `Integer*2` | None |
| `Integer*4` and `Integer*4` | None |
| `Real*8` and `Real*8` | None |
| `Real*4` and `Integer*2` | The integer operand is converted to real before the arithmetic is done. |
| `Real*4` and `Integer*4` | The `integer*4` operand is converted to `real*4` before the arithmetic is done. |
| `Real*4` and `Real*8` | The `real*4` operand is converted to `real*8` before the arithmetic is done. |
| `Integer*2` and `Real*8` | The integer operand is converted to `real*8` prior to arithmetic. |
| `Integer*4` and `Real*8` | The `integer*4` operand is converted to `real*8` prior to arithmetic. |

† `Real` and `Real`, for example, could mean `Real+Real`, `Real−Real`, `Real*Real`, or `Real/Real`.

Now, assuming that values are available for all the variables, we can examine some of these rules.

1. R1 = K2+I1
   I1's value is converted to double integer, added to K2, and the result (converted to real) is stored in R1.
2. R1 = R2+I1
   I1's value is converted to real, added to R2, and the result (real) is stored in R1.
3. I2 = R2+I1
   I1's value is converted to real and added to R2; the result (real) is converted to integer (any fraction is lost) and the integer value is stored in I2.
4. I1 = D2+K1
   K1's value is converted to double precision and added to D2. The result (converted to integer) is stored in I1. LE; R1 = R2+D2 R2's value is converted to double precision and added to D2. The double precision result is converted to single precision (real) and stored in R1.
5. I1 = D2+I2
   I2 is converted to double precision and added to D2. The double precision result is converted to integer and stored in I1.

**5.2.1.2 Conversion Rules for Exponentiation** Exponentiation is a special operation because it is much more complicated than the other four binary arithmetic operations. In almost all computers, addition, subtraction, multiplication, and division are actual instructions built into the hardware. This is not the case with exponentiation. Even though we can specify it as a single activity, FORTRAN has to build quite a complicated sequence of operations to produce the desired result. In order to make sure that this process works properly every time, exponentiation has its own set of conversion rules.

**Table 5.6** Conversion of Operands for **

| *If the Expression is* | *FORTRAN takes the following action:* |
|---|---|
| `Integer*2**Integer*2` | None |
| `Integer*4**Integer*4` | None |
| `Integer*4**Integer*2` | None |
| `Integer*2**Integer*4` | The `integer*2` is converted to `integer*4` and then raised to the `integer*4` power. |
| `Real*4**Real*4` | None |
| `Real*4**Integer*2` | None |
| `Real*4**Integer*4` | None |
| `Integer*2**Real*4` | The `integer*2` is converted to `real*4` and then raised to the real power |
| `Integer*4**Real*4` | The `integer*4` is converted to `real*4` and then raised to the `real*4` power |
| `Real*8**Real*8` | None |
| `Real*8**Integer*2` | None |
| `Real*8**Integer*4` | None |
| `Real*8**Real*4` | The real*4 value (the exponent) converted to `real*8` and then the exponentiation is performed. |
| `Integer*2**Real*8` | The `integer*2` is converted to `real*8` and then raised to a `real*8` power. |
| `Integer*4**Real*8` | The `integer*4` is converted to `real*8` and then raised to a `real*8` power. |
| `Real*4**Real*8` | The `real*4` operand is converted to `real*8` and then raised to a `real*8` power. |

These are summarized in Table 5.6. To illustrate,

1. 2 **4 requires no conversion
2. 2 ** 3.5 is performed as 2. ** 3.5
3. 2.6 ** 3 requires no conversion
4. 3 ** 2.6D−1 is performed as 3.0D0 ** 2.6D−1

**5.2.2 Priorities in Arithmetic Expressions** Now we shall examine what happens when FORTRAN is faced with an expression containing several operations. Regardless of whether or not any conversion is needed before a particular operation is performed, there are carefully defined rules that govern FORTRAN's behavior. As is the case with conversions, there are separate rules for exponentiation.

**5.2.2.1 Priorities for +, −, *, /** If an arithmetic expression contains no exponentiation, FORTRAN performs the operations in the following order:

1. Multiplication and Division
2. Addition, Subtraction, Identity, Negation

FORTRAN will go through the expression from *left to right*, doing any multiplications and divisions it finds. Then, it will go back and take care of the remaining operations. A few examples will illustrate exactly how this works. To avoid unnecessary complications, we shall assume that all the variables and constants are real.

1. Y=A+B/C−D
   1. perform B/C (call the result T1 just to call it something)
   2. perform A+T1 (call the result T2)
   3. perform T2−D (call the result T3)
   4. store T3 in Y
2. Y=A+B/C*D/E
   1. perform B/C (call the result T1)
   2. perform T1*D (call the result T2)
   3. perform T2/E (call the result T3)
   4. perform A+T3 (call the result T4)
   5. store T4 in Y
3. Y=−A+B*D*C/E*F−2.*G
   1. perform B*D (call the result T1)
   2. perform T1*C (call the result T2)
   3. perform T2/E (call the result T3)
   4. perform T3*F (call the result T4)
   5. perform 2.*G (call the result T5)
   6. perform −A (call the result T6)
   7. perform T6+T4 (call the result T7)
   8. perform T7−T5 (call the result T8)
   9. store T8 in Y

**5.2.2.2 Expressions with Exponentiation** Exponentiation is done ahead of the other arithmetic operations. We shall illustrate:

1. Y=A+B ** G/C*D/E **G
   1. raise B to the Gth power (call the result T1)

2. raise E to the Gth power (call the result T2)
3. perform T1*D (call the result T3)
4. perform T3*D (call the result T4)
5. perform T4/T2 (call the result T5)
6. perform A+T5 (call the result T6)
7. store T6 in Y

2. When there are two or more exponentiations in succession, FORTRAN handles them from *right to left*. Let us see what this means:

```
Y = A ** B ** C
```

1. raise B to the Cth power (call the result T1)
2. raise A to the T1th power (call the result T2)
3. store T2 in Y

If in the above example A is 4., B is 2., and C is 3., then we can restate the sequence in terms of actual values:

4. T1 = 2. ** 3 = 8.
5. T2 = 4. ** T1 = 4. ** 8. = 65536. 0 (Y will have 65536. 0 in it)

Note that the result would be different if the rule went the other way:

1. T1 = 4. ** 2. = 16.
2. T2 = 16. ** 3. = 4096. (Y would have 4096. 0 in it)

Thus, because of FORTRAN's right-to-left rule for exponentiations, the expression

```
A ** B ** C
```

gives the same result as the expression

```
A ** (B ** C)
```

As indicated in Section 5.1.1.4, the inclusion of parentheses is a good way to settle any doubts.

**5.2.2.3 Changing Priorities with Parentheses** Now that we understand FORTRAN's arithmetic priorities, we should take a second, brief look at parentheses in this new light: when we use parentheses in an expression, what we really are doing is forcing FORTRAN to upset its normal priority rules and handle the material inside the parentheses first, even though there may be other, higher priority operations that ought to come first. A simple illustration will demonstrate: Let us examine the assignment statement

```
Y = A * (B+C/E*F) / (D+E)
```

and the same variables and operations without parentheses:

```
Y = A * B+C/E*F/D+E
```

Applying the rules to the second statement, we set the following sequence:

1. perform A*B (call the result T1)
2. perform C/E (call the result T2)
3. perform T2*F (call the result T3)
4. perform T3/D (call the result T4)
5. perform T1+T4 (call the result T5)

6. perform T5+E (call the result T6)
7. store T6 in Y

Now, once we include the parentheses, things are different:

1. perform C/E (call the result S1)
2. perform S1*F (call the result S2)
3. perform B+S2 (call the result S3; note that the regular priorities still hold inside each pair of parentheses)
4. perform D+E (call the result S4)
5. perform A*S3 (call the result S5)
6. perform S5/S4 (call the result S6)
7. store S6 in Y

By comparing the sequence that FORTRAN follows for each of these two statements, we can see that when we do not include parentheses, it would be the same as writing

```
Y= (A*B) + (C/E) * (F/D) -E
```

As was pointed out before, the use of parentheses is convenient enough so that there is every reason to put them in whenever there is the slightest doubt as to how FORTRAN will handle an expression.

## 5.3 COMPUTATIONAL EXAMPLES

We shall look at several programs that illustrate the construction and use of assignment statements for handling computations. In the process of developing these programs, we shall introduce some simple programming techniques which help make it easier to set up more extensive calculations. It will be helpful for future work to take note of the fact that the actual computations often occupy a small part of the program's attention. In many instances we shall be more concerned with setting the stage for the computations and with convenient delivery of the results.

**Example 5.2** An equation that finds frequent use in scientific and engineering work is the polynomial, an example of which is shown below:

$$Y = C_0 + C_1X + C_2X^2 + C_3X^3 + C_4X^4$$

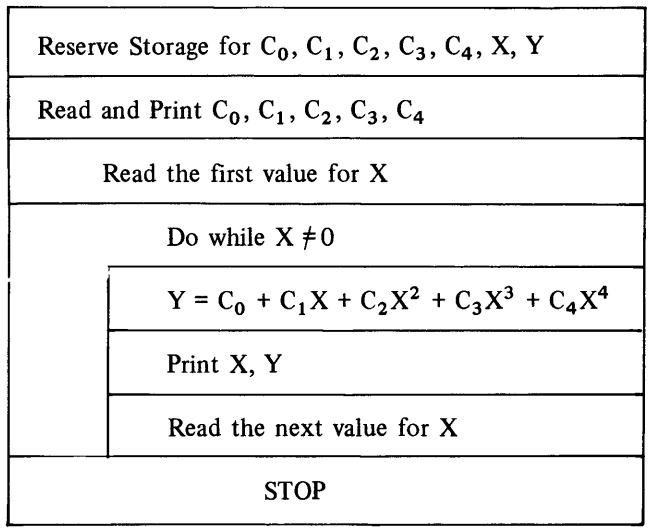
This example shows a fourth degree polynomial, so called because the highest power to which X is raised (in this particular case) is 4. In general, then, an nth degree polynomial is expressed as

$$Y = C_0 + C_1X + C_2X^2 + C_3X^3 + \cdots + C_nX^n$$

In many situations, the coefficients $C_0, C_1, C_2, \ldots, C_n$ are available, and values of Y are required for corresponding values of X. The program in this example is designed to do just that for a fourth degree polynomial: After reading values for the coefficients $C_0, C_1, C_2, C_3$, and $C_4$, it reads a value of X and computes the corresponding Y. This process is repeated for each of a succession of X values. A value of zero is used to indicate the end of the data.

Since there is no information about how big or how small the data values will be, we shall declare everything to be double precision. In this way, we make available the system's widest range of expression. The computation of Y is handled by a single assignment statement which, though long, is a direct carryover from the mathematical equation. Figure 5.3 gives a flowchart and pseudocode description for the program. The FORTRAN statements, along with a sample run, are shown in Figures 5.4 and 5.5. Note that the coefficients, as well as the X values, are submitted in regular (i.e., traditional) form (Figure 5.5). Yet, when the program displays them, they appear as double precision numbers. This emphasizes the fact that values for numerical variables are stored the way they are declared. If they are not in the declared form when they are read in, FORTRAN will do the necessary conversion automatically.

**Example 5.3** We shall solve the same problem as in the previous example with a slightly different method of computation. Although Example 5.2's assignment statement is straightforward enough, it




**(a)**

"Define variables $C_0$, $C_1$, $C_2$, $C_3$, $C_4$, X, and Y."
"Read and print $C_0$, $C_1$, $C_2$, $C_3$, and $C_4$."
"Read the first value of X."
**while** X is not equal to zero:

    "Print X and Y."
    "Read the next value of X."

**endwhile**
"Stop."

**(b)**

**FIGURE 5.3** **(a)** N-S Diagram for Example 5.2. **(b)** Pseudocode for Example 5.2.

```
************************************************************************
*                              EXAMPLE 5.2                             *
************************************************************************
*  C0, C1, C2, C3, AND C4 ARE THE POLYNOMIAL'S COEFFICIENTS;           *
*  X IS THE VALUE FOR WHICH THE POLYNOMIAL IS TO BE EVALUATED;         *
*  Y IS THE POLYNOMIAL'S VALUE FOR A GIVEN X;                          *
*  AN X OF ZERO ENDS THE RUN.                                          *
************************************************************************

      PROGRAM              EX502
      IMPLICIT             NONE
      REAL*8               C0,C1,C2,C3,C4,X,Y

************************************************************************
*  WE SHALL START THE PROCESS BY READING THE COFFICIENTS AND THE       *
*  VALUE OF X.                                                         *
************************************************************************

      PRINT *, 'ENTER THE COFFICIENT VALUES.'
      READ *,  C0, C1, C2, C3, C4
      PRINT *, 'NOW ENTER THE FIRST VALUE FOR X.'
      READ *,  X

      DO WHILE (X .NE. 0.0D0)
         Y = C0 + C1*X + C2*X**2 + C3*X**3 + C4*X**4
         PRINT *, 'X = :',X,'Y = :',Y
         PRINT *, 'ENTER THE NEXT VALUE FOR X.'
         READ (*,END=99) X
      END DO

   99 PRINT *, 'RUN COMPLETED.'
      STOP
      END
```

**FIGURE 5.4** FORTRAN Statements for Example 5.2.

```
 0.1000000000000000D 01   0.2000000000000000D 01   0.3000000000000000D 01   0.2000000000000000D
 0.1000000000000000D 01
 0.1000000000000000D 01   0.9000000000000000D 01
 0.4000000000000000D 01   0.4410000000000000D 03
-0.3000000000000000D 01   0.4900000000000002D 02
 0.2000000000000000D 01   0.4900000000000000D 02
```

**FIGURE 5.5** Sample Run for Example 5.2.

uses exponentiation. As pointed out earlier, this sets in motion a rather involved set of operations. In this instance we can avoid them because all of the powers of X are integers. If we wanted to raise X to a power with a fraction in it, like 3.72, we would have to use exponentiation; but since the polynomial makes no such demands, we can get the same result without forcing the system to do nearly as much computation. To see how this is done, we shall restate the original equation:

$$Y = C_0 + C_1X + C_2X^2 + C_3X^3 + C_4X^4$$

Note that the last four terms have X as a common factor. By taking that X out, we can rewrite the equation as

$$Y = C_0 + X(C_1 + C_2X + C_3X^2 + C_4X^3)$$

```
************************************************************************
*                            EXAMPLE 5.3                               *
************************************************************************
*  C0, C1, C2, C3, AND C4 ARE THE POLYNOMIAL'S COEFFICIENTS;           *
*  X IS THE VALUE AT WHICH THE POLYNOMIAL IS TO BE EVALUATED;          *
*  Y IS THE POLYNOMIAL'S VALUE AT A GIVEN X. HORNER'S METHOD IS        *
*  USED FOR EVALUATION. AN X OF ZERO ENDS THE RUN.                     *
************************************************************************

      PROGRAM             EX503
      IMPLICIT            NONE
      REAL*8              C0, C1, C2, C3, C4, X, Y

************************************************************************
*  WE SHALL START THE PROCESS BY READING THE COEFFICIENTS AND          *
*  THE FIRST VALUE FOR X.                                              *
************************************************************************

      PRINT *, 'ENTER THE VALUES FOR THE COEFFICIENTS'
      READ *,  C0, C1, C2, C3, C4
      PRINT *, 'ENTER THE FIRST VALUE FOR X.'
      READ *,  X

      DO WHILE (X .NE. 0.0D0)
         Y = C0 + X*(C1 + X*(C2 + X*(C3 + X*(C4))))
         PRINT *, X, Y
         PRINT *, 'ENTER THE NEXT VALUE FOR X'
         READ (*,END=99) X
      END DO

   99 PRINT *, 'RUN COMPLETED.'
      STOP
      END
```

**FIGURE 5.6** FORTRAN Statements for Example 5.3.

Now, the last three terms inside the parentheses have X as a common factor, and we can do the same thing to those three terms:

$$Y = C_0 + X(C_1 + X(C_2 + C_3X + C_4X^2))$$

Since we have something good going here, there is no reason to quit until we must. Clearly, we can take X out as a common factor one more time:

$$Y = C_0 + X(C_1 + X(C_2 + X(C_3 + C_4X)))$$

This way of expressing a polynomial is called *Horner's method*, an approach especially well suited for automatic computation. Now, since there is no change in the overall structure of the program, Figure 5.3 still describes its flow and Figure 5.6 shows the revised version, with the new assignment statement:

```
Y=C0+X*(C1+X*(C2+X*(C3+X*C4)))
```

This is just one of an endless number of instances in which it pays to do some thinking about a computational method rather than following a "brute force" approach. Thus, Example 5.3 takes advantage of the fact that the full strength (and complexity) of exponentiation is not needed to attend to the relatively simple requirements of the polynomial.

**Example 5.4** This time we shall look at a problem involving more complex computations. When engineers analyze the transfer of heat in various fluids, an important measure of such transfer is something called the *convective heat transfer coefficient*. This quantity, called H, depends on a variety of physical properties and operating conditions in the system being analyzed. For certain situations, H can be computed as follows:

$$H = 0.023 \left(\frac{K}{D}\right)\left(\frac{DVR}{U}\right)^{0.8}\left(\frac{UC}{K}\right)^{0.4}$$

where the variables have the following meanings:

H = the convective heat transfer coefficient
D = the diameter of the pipe carrying the fluid
V = the velocity of the fluid traveling through the pipe
R = the density of the fluid traveling through the pipe
U = the viscosity of the fluid traveling through the pipe
C = the specific heat of the fluid
K = the conductivity of the fluid

The value 0.023 is sometimes known as the D-B constant. For many situations, the size of the pipe and type of fluid are selected so that the fluid's properties (R, U, C, and K) are defined. Thus, what is required is a set of H values at various velocities.

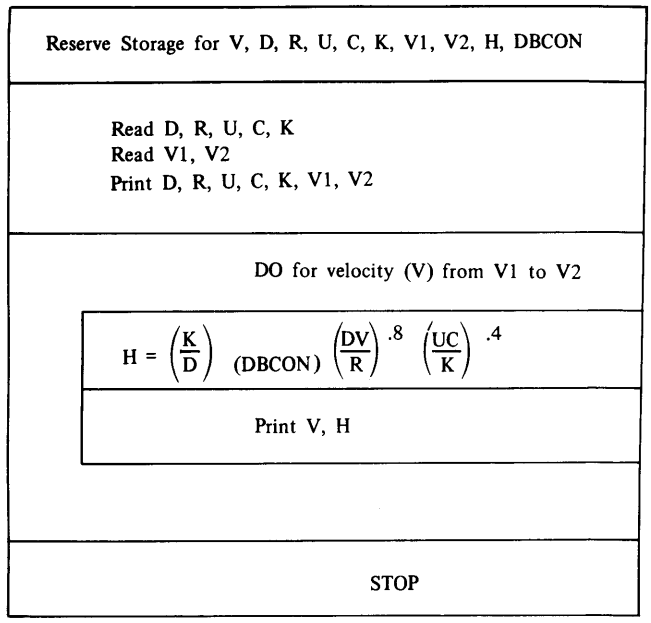
Accordingly, we shall write a program whose input consists of two basic types of data:

1. A description of the system (D, R, U, C, K)
2. The velocities of interest, expressed as a minimum (`V1`) and a maximum (`V2`).

The two input velocities will be integer values, expressed as real numbers. The values of H will be computed over the indicated range at intervals of one velocity unit. That is, the first value of H will be computed for `V1`, the next one for `V1+1.0`, the next for `V1+2.0`, and so on, all the way up to `V2-2.0`, `V2-1.0`, and finally `V2`.

The flow diagram and pseudocode, shown in Figure 5.7, are carried over directly in the FORTRAN statements of Figure 5.8. This is a clear demonstration of FORTRAN's computational powers, since the involved formula still can be represented by a single assignment statement, and the overall program remains quite simple.

**When a set of numerical computations becomes extensive, no good purpose is served by specifying all of the work in a single assignment statement. Even though it is possible to do so, it often is a better practice to break up the calculations into smaller pieces, each one of which is handled by a separate statement. The result of each partial group of computa-**




**(a)**

"Define variables V, D, R, U, C, K, V1, V2, H, DBCON."
"Read D,R,U,C,K."
"Read V1,V2."
"Print D, R, U, C, K, V1, V2."
**do** for all values of V from V1, V1 + 1, etc. to V2."

"Compute $H = 0.023 \left(\frac{K}{D}\right) \left(\frac{DVR}{U}\right)^{0.8} \left(\frac{UC}{K}\right)^{0.4}$."

"Print V and H."
**enddo**
"Stop."

**(b)**

**FIGURE 5.7** **(a)** Structured Flowchart for Example 5.4. **(b)** Pseudocode for Example 5.4.

```
************************************************************************
*                          EXAMPLE 5.4                                 *
************************************************************************
*  THIS EXAMPLE ILLUSTRATES THE USE OF A DO LOOP FOR THE SYSTEMA-*
*  TIC PRODUCTION OF A TABLE OF NUMERICAL VALUES. IN THIS CASE,  *
*  WE SHALL DEVELOP A TABLE OF H VALUES FOR A GIVEN SET OF VALUES*
*  FOR DIAMETER (D), DENSITY (R), VISCOSITY (U), CONDUCTIVITY (K)*
*  AND SPECIFIC HEAT (C). THE VALUES WILL BE DEVELOPED FOR A     *
*  RANGE OF VELOCITIES V1 TO V2 IN INCREMENTS OF 1.              *
************************************************************************

      PROGRAM       EX504
      IMPLICIT      NONE
      REAL*4        V,D,R,U,C,K,V1,V2,H
      PARAMETER     (DBCON = 0.023)

      PRINT *, 'ENTER THE SET OF VALUES FOR D,R,U,C, AND K.'
      READ *,  D, R, U, C, K
      PRINT *, 'NOW ENTER THE VALUES FOR V1 AND V2.'
      READ *,  V1, V2
      PRINT *, D, R, U, C, K, V1, V2

************************************************************************
*  HERE IS OUR LOOP. THE 1.0 IN THE DO STATEMENT INDICATES THAT  *
*  V WILL BE INCREASED BY 1.0 BEFORE THE BEGINNING OF EACH CYCLE *
************************************************************************

      DO  V = V1, V2, 1.0
         H = DBCON * (K/D) * (D*V*R/U)**0.8 * (U*C/K)**0.4
         PRINT *, 'V = ',V,'H = ',H
      END DO
*
      STOP
      END
```

**FIGURE 5.8** FORTRAN Statements for Example 5.4.

```
************************************************************************
*                          EXAMPLE 5.4 (REVISED)                       *
************************************************************************
*  THE ONLY DIFFERENCE BETWEEN THIS EXAMPLE AND THE PREVIOUS ONE       *
*  IS THAT WE INTRODUCE TWO INTERMEDIATE VARIABLES (RE AND PR)         *
*  FOR TEMPORARY STORAGE OF PARTIAL COMPUTATIONS. THIS CAN MAKE        *
*  THE PRESENTATION OF THE COMPUTATIONS EASIER TO FOLLOW SINCE         *
*  EACH ASSIGNMENT STATEMENT IS SIMPLER.                               *
************************************************************************

      PROGRAM        EX504A
      IMPLICIT       NONE
      REAL*4         V,D,R,U,C,K,V1,V2,H,RE,PR
      PARAMETER      (DBCON = 0.023)

      PRINT *, 'ENTER THE VALUES FOR D, R, U, C, K, VI, AND V2'
      READ *, D, R, U, C, K, V1, V2
      PRINT *, D, R, U, C, K, V1, V2

      DO V = V1, V2, 1.0
         RE = (D*V*R/U)**0.8
         PR = (U*C/K***0.4
         H = DBCON *(K/D) * RE * PR
         PRINT *, 'V = ',V,'H = ',H
      END DO

      STOP
      END
```

**FIGURE 5.9** FORTRAN Statements for Revised Example 5.4.

tions is stored in a temporary variable declared for that purpose. Then, the final step uses these temporary variables in a simple statement that produces the desired result. This type of separation makes it much easier for someone reading the program to follow the exact sequence of the calculations so that they can be checked for correctness. Just when this needs to be done, and how the breakup should occur, are matters of judgment. However, there should be no hesitation in doing this: There is no particular penalty in terms of efficiency, and there is much to be gained in keeping the program as clear as possible.

A second version of Example 5.4 (Figure 5.9) illustrates the idea. In this instance H is computed as a three-step process. First, temporary variables RE and PR are computed and stored, after which the final production of H involves simple multiplication. Of course, the advantages of doing this become more dramatic with more extensive computations.

## 5.4 SUMMARY

In order to guarantee consistency, FORTRAN performs its arithmetic in accordance with a set of rules:

1. There are four basic types of arithmetic: integer arithmetic, which involves only integer values and produces integer results; double integer arithmetic, which produces double integer results using double integer components; real arithmetic, involving only real values and producing real results; and double precision arithmetic, dealing only with double precision numbers. When a FORTRAN expression calls for arithmetic involving mixed data types, FORTRAN will convert automatically so that the values are of the same numerical type. This makes it possible to perform the arithmetic.

2. After FORTRAN finishes doing the arithmetic specified in an expression, it will convert the result, if necessary, so that it matches the numerical data type declared for the variable in which that result is to be stored.

3. When an expression specifies more than one arithmetic operation, FORTRAN repeatedly works through the expression from left to right. It does exponentiation first, then multiplications and divisions, and then additions and subtractions.

4. If an expression contains more than one exponentiation in succession, FORTRAN handles the exponentiation from right to left.

5. Parentheses will force a change in arithmetic priorities. FORTRAN will do the arithmetic specified inside parentheses separately, treating the parenthesized contents as a separate expression. When a result for that expression has been produced, FORTRAN moves on to the rest of the larger expression.

6. When FORTRAN divides an integer by another integer, the result is an integer value with any fractional part having been truncated. (For instance, 24/25 produces a result of zero.) The same holds true for double integer division.

**PROBLEMS**

1. For each of the following assignment statements:
   1. Rewrite the statement as a conventional equation.
   2. Specify the number of operators and operands and list them.
   3. Indicate the value in real variable OUTCOME as a result of the specified computations. Assume the following values in each computation:

```
XVAL=40.0     YVAL=4.0     ZVAL=6.0     WRT=2.5
```

   (a) `OUTCOME=XVAL+YVAL+ZVAL`
   (b) `OUTCOME=XVAL*YVAL*ZVAL+WRT*YVAL`
   (c) `OUTCOME=XVAL/YVAL+YVAL/XVAL+*WRT`
   (d) `OUTCOME=XVAL*YVAL/ZVAL+WRT*ZVAL/YVAL`
   (e) `OUTCOME=XVAL**2+YVAL**3-ZVAL/WRT`
   (f) `OUTCOME=XVAL**2*YVAL**ZVAL**2/WRT`
   (g) `OUTCOME=YVAL*ZVAL**2/XVAL-4*WRT`
   (h) `OUTCOME=XVAL/YVAL**2/ZVAL+WRT**4+YVAL**0.5`

2. Write FORTRAN assignment statements for each of the following computations:

   (a) $Z = XY + TW$

   (b) $Z = X^2Y^2 + \frac{T}{W}$

   (c) $Z = \frac{X^3}{Y^2} - \frac{T}{W^2} - 2.2W$

   (d) $Z = \frac{T^{1.4}W^{1.7}X^{0.8}}{Y^2}$

   (e) $Z = \frac{T^{.6}X^{.4}}{W^{.27}} - \frac{Y}{T^{.2}} + \frac{X^{.85}}{T}$

   (f) $Z = -\frac{TW^{1.8}}{X^{.6}} + \frac{TX^2Y}{-W} - 3.2X^{.7}$

   (g) $Z = \frac{X^{.9}Y^{.7}T^{.8}}{W^{.4}} - 2.2^{W}$

   (h) $Z = \frac{4}{3}\pi R^3$

3. Write FORTRAN assignment statements for each of the following computations:

   (a) $Z = \frac{X - Y}{X + Y}$

   (b) $Z = X^2 + \frac{Y}{T + W}$

   (c) $Z = X^Y + \frac{YW}{X + W^2}$

   (d) $Z = 4.42X^{Y-2} + \frac{WT^4}{XY}$

   (e) $Z = \frac{X}{Y + 6} - \frac{Y}{X + 6} + \frac{XY}{XW - T}$

   (f) $Z = \frac{X + 2.7Y^2 - 36W + 41.7T^4}{Y - 71.7X^2 + W}$

   (g) $Z = 3.88E3 - \frac{Y}{X + T}$

   (h) $Z = \frac{X - 2.1Y}{W - \frac{T}{X^2}}$

(i) $Z = \dfrac{X^{2.2} + 2.2Y}{\dfrac{W}{Y} + \dfrac{T}{X+7}}$

(j) $Z = \dfrac{\left(\dfrac{X}{Y} + \dfrac{W^2}{X}\right)}{\dfrac{Y^{.8}}{W+1} + \dfrac{X}{Y}} - \dfrac{Y^{1.7}}{-2}$

(k) $Z = X(2 + Y)$

(l) $Z = \dfrac{1}{X(2+Y)}$

(m) $Z = 2.8X - \left(\dfrac{Y}{W}\right)^2$

(n) $Z = \left(\dfrac{X}{Y}\right)^{1.8}\left(\dfrac{W}{T}\right)^{1.6}$

(o) $Z = X + \left(\dfrac{Y^2}{4-W}\right)^{.7} + 3.6(X+Y)^{1.2}$

(p) $Z = \dfrac{(X+Y)(Y-2.6W)}{(W+Y)^2}$

(q) $Z = \left(\dfrac{X}{Y}\right)^{2W-1}$

(r) $Z = \left(\dfrac{X+2.1Y}{W - \dfrac{T}{X^2}}\right)^{1.7}\left(\dfrac{2X(3W-7.4)}{T+W-2Y}\right)$

(s) $Z = \left(\dfrac{(X+3Y)^{\frac{1}{T}}(W-1.1XY)^{\frac{1}{T}-\frac{1}{X}}}{(1+Y^2)(3W(X^2+4T))}\right)^{\frac{3X^2}{2W}}$

4. For each of the statements given below:
   1. Rewrite the statement as a conventional algebraic equation.
   2. Using the following values:

      X=2.0 Y=1.5 T=0.6 W=0.4

      show the value of Z resulting from each of the specified computations. Write Z in floating point notation using six decimal places.

   (a) `Z=X*Y/T*W`
   (b) `Z=X*Y/2*T-W`
   (c) `Z=-2*X**2`
   (d) `Z=X*(Y/W)`
   (e) `Z=X*(Y/W)`
   (f) `Z=(X+2.8*Y)/(W-2.8*X)`
   (g) `Z=X+Y*(W+T*(S6*X))`
   (h) `Z=(Y*(X/W))**(T+1)`
   (i) `Z=(6.4+7.1*X-2.2E4*X**2+0.6E2*X**3)**2`
   (j) `Z=(2.8+Y+X**2+2.1*(Y-7.7)**1.1)**0.8/(X+0.8*W)`

5. Using the approach illustrated in Figure 5.9, rewrite each of the statements in the following problems as an equivalent sequence of several statements, each of which specifies part of the computations. As a rough guideline, assume that an expression should not include more than three operators. Use `T1`, `T2`, `T3`, etc. as names for intermediate variables that you may need. A possible breakdown of the statement in Problem 4(f) is shown as an additional illustration:

```
T1=X+2.8*Y
T2=W-2.8*X
 Z=T1/T2
```

   (a) Problem 4(g)
   (b) Problem 4(h)
   (c) Problem 4(i)
   (d) Problem 4(j)

6. Assuming that variables `X`, `Y`, `Z`, `W`, and `T` all are declared as real variables and have values assigned to them, indicate which of the following statements are illegal. Describe what is wrong with the illegal ones:

   (a) `Y=Y+X+W`
   (b) `6.3=2.8*X-17.4`
   (c) `Z=X+YW`
   (d) `Z-4=(X+Y)/3`
   (e) `Z=X+Y/(W+Z)`
   (f) `X=X+Y(Z-2W)`
   (g) `(Y)=(W+4)/(Y**(2-X))`
   (h) `T=(T+*X-Y)/(W+X)`
   (i) `Y=X*(2+3.8*W*(Z-7)**2`
   (j) `W=2*X**2+Y**2`
   (k) `X=(3+X)*(8-Y**2)`
   (l) `Z=((X+Y)*(Z-W))/4.4*T**2+X-)`

7. Suppose we have the following sequence of statements:

```
REAL*8              DVAL1,DVAL2,DVAL3
REAL*4              RVAL1,RVAL2,RVAL3
INTEGER*2           IVAL1,IVAL2,IVAL3
INTEGER*4           FVAL1,FVAL2,FVAL3
PARAMETER           (RPAR=6.0,DPAR=2.4D1,IPAR=-4)
DATA    DVAL1,DVAL2,DVAL3,RVAL1,IVAL1,IVAL2,FVAL1/2.0,3.0,1.0,4.0,10,8,5/
RVAL2=-1.0
IVAL3=-2
```

For each statement given below:
1. Specify the data type for the computed result.
2. Specify the data type for the stored result.
3. Show what the computed value is, using six and fourteen decimal places, respectively, for real and double precision results. Treat each statement independently of the others:

(a) `RVAL3=2.0*(RVAL1-RVAL2)**2`
(b) `RVAL2=RVAL2+((RVAL1-8.8)/(RVAL1+RPAR))/(8.0*RPAR)`
(c) `RVAL3=IPAR*(IVAL1+3*FVAL1)**2`
(d) `RVAL1=IPAR/(4+IPAR*(IVAL1-IVAL2))`
(e) `RVAL3=DVAL1*(CVAL2-3.0D-1*DPAR)`
(f) `RVAL1=(DVAL1+3.D0*DVAL2-DPAR)/(DVAL2/(2.0+DPAR))`
(g) `RVAL1=2*RVAL1 + (IVAL1-RVAL2)*FVAL1`
(h) `RVAL2=IPAR + (RVAL2+RVAL1) + 4.4*(IVAL3/(IVAL2+RPAR))`
(i) `RVAL1=DVAL1*RVAL2-DVAL2*RVAL1`
(j) `RVAL3=(RPAR-RVAL2*3.6)/(DPAR+1.6)**(RPAR/2.0)`
(k) `RVAL2=IVAL3*(DPAR-IVAL2)+(IVAL1/DVAL1)**2`
(l) `RVAL3=DPAR*(IVAL1+IVAL2)/(DVAL1*DVAL2-IPAR)`
(m) `RVAL1=IPAR*FVAL1*(DVAL1+4)*(RVAL2)/4.0`
(n) `RVAL1=IPAR*RVAL1+(IVAL1-IVAL2)*(DVAL1+5.4)/(DVAL1-RVAL2)`

8. Using the same statements as in the previous problem, apply the instructions given in that problem to each of the following statements:

(a) `IVAL3=2*(IVAL1+IVAL2)-IPAR`
(b) `IVAL3=IVAL3-(IVAL1+IVAL2)/(2*IPAR)**2`
(c) `IVAL1=RVAL1*(RPAR+RVAL2/RVAL2)`
(d) `IVAL2=((3.6+RVAL1)*(RVAL2-5.4))**(RPAR-4.0*RVAL1)`
(e) `IVAL1=DPAR+FVAL1-2.0D1*DVAL1`
(f) `IVAL3=2.4D2-(DVAL1+DVAL2)*(DPAR/5.0D-2)`
(g) `FVAL2=IVAL1*(RPAR-1)*(IVAL2+RVAL2)`
(h) `IVAL1=(IVAL1/(IVAL1-5))*(RVAL2/IPAR)**-2-RPAR/(2.0*RVAL1)`
(i) `IVAL3=IVAL1+DPAR*IVAL2/(DVAL1+FVAL1)`
(j) `FVAL3=(DVAL1+DVAL2+FVAL1)**IVAL3+(2.0D1+IVAL3)/(DVAL2-IPAR)`
(k) `IVAL2=RVAL1+DPAR*(RVAL2-DVAL2)`
(l) `IVAL2=(RPAR-RVAL2*3.6DO)/(DPAR+1.6)**(3.0*RVAL1)`
(m) `IVAL3=IPAR*(RVAL1-DAPR)/IVAL1`
(n) `IVAL1=RPAR*(IVAL1+IVAL2+DPAR)/(1.2*IVAL2+IPAR/2)`

9. Apply the instructions given in Problem 7 to each of the following statements. Use the same values given in Problem 7:

(a) `DVAL3=DPAR*DPAR*(DVAL1-2.D0*DVAL2)`
(b) `DVAL2=DVAL2+(DPAR/(DVAL1+DVAL2/6.D-1))`
(c) `DVAL3=RPAR*(RVAL1+2.6*RVAL2)`
(d) `DVAL3=(RVAL1+RVAL2+RPAR)**1.1/(3.6*RVAL1-RVAL2)`
(e) `DVAL2=(IVAL1+FVAL1)/IVAL2+3*IVAL2`
(f) `DVAL3=IPAR**2*(IVAL1/IVAL2+IVAL1/IPAR)/IPAR*(4*IVAL2/IVAL1))`
(g) `DVAL1=DPAR+(IPAR*DVAL1)`
(h) `DVAL3=(DVAL1*IVAL1)**(IVAL2/IPAR)/(DVAL2+IVAL1/IPAR)`
(i) `DVAL1=DVAL1+RVAL1/DVAL1`

(j) DVAL2 = RVAL2* (DVAL1+ (RVAL1−3. 0D0) **1. 2)
(k) DVAL3 = RVAL1* (IPAR+2*RVAL2) /IVAL1)
(l) DVAL3 = (IVAL1/IVAL2) **RVAL1 + (RVAL2/RVAL1) **IPAR
(m) DVAL2 = IVAL1 + 3* (RVAL1+DPAR)
(n) DVAL3 = (DVAL1/ (. 25*IVAL1+RVAL1) ) ** (RVAL2*IPAR−RVAL1)

10. Show the sequence of operations that FORTRAN will follow in producing a result for each of the following arithmetic assignment statements. Do not include data conversions. Use T1, T2, T3, etc. as names for places where any intermediate results may be stored temporarily:

(a) R = A + B/C − 2 * D
(b) T = (A+B) /C − 2*D
(c) R = A + 2. 2 * B/C * D
(d) S = A + 3. 8*B/C*3*D
(e) T = X−4*B/ (C+5) *D
(f) V = W * (A−B) /C−3*D
(g) Y = (A+W) * (B+C*D) / (B+W) * (A*C−D)
(h) Z = (A+W) * (B*C+D) / ( (B+W) * (A*C−D) )
(i) R = X + (Y*Z*W) * (2* (X−Z*W) +W* (X+2*Z) ) / ( (X+Z) / (W+Y) )
(j) S = (X−4*B) **2/ (C*5) *D**2
(k) T = ( (X+Y**2) / (Z**3−3. 6*S) ) **1. 8
(l) Z = (A*B**C**X + 3. 1*B*X**D**Y) **0. 36

11. Modify either Example 5.4 or 5.4A so that the output appears in printed form on the standard system output unit rather than on the user's terminal. NOTE: Direction of the output to this destination is handled by the statement

PRINT *, list of output variables

12. If a certain amount of money PRINCIPAL is invested at an annual interest rate of INVRATE, and the interest is reinvested along with the original amount, the resulting process is called *compounding*. That is, the interest earns interest and the money grows faster than it would without compounding. At the end of MUMOFYRS years, the total amount TOTALAMT can be computed by the formula

$$\text{TTLAMT} = \text{PRINC}\,(1 + \text{INTRST})^{\text{NUMYRS}}$$

For example, if we invest $1000.00 at an annual interest rate of 11.5% (i.e., 0.115), the thousand dollars will have grown in three years to

$$1000(1 + 0.115)^3$$

or $1386.19. Using this formula, write a program that reads and processes any number of input sets, with each set consisting of values for PRINCIPAL, INVRATE, and NUMOFYRS (NUMOFYRS is an integer). For each set, the program is to display a line of output showing the input values and the computed value for TOTALAMT. The run is stopped by a PRINCIPAL value of 0.0.

13. Using the formula given in Problem 12, write a program that reads PRINCIPAL, INVRATE, and NUMOFYRS as before. After printing an echo of the input, the program prints (displays) a little table in which it shows the value of TOTALAMT after one year, two years, three years, etc., up to and including NUMOFYRS. Each pair of output values (the number of years and the TOTALAMT value after that many years) is to appear on a separate line. Leave two blank lines between input sets and stop the run with a PRINCIPAL value of zero.

14. AN AMERICAN TRADITION:
In many areas of our Great Land, it is now a misdemeanor to publish, sell, lease, or otherwise distribute a book, leaflet, videotape or other material dealing with or claiming to deal with programming if that material does not include the Indian Problem at least once. So here it is:

Though most of us were not present at the sale, we have it on good authority that, in 1624, the Canarsie Indians struck a deal with a group of Dutch settlers. In exchange for $24.00 worth of trendy costume jewelry and pretty good theatrical accessories, the smiling Canarsies unloaded Manhattan Island, a trouble spot for as long as anyone could remember. Hours later, the merchandise had already been converted to cash, and the money was socked away in the Arrowroot Fund at an annual interest rate of 6%. And there it has sat. Meanwhile, the Canarsies' descendants have been busy with other matters, so that each year the Council resolves to Get Around To It at their Next Meeting.

To strengthen their resolve, write a program that reads a single integer value NOW, representing the current year. Using that value and the year of the sale (1624, known to the Canarsies as MANNAHATCHIGPO, The Year We Got Rid of That Stinking Rock), and displays TOTALAMT, the current value of their investment.

15. As a somewhat more challenging version of Problem 14, display the value of the Canarsies' investment as time went on, showing that value for 1634, 1644, and so on, with the last line showing the current value.

16. Write a version of the program for Problem 15 in which the output is printed on the system output unit.

17. Using the compound interest formula from Problem 12, write a program that reads sets of values for PRINCIPAL and INVRATE. For each set, the program displays the input, along with DOUBLE_YRS, an integer indicating the minimum number of whole years required for the original amount to double.

18. (Special Challenge): Write a program to meet the requirements described in Problem 17. However, instead of producing DOUBLE_YRS to the nearest year, compute and display YRS2 and MONTHS2, the number of years and months to the nearest month required for the original investment to double.

19. While it is nice to invest PRINCIPAL dollars at a compound interest rate of INVRATE, there also is inflation, reducing the actual value (purchasing power) of the investment. Assuming that there is a constant rate of inflation, write a program that shows this effect. Specifically, your program is to process any number of input sets where each set consists of PRINCIPAL (the amount originally invested), the interest rate (INVRATE), the inflation rate (call it INFLATION—a 7% inflation rate is represented in INFLATION as 0.07), the year the investment is made (STARTYR), and the year for which we want the result (CURRENTYR). The program is to display the input, followed by a second line showing TOTALAMT, the total number of dollars, and TOTALPCH, the value of the investment expressed in terms of what a dollar was worth in the year STARTYR. Leave a blank line between displays for different input sets, and stop the run with a PRINCIPAL value of zero.

    For example, suppose we invested $1000.00 in 1979 at 10% and the inflation rate was 6%. (Wouldn't that be nice?) After one year (i.e., CURRENTYR = 1980), TOTALAMT would be $1100.00. However, with inflation at 6%, the value of each 1980 dollar would be 0.94 (1.0 − 0.06) 1979 dollars, so that TOTALPCH would be `0.94*1100` or $1034.00. After two years, TOTALAMT would be $1210.00, but that 6% is still there, wearing away at the money's value, so that a 1981 dollar at that rate, is worth 88.36 cents. Consequently, our 1210 1981 dollars have a TOTALPCH value of $1069.16 in terms of 1979 dollars. Oh my.

20. As a more challenging version of Problem 19, write a program in which each input set consists of PRINCIPAL, INFLATION, and NUMOFYRS. PRINCIPAL and INFLATION have the same meanings as in Problem 17, and NUMOFYRS is an integer indicating the number of years for which the investment is to sit. For each set, the program is to display the input followed by BRK_EVEN, the rate at which the investment must earn interest in order to stay even with inflation. That is, the purchasing power of the total amount at the end of the investment period would be the same as that of the original investment at the beginning of the investment period. A PRINCIPAL value of zero stops the run.

21. A straight line, when displayed on a set of rectangular coordinates, can be described by the equation

    Y = A0 + A1*X

    where A1 is the *slope* of the line, a value indicating how rapidly Y's value changes as X changes; A0, the *intercept*, specifies what Y's value is when X is zero. Since A0 and A1 are constant values for a given line, we can define the Y-value (say YI) of any point on that line in terms of its X-value (XI) and these constants, i.e.,

    YI = A0 + A1*XI

    Thus, if we know the X-Y values for any two points on a line, we can develop a description for that line by using these values to solve for A0 and A1. This is summarized in Figure 5.10.

    Using these relationships, write a program that reads and processes any number of input sets where each set consists of four real values X1, Y1, X2, and Y2. For each set, the program is to produce three lines of output: The first line shows the character string `X VALUES:` followed by X1 and X2; the second line shows the character string `Y VALUES:` followed by Y1 and Y2. The third line shows the slope (A1) and intercept (A0), each preceded by an identifying character string. For instance, if we process an input set consisting of the four values shown below,

    ```
    1.0     2.0     5.0     4.0
    ```

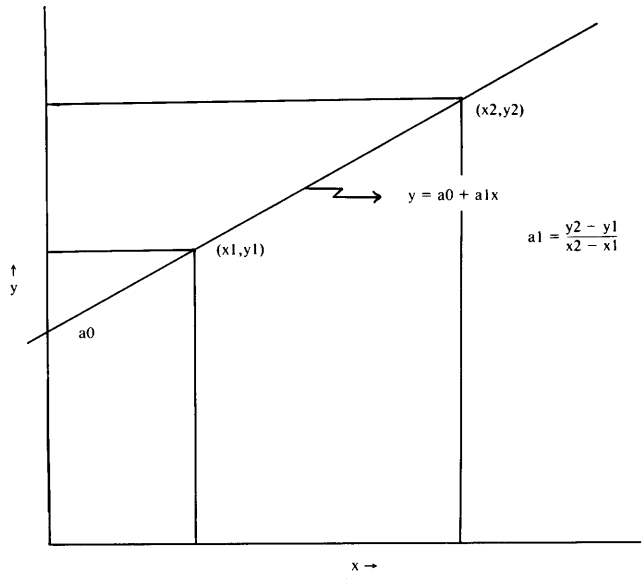



**FIGURE 5.10** Calculation of the Slope of a Line.

the output should look like this:

```
X VALUES:     0.100000E 01     0.500000E 01
Y VALUES:     0.200000E 01     0.400000E 01
SLOPE:     0.500000E 00        INTERCEPT:     0.150000E 01
```

Output sets should be separated by a blank line. To stop the run, use an input set (that is not processed) in which X1 and X2 are the same. Here are some suggested input sets:

| *X1* | *Y1* | *X2* | *Y2* |
|---|---|---|---|
| 0.0 | 0.0 | 6.0 | 6.0 |
| 0.0 | 3.0 | 3.0 | 6.0 |
| 1.0 | 1.0 | 2.0 | −1.0 |
| 5.4 | 8.6 | 12.6 | 10.0 |

22. Using the relationships discussed in the previous problem, write a program that processes a succession of input sets. This time, each input set consists of six real values describing three points: X1, Y1, X2, Y2, and X3, Y3. For each set, produce four lines of output: The first three lines are the same as for the previous problem, using (X1, Y1) and (X2, Y2) to determine the slope and intercept. The fourth line is to show X3 and Y3, followed by one of two messages (character strings): If (X3, Y3) lies on the line formed by the first two points, the message says POINT 3 IS ON THE LINE. If not, the message says POINT 3 IS NOT ON THE LINE. Stop the run as in the previous problem.

NOTE: In designing an algorithm to solve this problem, it is likely that, at some point, you will need to test whether two numerical values are the same. Because of the fact that each memory element can represent a value with a certain range, it is possible to compute numbers that are supposed to be exactly the same but are represented just slightly differently. We may choose to think of them as being close enough to each other so that they can be treated as being the same. However, the computer and FORTRAN (its agent) "sees" them as being different, and the .EQ. test will fail. To overcome this difficulty (it certainly is not the programmer's fault), such tests are set up to expect these small differences. Thus, instead of saying:

```
IF (X .EQ. Y)
```

we would construct the test to say, for instance,

```
IF (X-Y) .LE. 0.0001)
```

Now, we are testing whether X and Y are within 0.0001 of each other. (This assumes X is greater than Y.) The value 0.0001 is used here for illustration. In each situation, of course, it is up to the programmer to determine how far apart the two numbers must get before he or she is not willing to treat them as being "equal."

23. There are many instances in experimental work where data are produced as a series of (X,Y) points. Because of a variety of circumstances, there often are inconsistencies in the X and Y values so that, even though it is clear that the relationship between the X and Y values can be described by a straight line, it is impossible to draw a single line that will pass through all of the experimental points. When this happens, we look for the line that provides the best description of the data. "Best" does not necessarily mean that it will be the line that goes through the most points. Instead, it will be the line around which the points cluster most closely. Such a line, for a given set of X-Y data, is called the *least squares* line, and its slope ($A_1$) and intercept ($A_0$) can be computed as follows:

$$A_1 = \frac{N\Sigma X_i Y_i - \Sigma X_i \Sigma Y_i}{N\Sigma X_i^L - (\Sigma X_i)^L}; \quad A_0 = \frac{\Sigma Y_i - A_1 \Sigma X_i}{N}$$

where

$N$ = no. of X-Y points
$X_i$ = a single X value
$Y_i$ = a single Y value

Write a program that processes any number of X-Y points and produces the values of $A_0$ and $A_1$ that describe the least squares line for those points. All X and Y values are real numbers, with each input line containing a point (the X value followed by the Y value). For purposes of this problem, no X value in the data will be below −10.0. The program is to compute and display NUMPTS, the number of X-Y points processed, $A_1$ (the slope), $A_0$ (the intercept), X_AVG (the average value of X), and Y_AVG (the average value of Y). NUMPTS appears on one line, $A_0$ and $A_1$ on the second line, and X_AVG and Y_AVG on the third line.

24. As an additional challenge, expand the program in the previous problem so that it processes any number of sets of X-Y data, producing for each the three lines of output described above, separated by two blank lines. After the last set has been processed, the program is to leave three blank lines and show the following:

    1. NUMSETS (the number of sets processed);
    2. MAX_SLOPE (the value of the largest slope);
    3. MAX_SLP_NO (the number of the set whose slope is MAX_SLOPE. The first set is no. 1, etc.);
    4. MIN_SLOPE (the value of the smallest slope);
    5. MIN_SLP_NO (the number of the set whose slope is MIN_SLOPE);
    6. MAXPTS (the largest number of points in a set);
    7. MAXPTS_NO (the number of the set having the most points);
    8. MINPTS (the smallest number of points in a set);
    9. MINPTS_NO (the number of the set having the least points).

    You may assume that there will be only one maximum and minimum slope in a collection of input sets, and that single sets will have the maximum and minimum number of points. Moreover, you may assume that every set will have enough points (at least two) for the least squares algorithm to work properly. Finally, you may assume that each point is complete (i.e., both X and Y values are present.)

25. Modify the program specified in Problem 24 so that it does not make the last two assumptions listed above. That is, a particular "set" of X-Y data might have only one point, or even no points. Moreover, there may be points with the X value or the Y value missing (never both).

# 6
# *Computations with Built-in Functions*

The arithmetic operations discussed in the previous chapter are only a small part of FORTRAN's computational services. The language also includes a wide range of other operations which, though generally more complex than the five "basic" ones, are equally convenient to use.

These computational features are provided as a collection of *built-in functions.* Each of these functions actually is a prewritten program with its own name and place in a library that is part of the language. Accordingly, when one of these functions is needed, the program refers to it by name, and FORTRAN includes a copy of the appropriate instructions in that program. (The compiler keeps track of such references, called *invocations*, so that only a single copy is included no matter how many references are made.) To illustrate such an invocation, let us assume that A, B, and C all have been declared as real variables and that A and B each have values in them. Then, the assignment statement

```
C = A + SQRT(B)
```

produces the following activities:

1. FORTRAN, noting the reference to the built-in function named SQRT, uses SQRT to operate on B. Since the SQRT function is designed to produce the square root of the number given to it, the result will be that the square root of the value in B is made available to the expression.
2. This value and A's value are added together.
3. The result of the addition is stored in C.

The name SQRT is associated permanently with the particular program in the FORTRAN library that computes square roots. Similarly, each of the other built-in functions has its own name, selected to indicate the kinds of operations that the function performs.

The assignment statement shown above is a typical example of how a built-in function is used. The quantity inside the parentheses (B in the example) is the *argument* on which the function is to operate. FORTRAN delivers a number to the function and the function, after completing its task, returns the result. This automatic two-way transmission process is activated when the function is invoked. It has nothing to do with the way in which the transmitted value was developed. Consequently, the argument can be anything that produces a number for the function to process. For instance, in the statement

```
C = A + SQRT(3.0)
```

a value of 3.0 is delivered to the square root function. The same delivery mechanism applies to a more complicated invocation like the one in the statement

```
C = A + SQRT(3.0/(B*A))
```

In this statement the argument is an expression that FORTRAN must evaluate in order to produce the numerical result that will be delivered to SQRT. Accordingly, FORTRAN

1. Multiplies B by A (call the result T1).
2. Divides the constant 3.0 by T1 (call the result T2).
3. Delivers T2 to the square root function SQRT.
4. Returns a value from SQRT representing the square root of T2 (call that value T3).
5. Adds T3 to A (call the result T4).
6. Stores T4 in C.

The automatic mechanism for delivering numbers to functions is used for a wide variety of functions including those requiring more than one argument. This chapter examines a number of these convenient processes and illustrates their use.

## 6.1 FUNCTIONS FOR DATA CONVERSION

In the previous chapter we saw that FORTRAN used some automatic mechanisms to convert numerical values from one type to another in order to meet the requirements imposed by its rules of arithmetic or by the programmer's declarations. These conversion processes also are available to the programmer as built-in functions so that he or she can specify such changes at any time.

### 6.1.1 Conversion to Integer: The INT Function

Values represented (stored) as single or double precision numbers can be converted to integer form by the built-in function INT. As an example, consider the following group of statements:

```
      REAL*8        D1,D2
      REAL*4        R1,R2
      INTEGER*2     I1,I2
1  R1 × 36.9
2  D1 = 540.8
3  I1 = INT(R1) + INT(D1)
4  I2 = INT(R1+D1)
5  R2 = INT(R1) + INT(D1)
```

We shall go through these events statement by statement, so that we develop a detailed idea of what happens. This will establish a general understanding of the type of process that occurs with the other conversion functions:

*Statement 1:* A value of 0.3690000E+02 is stored in R1.

*Statement 2:* A value of 0.5408000000000000D+03 is stored in D1.

*Statement 3:* INT is invoked twice. The first time, it returns a value of 36. (Let us say the 36 is stored temporarily in T1.) The second time, it returns a value of 540. (Let us say the 540 is stored temporarily in T2.) T1 and T2 are added to produce the integer 576, which then is stored in I1.

*Statement 4:* R1 is converted to REAL*8 (in accordance with the automatic arithmetic rules as discussed in Chapter 5) and the result (0.3690000000000000D+02) is stored temporarily (let us say in T3). Then T3 is added to D1, giving the REAL*8 result 0.5777000000000000D+03. This is stored (let us say in T4). *After all that is done,* INT is invoked to convert T4 to an integer (577), which is delivered to some temporary place (say T5) and stored from there in I2. Note that I1 and I2 have different results because in statement 4 the addition was done *before* the INT function was invoked, so that the fractional portions still were available to contribute to the sum.

*Statement 5:* Since the expression in this statement is identical to the one in statement 3, exactly the same computations will take place, producing the same result. Thus, when FORTRAN is finished with the expression, it is ready to deliver a result of 576. Now, however, since the destination (R2) is a real variable, unlike I1 in statement 3, FORTRAN will be forced to convert the 576 to real, and a value of 5.760000E+02 will be stored in R2.

### 6.1.2 Conversion to Real Numbers: The REAL Function

An INTEGER*2, INTEGER*, or REAL*8 value can be converted to a REAL*4 number by invoking the REAL function. For example, if R1 is a real variable, D1 is REAL*8 and I1 is an integer, the statement

```
R1 = 2.6 * (REAL(I1)-REAL(D1))
```

causes the following actions:

1. I1 and D1 each are converted (by a separate invocation) to REAL*4.
2. The converted D1 is subtracted from the converted I1.
3. The result of the subtraction (a real number) is multiplied by the real value 2.6.
4. The product is stored in R1. No further conversion is needed since R1 is a real variable.

When a 32-bit integer is converted, FORTRAN will preserve as much of the precision as the real form can accommodate.

### 6.1.3 Conversion to Double Precision: The DBLE Function

A real or integer value can be converted to REAL*8 by invoking DBLE. For instance, if R1, R2, and R3 are real variables, the statement

```
R1 = DBLE(R2) * DBLE(R3)
```

causes R2 and R3 each to be converted to REAL*8. Then, the multiplication is performed and the result, a REAL*8 number, is converted to real for storage in R1. Thus, FORTRAN does the computation using double precision arithmetic even though the participating variables are single precision.

### 6.1.4 Conversion of Complex Numbers

Consistent with the ground rule established earlier, we shall deal with complex numbers in a separate section of the book (Appendix A).

## 6.2. BASIC NUMERICAL MANIPULATIONS

FORTRAN's library of bult-in functions includes a variety of services for manipulating numerical values. To make their examination convenient, we shall use the general name arg to indicate the argument given to the function. When a particular function uses more than one argument, we shall use the names arg1, arg2, and so on.

### 6.2.1 Operations with a Number's Sign

There are two functions relating to the sign of a number:

**6.2.1.1 The ABS Function** ABS(arg) provides the absolute value of arg. There is no conversion in the process.

**6.2.1.2 The SIGN Function** The SIGN function makes it possible to transfer the sign from one numerical value to another. SIGN (arg1, arg2) gives arg1 the same *sign* (not the same value) as arg2 and returns the new value of arg1. For example, if AMT1 has the value 23.76 and AMT2 has the value −166.889, the statement

```
Y = SIGN (AMT1, AMT2)
```

results in a value of −23.76 in Y and AMT1, and AMT2 retains its original value (−166). If arg's value is 0, the function value is not known and cannot be used.

### 6.2.2 Positive Differences (DIM)

This function returns arg1−arg2 if arg1 is greater than arg2. Otherwise, it returns a value of 0. Suppose that V1's value is 26.4 and V2's value is 17.2. The statement

```
Y = DIM (V1, V2)
```

places a value of 9.2 in Y. If we had written

```
Y = DIM (V2, V1)
```

instead, the resulting value in Y would be 0.

### 6.2.3 Double Precision Multiplication— The DPROD Function

If we multiply 2.2 * 2.2, the result (4.84) requires more decimal places than either of the two factors. In certain cases, this type of operation with real numbers can produce a fraction that will not fit a REAL*4 result. Consequently, FORTRAN provides a convenient way to specify REAL*8 multiplication. If arg1 and arg2 are real arguments, the expression

```
DPROD (arg1, arg2)
```

will persuade FORTRAN to use 64-bit arithmetic when the indicated multiplication is performed. The result, then, will be a REAL*8 number with none of the fractional portion lost. Of course, once that result is produced, it is up to the programmer to use it properly (e.g., place it in a REAL*8 variable, rather than one that is REAL*4).

### 6.2.4 Obtaining a Remainder— The MOD Function

This function, one of the most convenient in the language, divides arg1 by arg2 and returns the *remainder*, i.e., the amount left over. The same rules of arithmetic are used here as everywhere else in FORTRAN, so that the type of remainder produced by the function depends on the type of division that we direct the function to perform. To make this clear, we shall illustrate the operation of the MOD function using constants for arg1 and arg2. In the three cases shown below, the two arguments will have the same respective values, but the forms will change:

1. MOD (30, 13):
   Since the two arguments already are integers, FORTRAN does integer division. The remainder (4), therefore, is an integer.
2. MOD (30.0, 13.0):
   Now, the division is done in real arithmetic, and the result is delivered as a real whole number (4.0).
3. MOD (30.0D0, 13.0D0):
   This time, the division is done in double precision arithmetic and the remainder has the value 4.0D0.

If the MOD function did not exist, we still could get hold of a remainder by using a little trickery. This is an example where we can take advantage of integer division. Let us assume that `I1` is an integer variable to which we want to assign the remainder of `30/13`. If we let FORTRAN divide `30` by `13` (as integers), the result will be `2`, with the fractional portion gone forever (actually, gone to Carson City, if you paid attention). In this case, that is exactly what we want, because if we take that result and multiply it back by the divisor (`13`), we shall get a number that is different from the original dividend by an amount equal to the remainder. Specifically `(30/13) * 13` will give us `26`, since FORTRAN will do the integer division, followed by the multiplication. Then, `30-(30/13)*13` will produce `4`, the same value as `MOD (30, 13)`. Thus,

```
I1 = 30-(30/13)*13
```

and

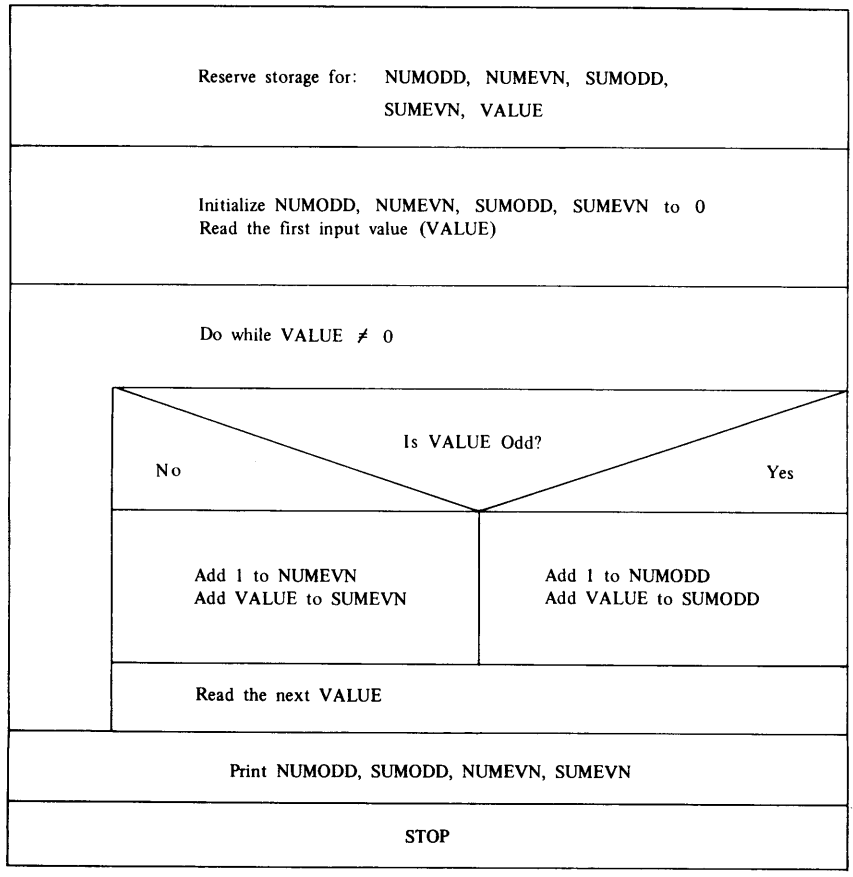
```
I1 = MOD(30,13)
```

will produce identical results.

**Example 6.1** One of the most important skills that a programmer needs to develop is the ability to convert directions in natural language (e.g., English, Spanish, Japanese, etc.) to statements in FORTRAN that will get the same thing done.

The requirement is to read a set of integer values (all of them greater than 0) and print:

the sum of all the odd values
the number of odd values read
the sum of all the even values
the number of even values read




(a)

"Reserve storage for NUMODD, SUMODD, NUMEVN, SUMEVN, VALUE."
"Set NUMODD, NUMEVN, SUMODD, SUMEVN to zero."
"Read the first input item (VALUE)."
**while** value is not equal to zero:
  "Display VALUE."
  **if**
    "VALUE is even"
  **then**
    "Add 1 to NUMEVN."
    "Add VALUE to SUMEVN."
  **else**
    "Add 1 to NUMODD."
    "Add VALUE to SUMODD."
  **endif**
  "Read the next input item (VALUE)."
**endwhile**
"Print NUMODD, SUMODD, NUMEVN, SUMEVN."
"Stop."

(b)

**FIGURE 6.1** **(a)** Structured Flowchart for Example 6.1. **(b)** Pseudocode Representation for Example 6.1.

```
************************************************************
*                          EXAMPLE 6.1                     *
************************************************************
*   NUMODD:  THE NUMBER OF ODD INTEGERS READ IN            *
*   NUMEVN:  THE NUMBER OF EVEN INTEGERS READ IN           *
*   SUMODD:  THE SUM OF THE ODD INTEGERS                   *
*   SUMEVN:  THE SUM OF THE EVEN INTEGERS                  *
*   VALUE:   AN INDIVIDUAL INPUT VALUE                     *
************************************************************

      PROGRAM     EX601
      IMPLICIT    NONE
      INTEGER*2   NUMODD,NUMEVN,SUMODD,SUMEVN,VALUE

      NUMODD = 0
      NUMEVN = 0
      SUMODD = 0
      SUMEVN = 0
      PRINT *, 'ENTER VALUE'
      READ *,  VALUE

************************************************************
* WE SHALL IDENTIFY EVEN NUMBERS AS BEING DIVISIBLE BY 2   *
* WITH NO REMAINDER. ACCORDINGLY, WE TEST WITH MOD.        *
************************************************************

      DO WHILE  (VALUE .NE. 0)
         PRINT *, 'VALUE = ',VALUE
         IF (MOD(VALUE,2) .EQ. 0) THEN
           NUMEVN = NUMEVN + 1
           SUMEVN = SUMEVN + VALUE
         ELSE
           NUMODD = NUMODD + 1
           SUMODD = SUMODD + VALUE
         ENDIF
         PRINT *, 'ENTER NEXT VALUE'
         READ (*,END=99) VALUE
      END DO

   99 PRINT *, 'NO. OF ODD VALUES = ',NUMODD
      PRINT *, 'SUM OF ODD VALUES = ',SUMODD
      PRINT *, 'NO. OF EVEN VALUES = ',NUMEVN
      PRINT *, 'SUM OF EVEN VALUES = '.SUMEVN
      STOP

      END
```

**FIGURE 6.2** FORTRAN Statements for Example 6.1.

Clearly, we need to set up two sums (we shall name them SUMODD and SUMEVN) and two counters (NUMODD and NUMEVN). The crucial issue will be how to look at a given input value (which we shall name VAL) and determine whether it is odd or even. The flowchart and pseudocode (Figure 6.1) tell us what we need to do and when we need to do it. How is another question.

```
VALUE =              121
VALUE =              212
VALUE =              333
VALUE =              342
VALUE =              556
VALUE =              302
VALUE =              654
VALUE =              211
VALUE =              455
VALUE =              754
VALUE =              655
VALUE =              855
VALUE =              320
NO. OF ODD VALUES =              6
SUM OF ODD VALUES =           2630
NO. OF EVEN VALUES =             7
SUM OF EVEN VALUES =          3140
```

**FIGURE 6.3** Output for Example 6.1.

Since the MOD function enables us to use any divisor we wish, we can take advantage of the fact that all even numbers are divisible by 2, and odd numbers are not. Another way of saying that, somewhat closer to what we know we can do in FORTRAN, is that an even number, when divided by 2, leaves a remainder of 0; an odd number does not.

This gives us the algorithmic solution we need, and we can proceed with the program (Figure 6.2). A sample run is shown in Figure 6.3.

### 6.2.5 Rounding and Truncation

Since a declaration reserves a certain amount of storage for a particular type of data, any value placed in such a variable will fill the storage available for it. For instance, if a real number can be expressed to six decimal places, it will always be expressed that way. There are many occasions where we would like to control the form of numerical values. (For example, "compute the income to the nearest cent" implies that we are not interested in anything after the second decimal place.) Several built-in functions are available to help the programmer exercise this type of control.

**6.2.5.1 Truncation: The AINT Function** AINT truncates a single or double precision value without converting it. (Recall that INT, described in Section 6.1.1, converts to integer.) Thus, for example, if real variable R1 has a value of 247.8990, the expression AINT(R1) produces a value (still real) of 247.0000. (INT(R1) would produce a value of 247.)

**6.2.5.2 The Nearest Integer: The NINT Function** NINT uses a 32-bit or 64-bit value as follows:

> If the value is positive, it adds 0.5 to it and then truncates. If the value is negative, it subtracts 0.5 from it and then truncates. In both cases, the result is converted to integer.

For instance, if R1 is a real variable with a value of 335.9087, NINT(R1) will produce a value of 335.9087+0.5000000 or 336., which then is converted to the integer 336. On the other hand, if R1 were to have a value of −335.9087, the expression NINT(R1) would produce a value of −335.9087−0.5000000 or −336.4087, which then is converted to the integer −336.

**6.2.5.3 The ANINT Function** ANINT works exactly the same way NINT does, except that it does not perform the final conversion to integer. It adds or subtracts 0.5 and truncates the result, leaving the value unconverted. Using the previous value of −335.9087 for R1, ANINT(R1) produces a value of −336.0000.

**6.2.5.4 Rounding Techniques** The ANINT and NINT functions give us the basic tools for rounding numerical values to the number of places we want. The basic technique involves three simple steps:

1. Multiplying the number to be rounded by an appropriate power of ten.
2. Applying ANINT or NINT to the adjusted number.
3. Dividing the result by the same power of ten used in step 1 to readjust the value to its proper magnitude.

What is meant by "an appropriate power of 10"? We need to adjust the value so that when the function adds or subtracts 0.5 to it, the 0.5 affects the proper decimal place.

We shall see exactly how this works by using the real variable R1 with a value of 335.9087 and another real variable R2 in which we shall place a rounded value.

*(a) Rounding to the nearest integer:* 335.9087, when rounded to the nearest integer, produces a value of 336. Consequently, if we add 0.5 to the original value and then discard the fractional part, we shall get what we want (335.9087+0.5 = 336.4087; discarding the .4087 leaves the desired value.) This means that use of either the ANINT or NINT function, as is, will do the rounding. The choice of the function will depend on the form in which we want the result. If we assume that the result is to be assigned to R2 (which is real), then we shall use ANINT, since no conversion is needed:

```
R2 = ANINT(R1)
```

*(b) Rounding to the nearest tenth:* 335.9087, rounded to the nearest tenth (i.e., to one decimal place), gives a result of 335.9. (Since the second decimal place is 0, it has no effect on the final value.) If we were to add 0.5 to the original value, the result obviously would be wrong. The adjustment we want to make in order to perform the operation correctly is to add 0.05 (not 0.5) to 335.9087, and then discard the part of the fraction we do not need. Since we are interested in the first decimal place, we would produce 335.9087+0.05 or 335.9587 and keep the 335.9. The ANINT function, by definition, always uses 0.5, and we cannot adjust it. Instead, we can adjust the 335.9087 to get the same result. If we multiply R1 by 10 and apply ANINT to *that* value, we can get the correct digits. Then, if we divide by 10, undoing what we did at first, we shall have the rounding we want. Specifically,

```
335.9087 * 10 = 3359.087
ANINT (3359.087) =3359.087+0.5 = 3359.587 truncated to
                                                3359.000
3359.000/10.0 = 335.9000
```

Thus, to round to the nearest tenth, we multiplied by 10.0, applied the ANINT function, and then readjusted by dividing by 10.0:

```
R2 = AINT (10.0*R1) / 10.0
```

R2 will have the value 335.9000.

*(c) Rounding to the nearest hundredth:* To establish the general technique, we shall round 335.9087 to the second decimal place, i.e., to the nearest hundredth. (The desired value, obviously, is 335.91.) As before, we must adjust the value so that ANINT, by adding 0.5, will produce the proper digits. Since we want two decimal places, the 0.5 must affect the third place (that is, one place to the right of the last one in which we are interested). Consequently, we shall multiply R1 by 100.0 this time, producing a value

(33590.87). The decimal point has been moved so that the digit previously in the third decimal place is now just to the right of the point. When we apply ANINT to this adjusted value, it will add 0.5, producing 33591.37. Then, it will truncate, leaving a result of 33591.00. We readjust this by taking out (dividing by) the 100.0. The final value, then, is 335.91 as required. Thus, our assignment says:

```
R2 = ANINT(100*R1)/100.0
```

*(d) Rounding to the nearest hundred:* We should not go under the impression that rounding applies only to fractions. There are many occasions when a number needs to be rounded to the nearest ten, or hundred, or even to the nearest million. The same technique is applied as before, with the only difference being in the power of ten that is needed for adjustment. To round to the nearest 10, we would adjust by multiplying by 0.1 (10 ** (−1)), apply ANINT, and readjust. To illustrate, let us suppose that R1, with a value of 68687.73, represents a dollar amount submitted as a bid for constructing a free-form pigsty. The person looking at the bids is interested in numbers to the nearest hundred dollars in order to simplify the comparisons. If R2 is the variable to which the rounded result is to be assigned, we can compute R2 by writing

```
R2 = ANINT (R1/100.) * 100.0
```

As a result, FORTRAN computes R1/100.0, giving a temporary result of 686.8773. ANINT applied to this value produces 686.8773+0.5 or 687.3773, truncated to 687.0000. Then, it multiplies the 687.0000 by 100.0, readjusting it to 68700.00, the result our professional pigsty promoter wants.

### 6.2.6 Functions for Extreme Values

The largest and smallest values in a specified collection of numerical items can be found by using the MAX and MIN functions, respectively.

**6.2.6.1 The MAX Function** This function selects the largest value from a list of arguments that can have any number of items. MAX gives a *value*, but there is no indication as to where it came from. For example, if we say

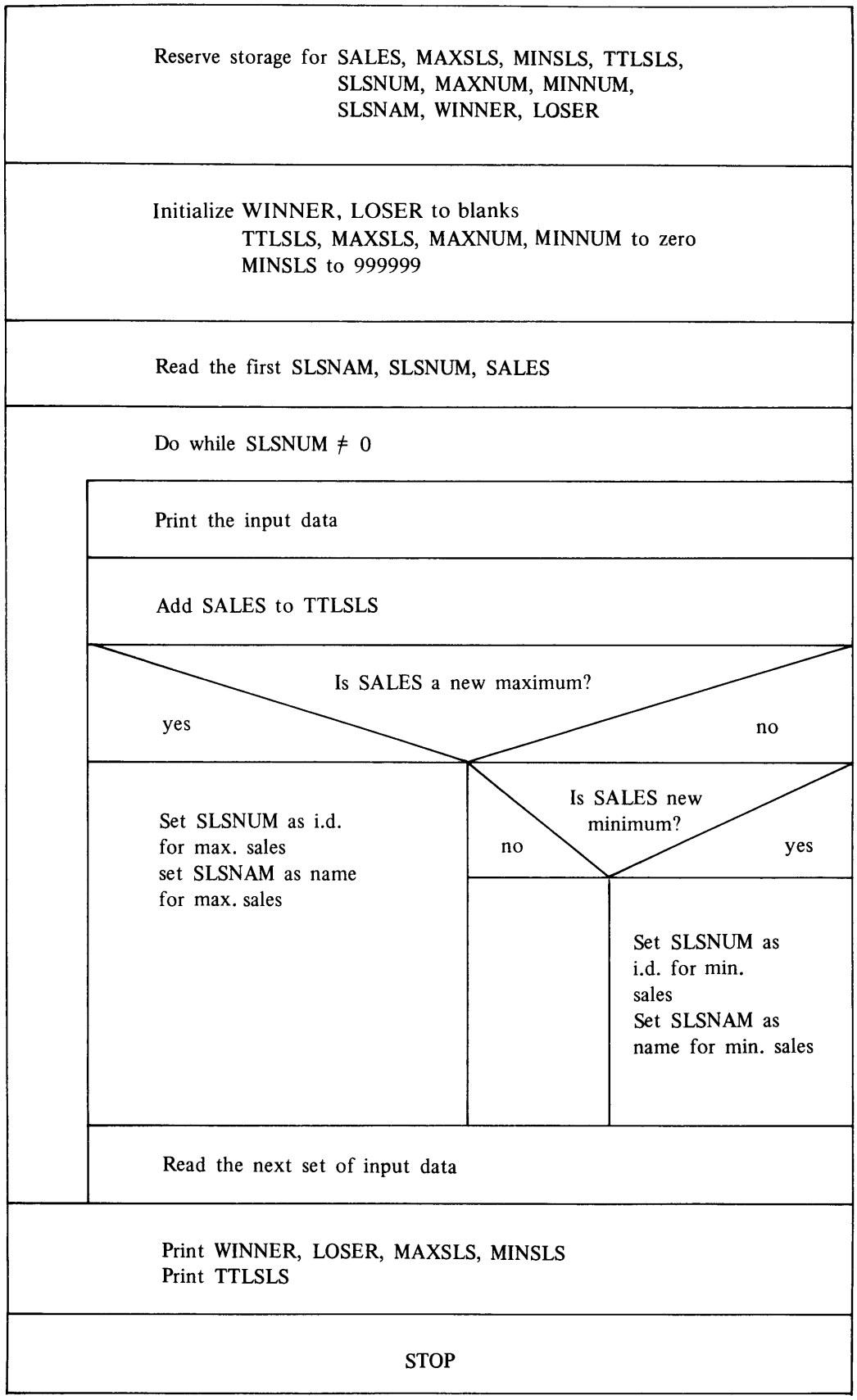
```
BIG = MAX (X, 3.0/Y, SUM, W+Z)
```

the variable BIG will receive the largest of the values represented by the four arguments. However, unless our program includes statements to perform additional tests, we shall not know which one was the largest. (If we just want to know how much, and not which one, then MAX gives us exactly what we want.)

**6.2.6.2 The MIN Function** MIN works exactly like MAX, producing the smallest value of a specified group. Again, there is no information regarding the exact source of that minimum.

We shall examine an example program in which minimum and maximum values are emphasized. The requirements are set up so that it will not be enough just to know the values themselves. Additional information will be needed such that the solution cannot be handled only with MIN and MAX. This will give us an opportunity to explore some useful programming techniques in addition to illustrating the use or built-in functions.

**Example 6.2** Joe's Travel Service ("Those Who Know Go With Joe") has a large sales staff that sells trips and tours to exotic places. These people are rewarded with monthly prizes for highest sales (a week in Brazil, Indiana) and lowest sales (ten days in Brazil, Indiana). A program is needed to determine the prizewinners. Each month, a set of input is prepared for each salesperson showing the name, employee number and total sales. Employee number is an integer between 1 and 9900 (never zero) and the sales are reported to the nearest cent. Since this business is very hectic, it is likely that the number of salespeople




**(a)**

**FIGURE 6.4** **(a)** Structured Flowchart for Example 6.2.

```
"Define SALES, MAXSLS, MINSLS, TTLSLS, SLSNUM, MAXNUM, MINNUM, SLSNAM, WINNER, LOSER."
"Set WINNER and LOSER to blanks."
"Set TTLSLS, MAXSLS, MAXNUM, MINNUM to zero."
"Set MINSLS to 999999."
"Read the first values for SLSNAM, SLSNUM, SALES."
while SLSNUM is not equal to zero:
  "Print the input data."
  "Add SALES to TTLSLS."
  if
    "SALES is a new maximum value"
  then
    "Set SLSNUM as i.d. for new maximum."
    "Set SLSNAM as the new name for maximum."
  else
    if
      "SALES is a new minimum"
    then
      "Set SLSNUM as i.d. for mininum."
      "Set SLSNAM as name for minimum."
    else
    endif
  endif
  "Read the next set of input data."
endwhile
"Print WINNER, LOSER, MAXSLS, MINSLS."
"Print TTLSLS."
"Stop."
```

**(b)**

**FIGURE 6.4** **(b)** Pseudocode for Example 6.2.

will change from one month to the next. To keep it simple, we shall say that there will be no two people with the same total sales in any given month. Furthermore, after printing the prizewinners' names, numbers and total sales, the program is to print the overall sales for the firm, rounded to the nearest dollar.

One part of the required program is simple enough: We need to keep a running total of all sales figures so that the overall amount can be printed after all the data have been read and processed. For this purpose, a variable (TTLSLS) will be initialized to zero and each sales figure (SALES) will be added to TTLSLS. The other part, i.e., finding the maximum and minimum salespeople and amounts, will require somewhat more intricate processing. Our algorithm will keep checking each newly read sales figure so that, at any given time, the program has the largest and smallest sales figures found thus far, along with the corresponding names. If the sales figure just read exceeds the largest one found so far, the new value replaces the old maximum. (Alternatively, the same thing would happen with respect to the old minimum if the new value turned out to be less than the smallest one read so far.) Consequently, at the end of the run, we shall have the required results regardless of the number of input values. This is shown as a structured flowchart in Figure 6.4(a) and as pseudocode in figure 6.4(b). The FORTRAN program is given in Figure 6.5, and a sample run is seen in Figure 6.6.

## 6.3 COMPUTATIONAL FUNCTIONS

At the beginning of this chapter, we used the square root function (SQRT) as an example of how a built-in function works. This is one of a collection of functions designed to provide convenient aids for performing more extensive computations. Now that we are familiar with the general ideas underlying the use of this library, it will not be necessary to deal with each function in detail. Instead, we shall summarize the major groups of functions and their operation.

```
************************************************************
*                        EXAMPLE 6.2                       *
************************************************************
*  THIS PROGRAM READS AND PROCESSES AN ARTIBRARY NUMBER OF  *
*  DATA LINES, EACH CONTAINING AN EMPLOYEE'S NAME, NUMBER,  *
*  AND SALES FOR A GIVEN MONTH. THE PROGRAM ECHOES EACH     *
*  LINE AND, AT THE END OF THE RUN, IT PRINTS THE NAMES,    *
*  NUMBERS, AND SALES AMOUNTS FOR THE EMPLOYEES WITH THE    *
*  MOST AND LEAST SALES FOR THAT MONTH. FINALLY, IT PRINTS  *
*  THE TOTAL SALES FOR THE MONTH TO THE NEAREST DOLLAR.     *
*  SLSNAM: EMPLOYEE'S NAME;    SLSNUM: EMPLOYEE'S NUMBER    *
*  WINNER: WINNER'S NAME;      LOSER:  LOSER'S NAME         *
*  MAXNUM: WINNER'S NUMBER;    MINNUM: LOSER'S NUMBER       *
*  MAXSLS; WINNER'S SALES;     MINSLS: LOSER'S SALES        *
*  SALES:  EMPLOYEE'S SALES;   TTLSLS: FIRM'S TOTAL SALES   *
************************************************************

     PROGRAM         EX602
     IMPLICIT        NONE
     REAL*4          SALES,MAXSLS,MINSLS,TTLSLS
     INTEGER*2       SLSNUM,MAXNUM,MINNUM
     CHARACTER*25    SLSNAM,WINNER,LOSER

************************************************************
*                     INITIALIZATION                       *
************************************************************
*  WINNING AND LOSING NAMES ARE SET TO BLANKS SO WE KNOW THE*
*  VALUES AT THE START. TTLSLS IS SET TO ZERO SO WE CAN ADD *
*  TO IT IN OUR LOOP. MAXSLS IS SET TO ZERO SO THE FIRST IN-*
*  PUT VALUE AUTOMATICALLY WILL BE THE NEW MAXIMUM AGAINST  *
*  WHICH SUBSEQUENT VALUES CAN BE COMPARED. THE SAME THING  *
*  IS DONE WITH MINSLS BY INITIALIZING IT TO A VERY HIGH VA-*
*  LUE.                                                     *
************************************************************

      WINNER = '
      LOSER = WINNER
      TTLSLS = 0.0
      MAXSLS = 0.0
      MINSLS = 999999.0
      MAXNUM = 0
      MINNUM = 0
      PRINT *, 'ENTER SLSNAM, SLSNUM, SALES VALUES'

      READ *, SLSNAM,SLSNUM,SALES

************************************************************
*                    PROCESSING LOOP:                      *
************************************************************
*  MAX AND MIN FUNCTIONS ASSIGN NEW HIGH AND LOW SALES VA-  *
*  LUES. MAXSLS IS SET TO THE LARGER OF THE NEW INPUT VALUE *
*  OR THE CURRENT MAXIMUM. THIS GIVES US THE VALUE, BUT WE  *
*  DO NOT KNOW WHICH ONE IT WAS. ACCORDINGLY, THE TESTS ARE *
*  INCLUDED TO FIND THAT OUT.                               *
************************************************************


      DO WHILE  ((SLSNUM .GE. 1) .AND. (SLSNUM .LE. 9900))
         PRINT *, SLSNAM,SLSNUM,SALES
         TTLSLS = TTLSLS + SALES
         MAXSLS = MAX (MAXSLS,SALES)
         MINSLS = MIN (MINSLS,SALES)
         IF (MAXSLS .EQ. SALES) THEN
           MAXNUM = SLSNUM
           WINNER = SLSNAM
         ELSE IF (MINSLS .EQ. SALES) THEN
                MINNUM = SLSNUM
                LOSER = SLSNAM
              ELSE
         END IF
         PRINT *, 'ENTER NEXT SET FOR SLSNAM, SLSNUM, SALES'
         READ (*,END=99)  SLSNAM,SLSNUM,SALES
      END DO

   99 PRINT *, 'WINNER: ',WINNER,MAXNUM,MAXSLS
      PRINT *, 'LOSER: ',LOSER,MINNUM,MINSLS
      TTLSLS = ANINT (TTLSLS)
      PRINT *, 'TOTAL SALES THIS MONTH: ',TTLSLS
      STOP

      END
```

**FIGURE 6.5** FORTRAN Program for Example 6.2.

```
CAROLINE MENDELBLEB                42212     0.7540000E 04
FARNSWORTH BIGGLE                  54111     0.5435051E 04
FILLMORE FREEDLE                     228     0.1225000E 05
VARNA BAGLEY                        7723     0.1233250E 05
MARCIA PHNEPH                      44227     0.8755430E 04
SANFORD KIFINNE                      532     0.1877775E 05
TOOTHA DENTIPOD                    70077     0.1277010E 05
MARSHALL MARCIALE                  10127     0.2002250E 05
RIKKE RIKKEBAAKKE                  53340     0.5404250E 04
HAROLD SMITH                       33020     0.4220172E 04
WINNER:    MARSHALL MARCIALE                  10127     0.2002250E 05
LOSER:    HAROLD SMITH                       33020     0.4220172E 04
TOTAL SALES THIS MONTH:        0.1075080E 06
```

**FIGURE 6.6** Output for Example 6.2.

### 6.3.1 Algebraic Functions

There are four functions in this category, each requiring a single argument that may be either single or double precision. The result of the computation (i.e., the value returned by the function) will be the same type as the argument. As shown in Table 6.1, these functions include the square root, exponential (e to a power), and logarithms using two bases (10 and e).

**Table 6.1** Algebraic Built-in Functions

| *Function Name* | *Remarks* |
|---|---|
| SQRT | The argument must be zero or greater |
| LOG10 (common logarithm) | The argument must be zero or greater |
| LOG (natural logarithm) | The argument must be greater than zero |
| EXP (exponential) | The return value is $e^{arg}$ |

### 6.3.2 Trigonometric Functions

There are twelve functions in this category (Table 6.2). As is true with the algebraic functions, the arguments to these functions may be real (single precision) or double precision, with the result being the same data type as the argument. The functions require a single argument, with one form of the arctangent function (ATAN2) accepting two arguments. When ATAN2 is used, both arguments must be of the same type.

### 6.3.3 Random Number Generation

A variety of computer applications require data whose values appear to be governed by the laws of chance. For instance, we might have a program whose operation simulates the behavior of the traffic at a particular point on a busy highway. Over the long run, a known mixture of automobiles, trucks and other vehicles of various kinds cross that point. However, we cannot predict exactly what kind of vehicle will be the next one to come along at any given moment. Thus, every time we would want to simulate the appearance

**Table 6.2** Trigonometric Built-In Functions

| *Function Name* | *Remarks* |
|---|---|
| SIN | The argument is assumed to be in radians. Returned value is sine(arg). |
| COS | The argument is assumed to be in radians. Returned value is cosine(arg1). |
| TAN | The argument is assumed to be in radians. Returned value is tangent(arg). |
| ASIN | The argument must be in the range −1.0 to 1.0. Returned value (in radians) is arcsine(arg). |
| ACOS | The argument must be in the range −1.0 to 1.0. Returned value (in radians) is arccosine(arg). |
| ATAN, ATAN2 | If one argument is given, ATAN returns arctan(arg). If two arguments are given, i.e., ATAN(arg1, arg2), ATAN or ATAN2 returns arctan(arg1/arg2). |
| SINH | SINH returns the hyperbolic sine of arg. |
| COSH | COSH returns the hyperbolic cosine of arg. |
| TANH | TANH returns the hyperbolic tangent of arg. |
| ASINH | ASINH returns the hyperbolic arcsine of arg. |
| ACOSH | ACOSH returns the hyperbolic arccosine of arg. |
| ATANH | ATANH returns the hyperbolic arctangent of arg. |

of a vehicle, the directions to the simulating program would say, in effect, "Select a vehicle at random from the repertoire of available vehicle types, and send it across the point of interest at a certain velocity." Each vehicle would be represented (in the program) by a particular numerical value so that, in response to the request, the program would produce a *random number* within the desired range. Actually, generation of truly random numbers on a computer requires equipment whose cost and complexity is not justifiable in most instances. Instead, special algorithms are used to compute *pseudorandom numbers.* Collections of these numbers behave like collections of random numbers, and sequences of individually computed values appear to follow a random pattern. However, since pseudorandom numbers are computed systematically in accordance with the steps described by a particular algorithm, we can predict the value of each number in a sequence from that of its predecessor. For most purposes, this departure from true randomness is not a serious flaw.

HP 1000 FORTRAN 77 has three built-in functions for computing pseudorandom numbers. The three functions differ with regard to the mixture of numbers they produce over the long run. However, each produces a value on demand, based on the previous value produced. Accordingly, the functions must be supplied with a starting value called a *seed.* This is done for all of the functions via a special subprogram named SSEED. The initialization is done with a statement that looks like this:

CALL SSEED *(seed value)*

The parenthesized seed value is an integer. A logical place for this statement is near the beginning of a program, together with the rest of the initializations. If the programmer does not provide a seed value, the compiler uses 12345.

**6.3.3.1 The URAN Function** URAN produces a real number between 0.0 and 1.00. The underlying algorithm is set up so that a series of numbers generated by URAN will be uniformly distributed. For example, suppose we declared an integer variable RNUM and said

```
RNUM = INT (100.0*URAN())
```

(The empty set of parentheses tells the program that this is a call to a function without an accompanying list of arguments. If we were to use URAN in the above statement over and over say, 1000 times, we would get 1000 integers in the range 0–99 such that about 100 of them would be in the range 0–9, 100 of them would be in the range 10–19, and so on. In order for URAN to be used, it must be declared as a function in the program.

**6.3.3.2 The GRAN Function** Many natural populations can be described as Gaussian (normal) distributions. For example, if we were to weigh each of a large number of randomly selected navel oranges to the nearest ounce and record the number of occurrences for each weight, we would get a bell-shaped curve with its shape defined by a certain mathematical relationship. GRAN produces pseudorandom numbers that follow this distribution. Specifically, each number is a real value between −5.0 and +5.0, and a collection of such numbers will have a mean of 0.0 and a standard deviation of 1.0. A single number is produced by the expression

```
GRAN()
```

The empty parentheses have the same meaning as with URAN. As with GRAN, URAN must be declared as a function in the program in which it is to be used.

**6.3.3.3 The IRANP Function** Another distribution useful in many types of applications is the Poisson distribution. Values produced by HP 1000 FORTRAN 77's IRANP are integers that follow this distribution. Each value is obtained via the expression

```
IRANP (arg)
```

where arg is a positive real value (supplied by the programmer) less than 50.0.

Similar pseudorandom number generators are available in HP9000 FORTRAN 77. Consult the "HP FORTRAN/9000 Reference Manual," no. 5955–8845 for details.

### 6.3.4 Nested Invocation of Built-In Functions

Earlier in the chapter, it was mentioned that each built-in function is basically an independent program designed to perform certain operations on whatever data value it is given. Once the result has been produced and made available to the expression in which the function was used, it is as if the function had never been invoked. This means that we can use any or all of the functions when and where it suits our purpose with each usage being a separate activity. (We have already seen an example of two invocations of a function in the same expression back in Section 6.1.1.) This can be carried further by having invocations inside each other. This construction is called *nesting*. If V1, V2 and V3 are real variables, and values are available for V1 and V2, an assignment statement such as

```
V3 = LOG(SQRT(V1)+2.87*LOG(V2))
```

is perfectly legal and easily analyzed, since its evaluation follows the same rules governing all parentheses:

1. SQRT is invoked, returning the square root of V1 (call it T1).
2. LOG is invoked, returning the natural logarithm of V2 (call it T2).
3. T2 is multiplied by 2.87 (call the result T3).
4. T1 and T3 are added together (call the result T4).
5. The LOG function is invoked again, producing the natural logarithm of T4 (call it T5).
6. T5 is assigned to V3.

### 6.3.5 Generic Functions and The Good Old Days

In our examination of built-in functions, we have seen that each function has a permanently assigned name that is used to invoke that function. This is known as a *generic name*, the term implying that FORTRAN will look at the type of argument (or arguments) supplied to the function and make whatever adjustments are needed to deal with that data type. In earlier versions of FORTRAN, there were no generic names. Instead, each name used by the programmer to invoke a particular function depended on the type of argument being supplied to that function. For instance, if somebody wanted the square root of a single precision variable (e.g., `R1`), the expression would say `SQRT(R1)`. So far, so good. But, if the argument were a double precision variable (e.g., `D1`) the invocation would have to be `DSQRT(D1)`; `SQRT` would not work. FORTRAN 77 is designed to recognize and accept these multiple names so that programs written in earlier days can be handled by the new language. For this reason we include Table 6.3, which lists the names and their usage.

**Table 6.3** Generic functions and Their Data-Dependent Names

| *Generic Name* | *Argument Type* | *Specific Name(s)* |
|---|---|---|
| INT | Real (Real*4) | INT or IFIX |
| | Double Precision (Real*8) | IDINT |
| REAL | Integer*2, Integer*4 | FLOAT |
| | Real (Real*4) | SINGL |
| DBLE | Integer*2, Integer*4 | DFLOAT |
| | Real (Real*4) | DBLE |
| ABS | Integer*2, Integer*4 | IABS |
| | Real (Real*4) | ABS |
| | Double Precision (Real*8) | DABS |
| SIGN | Integer*2, Integer*4 | ISIGN |
| | Real (Real*4) | SIGN |
| | Double Precision (Real*8) | DSIGN |
| DIM | Integer*2, Integer*4 | IDIM |
| | Real (Real*4) | DIM |
| | Double Precision (Real*8) | DDIM |
| MOD | Integer*2, Integer*4 | MOD |
| | Real (Real*4) | AMOD |
| | Double Precision (Real*8) | DMOD |
| AINT | Real (Real*4) | NINT |
| | Double Precision (Real*8) | IDNINT |
| ANINT | Real (Real*4) | ANINT |
| | Double Precision (Real*8) | DNINT |
| MAX | Integer*2, Integer*4 | MAXO, AMAXO |
| | Real (Real*4) | AMAX1, MAX1 |
| | Double Precision (Real*8) | DMAX1 |
| MIN | Integer*2, Integer*4 | MINO, AMINO |
| | Real (Real*4) | AMIN1, MIN1 |
| | Double Precision (Real*8) | DMIN1 |
| SQRT | Real (Real*4) | SQRT |
| | Double Precision (Real*8) | DSQRT |
| LOG10 | Real (Real*4) | ALOG10 |
| | Double Precision (Real*8) | DLOG10 |
| LOG | Real (Real*4) | ALOG |
| | Double Precision (Real*8) | DLOG |

*(Continued)*

**Table 6.3** *(Continued)*

| *Generic Name* | *Argument Type* | *Specific Name(s)* |
|---|---|---|
| EXP | Real (Real*4) | EXP |
| | Double Precision (Real*8) | DEXP |
| SIN | Real (Real*4) | SIN |
| | Double Precision (Real*8) | DSIN |
| COS | Real (Real*4) | COS |
| | Double Precision (Real*8) | DCOS |
| TAN | Real (Real*4) | TAN |
| | Double Precision (Real*8) | DTAN |
| ASIN | Real (Real*4) | ASIN |
| | Double Precision (Real*8) | DASIN |
| ACOS | Real (Real*4) | ACOS |
| | Double Precision (Real*8) | DACOS |
| ATAN, ATAN2 | Real (Real*4) | ATAN, ATAN2 |
| | Double Precision (Real*8) | DATAN, DATAN2 |
| SINH | Real (Real*4) | SINH |
| | Double Precision (Real*8) | DSINH |
| COSH | Real (Real*4) | COSH |
| | Double Precision (Real*8) | DCOSH |
| TANH | Real (Real*4) | TANH |
| | Double Precision (Real*8) | DTANH |
| ASINH | Real (Real*4) | ASINH |
| | Double Precision (Real*8) | DASINH |
| ACOSH | Real (Real*4) | ACOSH |
| | Double Precision (Real*8) | DACOSH |
| ATANH | Real (Real*4) | ATANH |
| | Double Precision (Real*8) | DATANH |
| NOT | Integer*2, Integer*4 | NOT |
| IAND | Integer*2, Integer*4 | IAND |
| IOR | Integer*2, Integer*4 | IOR |
| IXOR, IEOR | Integer*2, Integer*4 | IEOR |
| IBSET | Integer*2, Integer*4 | IBSET |
| IBCLR | Integer*2, Integer*4 | IBCLR |
| IBITS | Integer*2, Integer*4 | IBITS |
| MVBITS | Integer*2, Integer*4 | MVBITS |
| ISHFT | Integer*2, Integer*4 | ISHFT |
| ISHFTC | Integer*2, Integer*4 | ISHFTC |
| BTEST | Integer*2, Integer*4 | BTEST |

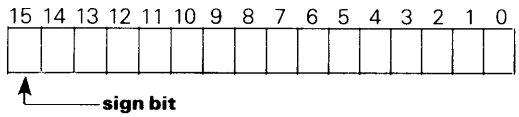
## 6.4 LOGICAL ARITHMETIC WITH INTEGER DATA

In addition to FORTRAN 77's standard arithmetic capabilities, HP FORTRAN 77 provides a powerful facility for handling integer values as strings of individual bits. This is done by allowing FORTRAN's logical operators to be applied to integer values.

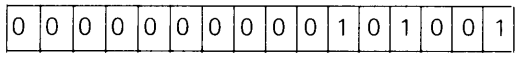
### 6.4.1 Internal Representation of Integer Values for the HP1000

Unlike the case with standard arithmetic, exploitation of HP FORTRAN 77's versatile logical facility requires a detailed understanding of the way integer data are stored in HP1000 systems. As stated earlier, an integer value occupies one word in storage. The

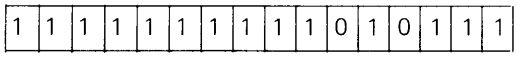
bits of that word are assigned as follows:



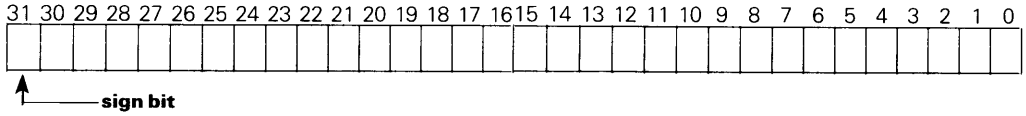

Positive values have a sign bit of zero and the other fifteen bits give the absolute value. Thus, the integer value +41 looks like this:



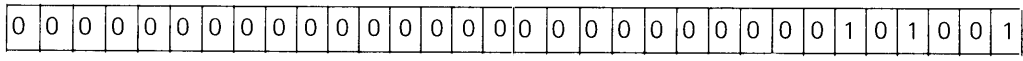

Negative values have a sign bit of 1, and the other fifteen bits give the 2's complement of the absolute value. The 2's complement of a binary number is obtained by changing every 1 to a 0 and vice versa, and then adding 1. Thus, the integer value −41 looks like this:



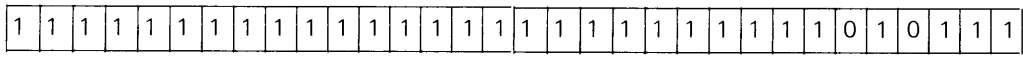

32-bit integer values, occupying two HP1000 words or one HP9000 word, use the 32 bits as follows:




The value +41, stored as a 32-bit integer, has the following form:




Similarly, the value −41, when stored as a 32-bit integer, looks like this:




### 6.4.2 Logical Operations on Integer Values

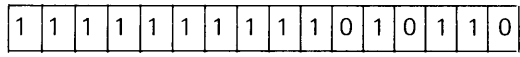
HP FORTRAN 77 supports five logical operations. When applied to integer values, they process each bit independently, producing a result of 1 or 0 for each operation. We shall

illustrate these for 16-bit integers (i.e., regular integers on the HP1000 or short integers on the HP9000). However, the same rules apply for 32-bit integers as well.

**6.4.2.1 The .NOT. Operation** The . NOT. operation (both periods are needed to specify the operator) works on a single integer value, changing each 0 to a 1 and each 1 to a 0. For instance, suppose VCOUNT is declared as a 16-bit integer variable and it has a value of 41. (We saw what that looks like in the previous section.) If we were to write

```
VCOUNT = .NOT. VCOUNT
```

each of VCOUNT's 16 bits would be changed (*inverted*) and the resulting value stored back into VCOUNT would be




which corresponds to a value of −42. The same process applies to 32-bit integer values.

**6.4.2.2 The .AND. Operation** The . AND. operation works on corresponding bits from two integer values to produce a 1 or 0 according to the following rules:

1. 1 . AND. 1 → 1
2. 1 . AND. 0 → 0
3. 0 . AND. 1 → 0
4. 0 . AND. 0 → 0

Thus, if integer variable VCOUNT has a value of 41, and NEWV is another integer variable, the statement

```
NEWV = VCOUNT . AND. 25
```

will place a value of 9 in NEWV. Let us see how that happens:

```
          0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1     VCOUNT (41)
. AND.    0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1        25
yields    0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1         9 (NEWV)
```

**6.4.2.3 The .OR. Operation (inclusive or)** The . OR. operation is the "mirror image" of the . AND. operation:

1. 1 . OR. 1 → 1
2. 1 . OR. 0 → 1
3. 0 . OR. 1 → 1
4. 0 . OR. 0 → 0

If we use the same values as before, changing only the operation, i.e.,

```
NEWV = VCOUNT . OR. 25
```

the value stored in NEWV will be 57:

```
          0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1     VCOUNT (41)
. OR.     0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1        25
yields    0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1        57 (NEWV)
```

**6.4.2.4 The .EQV. Operation (equivalence)** The .EQV. operation produces a 1 only when both operands are the same:

1. 1 .EQV. 1 → 1
2. 1 .EQV. 0 → 0
3. 0 .EQV. 1 → 0
4. 0 .EQV. 0 → 1

Using the same values as before, the statement

```
NEWV = VCOUNT .EQV. 25
```

places a value of −49 in NEWV:

```
          0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1     VCOUNT (41)
.AND.     0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1       25
yields    1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1      -49 (NEWV)
```

**6.4.2.5 The .NEQV. Operation (exclusive or)**

1. 1 .NEQV. 1 → 0
2. 1 .NEQV. 0 → 1
3. 0 .NEQV. 1 → 1
4. 0 .NEQV. 0 → 0

Referring to our example one more time, the statement

```
NEWV = VCOUNT .NEQV. 25
```

produces a value of 48, which then is stored in NEWV:

```
          0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1     VCOUNT (41)
.NEQV.    0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1       25
yields    0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0       48 (NEWV)
```

### 6.4.3 Bit-Handling with Logical Operations

The application of logical operations to integer data makes it possible to manipulate individual bits conveniently. There is virtually no end to the techniques that can be built based on these opportunities. Consequently, we must limit ourselves to an examination of some fundamental approaches that can serve as guidelines for more ambitious extensions.

**6.4.3.1 Bit-Setting Operations** If we think of an integer as a string of bits in which each bit represents an individual switch that may be on (1) or off (0), it would be useful to be able to manipulate these switches independently of each other.

One such manipulation involves placing designated bits in an integer while leaving the others unaffected. (Placing a 1 in a certain position is called *setting*; placing a 0 there is called *clearing*.) To illustrate, suppose we wanted to set bits 3 and 4 of integer variable VCOUNT. (Recall that the leftmost bit is bit 15, the next one is bit 14, and so on.) A simple way to set a bit is to set up a *mask*—an integer in which all the bits are 0 except for those of interest (bits 3 and 4 in this case). this is easily done using HP FORTRAN 77's hexadecimal form for integer constants. If we write out the desired mask bit by bit, we get

```
0 0 0 0    0 0 0 0    0 0 0 1    1 0 0 0
```

Replacing each group of four bits with its hexadecimal equivalent gives us Z'0018', so that the declaration of our mask would say

```
INTEGER*2   MASK_34
DATA        MASK_34/Z'0018'/
```

Then, if we wanted to set bits 3 and 4 of VCOUNT, the simple assignment

```
VCOUNT = VCOUNT .OR. MASK_34
```

would do it.

Clearing particular bits follows a similar pattern. Suppose we wanted to clear bits 2 and 11 of VCOUNT without affecting the others. This time, our mask would consist entirely of 1s except for bits 2 and 11:

```
1 1 1 1    0 1 1 1    1 1 1 1    1 0 1 1
```

Accordingly, our declaration would say

```
INTEGER*2        MASK2_11
DATA             MASK2_11/Z'F7FB'/
```

and the clearing operation would be achieved by the assignment

```
VCOUNT = VCOUNT .AND. MASK2_11
```

(Examination of the .AND. operation's properties makes it clear that when the mask bit is 0, the result inevitably is 0; when the mask bit is 1, the result is unchanged from its original value.)

**6.4.3.2 Bit-Testing Operations** A useful companion to bit-setting and bit-clearing is the ability to test individual bit values so that they can motivate subsequent decisions. One simple way to do this is to copy the value of interest into another location, clear all the bits except those to be tested, and then test against the mask that was used to clear the irrelevant bits.

To illustrate, suppose we wanted to base a particular decision on the presence of 1s in bit 7 and 13 of integer variable VCOUNT. We shall use TST_VCOUNT as the variable on which the test is performed. (The original value still will be preserved in VCOUNT.) The appropriate mask, prepared in variable MASK7_13, looks like this:

```
0 0 1 0    0 0 0 0    1 0 0 0    0 0 0 0
```

This corresponds to a hexadecimal value of Z'2080'. Consequently, our sequence will be as follows:

```
INTEGER*2      VCOUNT, TST_VCOUNT, MASK7_13
DATA           MASK7_13/Z'2080'/
               . . . . . .
TST_VCOUNT =   VCOUNT .AND. MASK7_13
               . . . . . .
IF (TST_VCOUNT .EQ. MASK7_13) THEN
                  etc.
```

As a result, bit positions 7 and 13 in TST_VCOUNT will preserve the corresponding values from VCOUNT and all the other bit positions will be cleared. Then, since bits 7 and 13 of the mask were set by the declaration, an equal comparison with TST_VCOUNT will mean that those bits are set in VCOUNT as well.

## 6.5 FUNCTIONS FOR BIT MANIPULATION

In addition to the fundamental tools described in Section 6.4, HP FORTRAN 77 provides a set of built-in functions (and a subroutine) that offer an alternative way of handling individual bits in integer data. Many of these functions are included in the military extension to FORTRAN 77 (MIL-STD-1753). There is no clearcut set of criteria that identifies the use of these functions as being universally superior to the use of masking and other logical techniques outlined earlier. Consequently, it is helpful for the programmer to be aware of both sets of possibilities so the appropriate choice can be made for each situation.

There are eleven bit-related subprograms. These are categorized and described in the following sections.

### 6.5.1 Basic Logical Functions

HP FORTRAN 77 includes functions for four logical operations:

1. The NOT function is identical to the .NOT. operator:

   ```
   VCOUNT = NOT (VCOUNT)
   ```

   and

   ```
   VCOUNT = .NOT. VCOUNT
   ```

   produce the same result.

2. The IAND function duplicates the .AND. operator:

   ```
   NEWV = IAND (VCOUNT, 25)
   ```

   and

   ```
   NEWV = VCOUNT .AND. 25
   ```

   produce the same result.

3. The IOR function duplicates the .OR. operator:

   ```
   NEWV = IOR (VCOUNT, 25)
   ```

   and

   ```
   NEWV = VCOUNT .OR. 25
   ```

   produce the same result.

4. The IXOR function (also named IEOR) is identical to the .NEQV. operator:

   ```
   NEWV = IXOR (VCOUNT, 25)
   ```

   and

   ```
   NEWV = VCOUNT .NEQV. 25
   ```

   produce the same result.

There is no function for equivalence, but its effect is produced easily enough:

```
NEWV = VCOUNT .EQV. 25
```

can be duplicated by

```
NEWV = NOT (IXOR(VCOUNT, 25))
```

or

```
NEWV = .NOT. (IEOR(VCOUNT, 25))
```

or even

```
NEWV = .NOT. (VCOUNT .NEQV. 25)
```

### 6.5.2 Bit-Setting Functions

Two functions make it possible to set or clear a particular bit in an integer value:

1. IBSET (*str, p*) sets the bit in position *p* of integer value *str*. The other positions are unaffected. If *str* is a 16-bit integer, *p* must be 0–15; if *str* is a 32-bit integer, *p* must

be 0–31. Otherwise, the function has no effect. Either or both arguments may be integer expressions.

2. IBCLR (*str,p*) clear position *p* in integer *str*. The same stipulations described for IBSET are true here as well.

Thus, assuming VCNT1, VCNT2 and NEWV are integer variables with values in them, the statement

```
NEWV = IBSET (VCNT1, 12) + IBCLR (VCNT2, 5)
```

sets bit 12 in a temporary copy of VCNT1, clears bit 5 in a temporary copy of VCNT2, adds these two values together, and stores the result in NEWV. Note that VCNT1 and VCNT2 retain their original values. If we had wanted VCNT1 and VCNT2 changed, we would have to say

```
VCNT1 = IBSET (VCNT1, 12)
VCNT2 = IBCLR (VCNT2, 5)
NEWV = VCNT1 + VCNT2
```

### 6.5.3 Bit-Manipulating Subprograms

Four subprograms provide facilities for isolating bits and moving them around.

**6.5.3.1 The IBITS Function** This function makes it possible to isolate portions of an integer so that these portions can be treated as separate values. For instance, suppose we have the following declarations:

```
INTEGER*2      OLDV, NEWV
DATA           OLDV/Z'F632'/
```

Then, the statement

```
NEWV = IBITS (OLDV, 4, 5)
```

produces the following activities:

1. The five bits starting in position 4 of OLDV and going to the left are copied into the rightmost positions of a temporary location (let us call it T1). In other words, bits 4, 5, 6, 7, and 8 of OLDV are places in positions 0, 1, 2, 3, and 4, respectively, of T1.
2. The remaining positions of T1 are cleared.
3. T1 is copied into NEWV.

For the values in our example, this means that NEWV will end up with a value of Z'0003'. Thus, the first argument specifies the value from which the bits are to be extracted, the second indicates the starting position (the rightmost position) of the portion to be extracted, and the third argument specifies the length of the desired extraction. The latter argument must be greater than zero, and it must be consistent with the second argument. For instance, we cannot get a 7-bit string starting in position 12 of a 16-bit integer. Any or all of the arguments may be integer expressions.

**6.5.3.2 The MVBITS Subroutine** This subprogram replaces part of one integer value with a string of bits extracted from another integer value. The general form for its usage consists of a complete statement appearing as follows:

CALL MVBITS (*str1, p1, size, str2, p2*)

The third argument (*size*) specifies the number of bits to be transferred. These come from integer *str1* starting in position *p1* and are copied into integer *str2* starting in position

*p2*. To illustrate, the statement

```
CALL MVBITS (OLDV1, 8, 5, OLDV2, 3)
```

copies bits 8, 9, 10, 11, and 12 from integer OLDV1 into respective positions 3, 4, 5, 6, and 7 of integer OLDV2. Any or all of the five arguments may be integer expressions.

**6.5.3.3 Bit-Shifting Functions** ISHFT (logical shift) and ISHFTC (circular shift) enable the programmer to change an integer value by changing the positions of bits within the value. To illustrate, suppose we say

```
INTEGER*2      OLDV, NEWV
DATA           OLDV/Z'F632'/
. . . . . .
NEWV = ISHFT (OLDV, -3)
```

As a result, the bits in OLDV will be shifted three positions to the *right*. The rightmost three bits fall off the end into a large plastic bit bucket which is emptied periodically by persons unknown. The newly vacated positions at the left are cleared. Therefore, the value set to NEWV will be Z'1EC6', i.e.,

```
1 1 1 1  0 1 1 0  0 0 1 1  0 0 1 0
```

shifted 3 positions right yields

```
0 0 0 1  1 1 1 0  1 1 0 0  0 1 1 0
```

Had we said

```
NEWV = ISHFT (OLDV, 3)
```

the shift would go in the other direction (to the *left*). Accordingly, the leftmost bits fall into the bit bucket and the rightmost ones are cleared. Using the same original value for OLDV, the value now stored in NEWV will be Z'B190', i.e.,

```
1 1 1 1  0 1 1 0  0 0 1 1  0 0 1 0
```

shifted 3 positions right yields

```
0 0 0 1  1 1 1 0  1 1 0 0  0 1 1 0
```

Note that a right shift followed by a left shift of the same length will not necessarily bring back the original value. (Troublesome thing, that bit bucket.)

The ISHFTC function performs a *circular shift*. That is, it treats the value as if it were a closed ring. Consequently, bits do not fall off, and the bit bucket is denied its booty. Instead, bits shifted out of one end are shifted into the other end. As a result, only the positions change. The general form is

```
ISHFTC (str, npl, size)
```

where *size* is the number of bits to be shifted and *npl* indicates the number of places over which the shift is to occur. ISHFTC always shifts the rightmost bits. As an example, let us use the declared value for OLDV with the statement

```
NEWV = ISHFTC (OLDV, -4)
```

As a result, the rightmost four bits of OLDV become the leftmost four bits of NEWV and everything else is pushed along accordingly. Thus, NEWV will receive a value of Z'2F63'. Had we said

```
NEWV = ISHFTC (OLDV, 4)
```

the rightmost four bits of OLDV would be shifted circularly to the left, so that the leftmost four bits of OLDV would become the rightmost four bits of NEWV. That is, NEWV's value

will be Z'632F'. The absolute value of *npl* must be no greater than size. Here again, any or all of the arguments may be integer expressions.

### 6.5.4 Bit-Testing with the BTEST Function

An individual bit in an integer value may be tested with the BTEST function whose general form is

BTEST (*str, p*)

If position *p* of integer *str* has a 1, the outcome is .TRUE. ; a *p* value greater than 15 (or greater than 31 if *str* is a double integer) automatically produces a .FALSE. outcome.

## 6.6 SUMMARY

FORTRAN includes a library of independent programs designed to perform certain computational tasks and make it appear as if these tasks were single operations. These programs, called *built-in functions*, are available to any and all FORTRAN programs. A built-in function is used by referring to its name and supplying it with data on which to operate. This referral is known as *invocation* and the data supplied to a function is called an *argument*. A function may require one or more arguments, depending on the type of operations it is designed to perform. When a function completes its task, it *returns* a single result which then is treated like any other value in an expression.

## PROBLEMS

1. Show the result of each of the following computations. Both the value and the data type are to be considered:

(a) INT(26.97)
(b) 32.16 + 2*INT(0.4741)
(c) INT(3+2.479) + 4*INT(0.217D+03)
(d) INT(8.3+INT(2.89))
(e) 3*REAL(41)
(f) REAL(4.71+0.538406275811D+01)
(g) REAL(INT(4.84/2.2))
(h) REAL(INT(REAL(22.84.28.84)))
(i) DBLE(48/12+6.3)
(j) DBLE(REAL(INT(7.2/3.6)))

2. Show the result of each of the following computations. Both the value and the data type are to be considered:

(a) AINT(3.62+8.7)
(b) AINT(8.81+INT(4.58))
(c) REAL(5/2+AINT(11.0/4.0))
(d) DBLE(17/4 + AINT(0.3486D+03))
(e) AINT(REAL(17/4)+AINT(40.0+INT(14.4)))
(f) AINT(-7.3)+AINT(-7.7)
(g) ABS(-2*7.2)
(h) ABS(AINT(-8.81))
(i) REAL(ABS(-3+INT(6.6*2**3)))

3. Show the result of each of the following computations. Both the value and the data type are to be considered:

(a) ANINT(3.62+7.45)
(b) ANINT(0.54600E2)
(c) ANINT(REAL(8/5)+INT(3.6*1.1)+ANINT(3.88))
(d) NINT(3.68)
(e) NINT(3.68+ANINT(3.68))
(f) NINT(REAL(61/8)+ANINT(4.81)+AINT(5.58))
(g) SIGN(3.81,2.7)
(h) SIGN(-7.7,6.1)
(i) SIGN(32.8,AINT(-8.1))
(j) DIM(14.1,7.1)
(k) DIM(81,104)*AINT(49.72/3.12)
(l) NINT(MAX(21,18)/3*SIGN(DIM(14.8,6.8),-4.4)+AINT(6.6))

4. Write FORTRAN expressions for each of the following:

(a) $2.7 + 3\sqrt{X + Y}$

(b) $\dfrac{\sqrt{X + Y}}{\sqrt{X - Y}}$

(c) $\log 3X + Y$

(d) $\log (3X + Y)^2$

(e) $2e^{-0.1X} \sin 0.7X$

(f) $X^{-\log \sin Y}$

(g) $X^{\cos Y} - \sqrt{\log_{10} \sin X^2}$

(h) $\dfrac{Y^{-\sin^2 X}}{1 + \log\left(\dfrac{X}{Y}\right)}$

(i) $\sin^2 (e^{-\cos 2X})$

(j) $\left(\dfrac{X}{\sqrt{\log Y}} - \dfrac{Y}{\sqrt{2 \log X^2}}\right)^{[\log Z]}$

(k) $\sin \log X^{e^Y}$

(l) $\log \dfrac{\text{Max}\,(e^{\sin X}, |\sqrt{3Y} - 41^x|)}{\sin^{-1} Z}$

(m) $\dfrac{(X^{\sqrt{\pi}} - e^{\sqrt{X+Y}}) \tan^2 \log Z}{1 + \dfrac{\log_{10}\sqrt{X}}{3\pi Z}}$

5. Many mathematical functions can be computed as series in which the accuracy of the function value depends on the number of terms used in the computation. For example, we can get a value of e (the base of natural logarithms) from FORTRAN by using the EXP function with an argument of `1.0`. This value (which turns out to be `0.2718281E+01`) can be computed from the series

$$e = 2 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} \cdots + \frac{1}{n!}$$

We can examine the effect of each additional term by evaluating the series with 1 term, 2 terms, 3 terms, etc., and comparing each of the results with the reference value. Thus, for e, this comparison would look like this:

| *no. of terms* | *series value* | *ser val = ref val* |
|---|---|---|
| 1 | 0.1000000E01 | −0.1718281E+01 |
| 2 | 0.2000000E01 | −0.7182810E+00 |
| 3 | 0.2500000E01 | −0.2182810E+00 |
| 4 | 0.2666667E01 | −0.5161400E−01 |
| 5 | 0.2708334E01 | −0.9947400E−02 |
| 6 | 0.2716667E01 | −0.1613700E−02 |
| 7 | 0.2718056E01 | −0.2252000E−03 |
| 8 | 0.2718230E01 | −0.5140000E−04 |
| | etc. | |

We can see that by the eighth term, the computed value agrees with the reference value to four decimal places.

Prepare such a table for each of the following functions. Stop at ten terms. Obtain the reference value by invoking the appropriate FORTRAN built-in function. Prepare a structured flowchart or pseudocode representation of your design. (NOTE: Errors in roundoff and truncation can be minimized in such computations by adding the terms together from smallest to largest):

(a) $\sin X = X - \dfrac{X^3}{3!} + \dfrac{X^5}{5!} - \dfrac{X^7}{7!} + \dfrac{X^9}{9!} - \cdots$ (use X = 0.5)

(b) $\cos X = 1 + \dfrac{X^2}{2!} + \dfrac{X^4}{4!} + \dfrac{X^6}{6!} + \dfrac{X^8}{8!} + \cdots$ (use X = 0.5)

(c) $\log X = 2\left[\dfrac{X-1}{X+1} + \dfrac{1}{3}\left(\dfrac{X-1}{X+1}\right)^3 + \dfrac{1}{5}\left(\dfrac{X-!}{X+1}\right)^5 + \cdots\right]$ (use X = 2.0)

(d) $\tan^{-1} X = X - \dfrac{1X^3}{3} + \dfrac{1X^5}{5} - \dfrac{1X^7}{7} + \cdots$ (use X = 0.5)

(e) $e^x = 1 + X + \dfrac{X^2}{2!} + \dfrac{X^3}{3!} + \dfrac{X^4}{4!} + \cdots$ (use X = 2.5)

6. Repeat any or all of the sections of Problem 5 so that the computations are halted when the difference between the series value and the reference value is less than 0.001. In any event, do not let the computations go beyond ten terms.

7. Write FORTRAN expressions for each of the following computations:

   (a) Round the value 2.7 to the nearest integer.
   (b) Round the value 81.99 to the nearest tenth.
   (c) Round the value 786.972 to the nearest thousand.
   (d) Round the value 786.972 to three decimal places.
   (e) Round the real value in variable FMAX to the nearest hundredth.
   (f) Bring the real value in variable GMIN to the nearest whole number.
   (g) Round the largest of the three real variables X, Y and Z to four decimal places.
   (h) Round the sum of the largest and smallest of the values in real variables G1, G2, G3, G4 and G5 to the nearest thousandth.

8. Write a statement or sequence of statements to perform each of the following processing activities:

   (a) Set bit 7 of integer variable VMAX.
   (b) Set bits 0 and 15 of integer variable VMAX.
   (c) Clear bits 12–14 and set bits 21–25 of double integer variable SBIG.
   (d) Clear every fourth bit (starting with position 0) of integer FROLL.
   (e) Duplicate the bit values in positions 5–11 of integer BSOURCE in the same respective positions of integer BDEST.
   (f) Duplicate the bit values in positions 11–14 of integer BFONT in positions 12–15 of integer BREC.
   (g) Duplicate the bit values in positions 8–10 of integer BVAL in positions 1–3 of BVAL.
   (h) Copy the bit value in position 9 of double integer DBVAL into positions 22 through 27 of DBVAL.
   (i) Exchange the upper and lower halves of integer FOLDEM.
   (j) Reverse the order of the bits in double integer MIRROR.
   (k) Clear integer MNSW except for bit positions 7, 10, and 13.
   (l) Set all the bits in integer POTSW except for positions 8, 11, and 12.
   (m) If bit 10 of integer GSW is 1, set GSW's even-numbered bit positions; otherwise, clear bit 10 and set bit 4.
   (n) If at least four bits in integer SWTST have 1 in them, clear integer STARTB, set all the bits in integer HOLDB, and store the sum of STARTB and HOLDB in NEWSUM; otherwise, set the lower half (position 0–7) of HOLDB, clear the upper half of STARTB, and store the sum of HLDB and STARTB in NEWSUM.

9. Write a program that reads a succession of three-digit positive integers, storing each one, in turn, in INTG. The program is to keep track of the number of values read in (NUMVAL) and the number of values (NUMSYM) in which the first and third digits are the same. Each time such a value is read, it is to be printed. At the end of the run (you must determine how that is recognized), the program prints NUMVAL and NUMSYM. Thus, a value of 484, or 101, or 777 fits the description, while a value of 228, 487, or 822 does not.

10. Nowhere in the world is there a more spectacular St. Patrick's Day parade than in Brooklyn. Tens of thousands have thrilled to its floats, bands, green goats and baton twirlers, but the main attraction (as long as anyone can remember) has been the thousands of marchers moving crisply in precise, equal rows. All nationalities, all ages, all colors, all sizes, and (on that day) all Irish. For 34 years this organizational marvel has been in the competent hands of Paddy Gogarty, and this year Paddy is about to make it all work again. But there is a sadness in his eyes: His good friend O'Casey has joined the Great Parade and forevermore will he be missing from his accustomed position at the end of the last row.

    Anyway, there is Gogarty watching the rows form on the vast Prospect Park Parade Grounds and trying to ignore his longtime aide Flynn:

    "O'Casey's ghost will march in this parade, sure as anything!"

    Everything was ready for the run-through and Gogarty watched as the rows of six marched by, signalling adjustments and corrections. When the last row arrived, Paddy's eyes widened: One marcher was missing from the end of the row.

    "O'Casey!" whispered Flynn.

    "Shaddup!!" shouted Gogarty. "March them in fives."

    The marchers regrouped and marched again. Gogarty looked, nodded, then blinked. There, in the last row, was an empty position.

    "Fours!" said Gogarty. Flynn said nothing.

    Back they came, four in a row. The last row had three. Gogarty ordered threes; the last row had two. O'Casey's face flashed in Paddy's head but he grimly asked for rows of two and said some other things.

(No need repeating them here; they have little to do with FORTRAN.) By now, he was not surprised to see a lone marcher in the last row.

Almost at that instant, Paddy felt a tug at his sleeve, causing him to jump 2.816 feet straight up. But it was only little Kevin McFardle, bringing help. Help was in the form of a FORTRAN program that Kevin had written. Knowing how many marchers there were (MRCHRS), Kevin was able to devise an algorithm that would tell him what size row was needed to make all the rows equal.

"Use sevens" said Kevin.

A desperate Paddy tried it and, sure enough, as the last row came by, a beaming Gogarty counted seven marchers. Rest in peace, O'Casey.

Knowing what happened with six, five, four, etc., knowing that seven works, and knowing that Paddy has at least 500 marchers in his ranks, write a program that computes and prints the first three values of MRCHRS for which this situation works.

11. The distance between two points (X1,Y1) and (X2,Y2), when plotted on rectangular coordinates, can be computed by the formula

$$d_{12} = \sqrt{(X2 - X1)^2 + (Y2 - Y1)^2}$$

Write a program that reads and processes sets of four points (X1,Y1), (X2,Y2), (X3,Y3) and (X4,Y4). The program is to compute and print the distances between all pairs of points. Use an (X1,Y1) of (−100,−100) to stop the run. Display each distance on a separate line and leave a blank line between sets.

12. This program is to meet the same requirements described for the previous problem, with one difference: The distances are to be printed along with the names of the two pertinent points.

13. Write a program that reads two pair of integer values for each input set: The time now in hours and minutes (HRSNOW and MNSNOW) and the elapsed time (HRSLPS and MNSLPS). After printing each pair of values on a separate line, the program is to print the new time. HRSNOW is in military hours (noon = 12, 1:00 PM = 13, etc., midnight = 0) and your answer should be in military time (on a third line, appropriately labeled). For example, if HRSNOW and MNSNOW are 8 and 45, respectively (i.e., 8:45) and HRSLPS and MNSLPS are 5 and 37, respectively (i.e., a time lapse of 5 hours and 37 minutes), the new time will be 2:22 PM, or 14 (for hours) and 22 (for minutes). Similarly, if MRSNOW and MNSNOW are 21 and 14, respectively, a time lapse of 7 hours and 49 minutes will give a new time of 5:03 AM (i.e., an hour value of 5, minute value of 3). Use an HRSNOW value of −1 to stop the run.

14. When forces are exerted on an object in a geometric plane, it is helpful to examine these forces abstractly, in a diagram such as the one shown in Figure 6.7. Here we see three forces F1, F2 and F3 being exerted at three angles A1, A2, and A3. Each of the forces has a horizontal component equal to the force times the

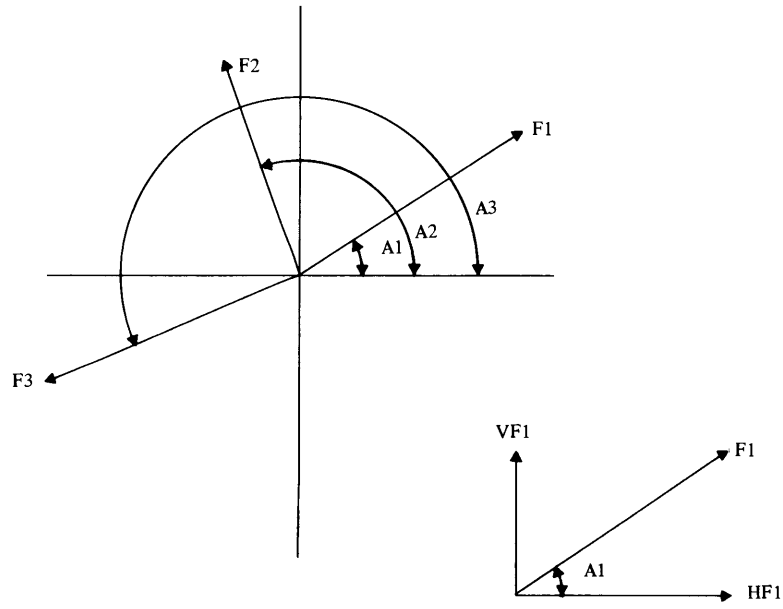



**FIGURE 6.7** Horizontal and Vertical Force Components Acting on an Object.

cosine of the angle it makes with the positive X-axis, and a vertical component equal to the force times the sine of the angle it makes with the positive X-axis. In terms of Figure 6.7, F1's horizontal component, for example, is

HF1 = F1 cos A1

If we add all the horizontal components together, we can determine the total (net) force exerted parallel to the X-axis. In the case of the three forces shown in Figure 6.7, this total horizontal force would be

HF = HF1 + HF2 + HF3 = F1 cos A1 + F2 cos A2 + F3 + cos A3

The same type of value can be computed for the vertical components:

VF = VF1 + VF2 + VF3 = F1 sin A1 + F2 sin A2 + F3 sin A3

The resultant force, then, is the total force exerted by the combination and it can be computed as

$$R = \sqrt{VF^2 + HF^2}$$

The angle that this resultant force makes with the positive X-axis can be computed from its tangent:

$$\text{Tan AR} = \frac{VF}{HF}$$

Write a program in which each input set consists of six values: F1, A1, F2, A2, F3, and A3. All values are real and the angles are given in degrees. (Remember that arguments to the trigonometric functions must be in radians. There are 3.14159 radians in 180 degrees.) Output for each set is to consist of an echo of the input (each force and its angle on a separate line) followed by a fourth line showing the resultant force and its angle, in degrees. A blank line is to separate output for each set and an F1 value of 0.0 stops the run. Operate your program with the following input sets:

| *F1* | *A1* | *F2* | *A2* | *F3* | *A3* |
|---|---|---|---|---|---|
| 100.0 | 30.0 | 100.0 | 45.0 | 100.0 | 60.0 |
| 100.0 | 30.0 | 100.0 | 150.0 | 100.0 | 90.0 |
| 10.0 | 45.0 | 250.0 | 90.0 | 400.0 | 180.0 |
| 25.0 | 0.0 | 50.0 | 195.0 | 200.0 | 310.0 |

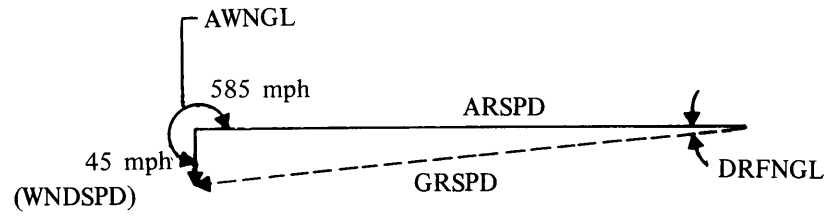



**FIGURE 6.8** Forces Acting on an Aircraft in Flight.

15. When an airplane sets out to fly in a certain direction, that direction may be affected by a prevailing wind. For example, an airplane traveling due west at 585 miles per hour might encounter a wind blowing due south at 45 miles per hour (Figure 6.8). This means that after, say, an hour of flying at these conditions (without correcting for them) the plane would have traveled 585 miles west and 45 miles south. In order to correct for this effect, it is necessary to determine the ground speed of the aircraft (GRSPD) and the drift angle (DRFNGL). Thus, if ARSPD is the airspeed, WNDSPD is the windspeed and AWNGL is the angle between the aircraft heading and the wind direction (90 degrees in the example of Figure 6.8), we can use the law of cosines to solve for GRSPD:

The Law of Cosines is $\cos A = \frac{b^2 + c^2 - a^2}{2bc}$

and then the law of sines will help us solve for DRFNGL:

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$$

Write a program in which an input set consists of ARSPD, WNDSPD and AWNGL (the latter in degrees) and the output is to echo the input (one line for each value, with each one labeled) followed by GRSPD and DRFNGL (on one line). DRFNGL is to be printed in degrees and a blank line is to separate the input sets. Use an ARSPD of zero to stop the run and operate your program with the following values:

| *ARSPD* | *WNDSPD* | *AWNGL* |
|---|---|---|
| 325.0 | 0.0 | 0.0 |
| 280.0 | 10.0 | 0.0 |
| 300.0 | 20.0 | 180.0 |
| 300.0 | 55.0 | 90.0 |
| 400.0 | 50.0 | 120.0 |

16. The tramp steamer was a pretty tacky vessel to begin with, so that even in calm seas it had its troubles. Thus, it was not surprising that it went under during the terrific storm of '93 (known for decades thereafter as The Terrific Storm of '93). When things settled down, what was left was a trio of survivors (two rogues and an assistant rogue) washed up on a small island. Hours later, they awoke and began to look around. All they could find was a grove of coconut trees. An afternoon's hard work produced a respectable pile of coconuts and a medium sized monkey who observed the proceedings. Exhausted by their labors, the group decided to go to sleep, planning to divide the coconuts when they awoke. Yes indeed.

    Survivor A awoke a little later, bothered by this sudden outbreak of honesty. Action quickly followed thought as she divided the pile into three equal groups. This left a single coconut which she gave to the monkey. Then, she buried one of three equal piles, made a single pile of the remaining nuts and went back to sleep.

    Survivor B awoke soon after that and did the same thing. After dividing the pile into three equal piles, there was one coconut left over, and that went to the ever watchful monkey. One of the piles was buried, the other two recombined, and back to sleep went B. Watch for it.

    Yup; here it comes: C gets up, looks at the pile, imagines it was bigger the last time he saw it, but is not sure. Anyway, the pile is split into three piles, the monkey gets the single coconut that is left over, one of the piles is buried, the other two recombined, and that is that.

    Comes the dawn and the group wakes up, ready to split the harvest. By now, you can imagine that the shrinkage is noticeable, but nobody says anything. They go to work and split the pile (mini-pile?) into three equal piles. This leaves one coconut, and the monkey gets that. What happens afterward is not of immediate interest here. What is of interest is the following: Write a program that computes and prints the minimum number of coconuts for which this story would be true. If you design and set up a reasonable algorithm, it will be easy to find other numbers for which this works once you find the minimum, so find the next three numbers and print those.

# 7
# *Arrays*

It is often convenient to be able to treat a number of similar data items as an organized collection. Such groupings, known in FORTRAN as *arrays*, make it possible to store collections of integers, real numbers, characters, or logical values under a common name. A wide variety of operations then can be applied either to the entire array or to any individual items in it.

## 7.1 ORGANIZATION OF ARRAYS

An array consists of a group of *elements* where each element is the same type as the others. The size of an array (i.e., the number of elements) is defined by the programmer with a declaration. For example, suppose that we are designing a program dealing with six different types of paintbrushes and we need to store the six prices so that any or all of them are available on demand. This is handled easily by treating the prices as an array. Using BRUSH_PRICE as the array's name, we can construct a mental picture of this collection (Figure 7.1). Note that the name BRUSH_PRICE refers to the entire collection. However, when we attach a number to the general name, that number (called a *subscript*), enables us to identify a single element (e.g., the second element, representing the price of the second paintbrush type) and process it separately.

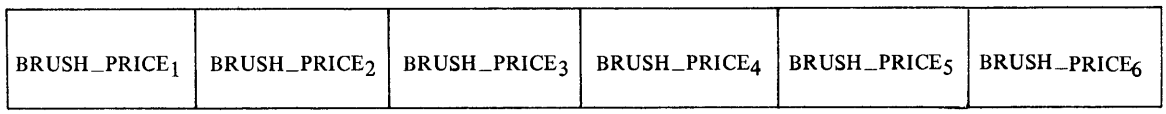



**FIGURE 7.1** Organization of BRSHPR as a One-Dimensional Array.

Although the details of array declaration are discussed later in this chapter, it will be helpful to introduce the basic form at this point. For the array of brush prices just discussed, we can write

```
REAL BRUSH_PRICE(6)
```

and FORTRAN will reserve storage for six real values under the collective name BRUSH_PRICE. Each of the elements then can be identified separately by using a subscript along with the array name. By specifying BRUSH_PRICE in the above declaration, we are telling FORTRAN that the six elements are to be named BRUSH_PRICE(1), BRUSH_PRICE(2), BRUSH_PRICE(3), BRUSH_PRICE(4), BRUSH_PRICE(5), and BRUSH_PRICE(6).

### 7.1.1 Dimensionality of Arrays

The array named BRUSH_PRICE is an example of a *one-dimensional* array. This means that there is just one factor (the brush type) that distinguishes each brush price from the

other five. We can find the price of, say, a type 4 brush by adding a single piece of information, i.e., the (4) to the collective name BRUSH_PRICE. In certain circumstances this type of classification is not complete enough. We may want to identify an element in an array by two separate categories. For instance, it may not be adequate to think of our six different paintbrushes as being simply six different types. Instead, suppose these are made in three different sizes, and each size comes in two different grades. Now, it takes two pieces of information, i.e., the size and grade, to pinpoint which of the six types of brushes we mean. If we want to store the six prices to reflect this organization of the data, we would set up a *two-dimensional* array. One way to declare it is as follows:

```
REAL BRUSH_PRICE (3, 2)
```

This still tells FORTRAN to reserve room for six real numbers, but now we have added the requirement that these six elements be organized in three *rows* and two *columns*, as represented in Figure 7.2. Each row would be used to hold prices for the two brushes having the same size, and each column is designed to hold prices for the three brushes having the same grade. Consequently, we must specify a size (row) and grade (column) in order to get to an individual element. For example, the assignment statement

```
BRUSH_PRICE (2, 1) = 4. 15
```

assigns the value 4. 15 to the element in row 2, column 1. In our array organization, this element represents a second size, first grade brush.

Using exactly the same requirements (three sizes, two grades), we could represent the same data with a slightly different organization: The declaration

```
REAL BRUSH_PRICE (2, 3)
```

reserves room for six elements as before, but this time they are organized as two rows and three columns (Figure 7.3), with each row representing a size. Now, if we wanted to assign

| | Column 1 | Column 2 |
|---|---|---|
| Row 1 | BRUSH_PRICE (1,1) | BRUSH_PRICE (1,2) |
| Row 2 | BRUSH_PRICE (2,1) | BRUSH_PRICE (2,2) |
| Row 3 | BRUSH_PRICE 3,1) | BRUSH_PRICE (3,2) |

**FIGURE 7.2** Organization of BRSHPR as a Two-Dimensional Array (3 × 2).

| | Column 1 | Column 2 | Column 3 |
|---|---|---|---|
| Row 1 | BRUSH_PRICE (1,1) | BRUSH_PRICE (1,2) | BRUSH_PRICE (1,3) |
| Row 2 | BRUSH_PRICE (2,1) | BRUSH_PRICE (2,2) | BRUSH_PRICE (2,3) |

**FIGURE 7.3** Organization of BRSHPR as a Three-Dimensional Array (2 × 3).

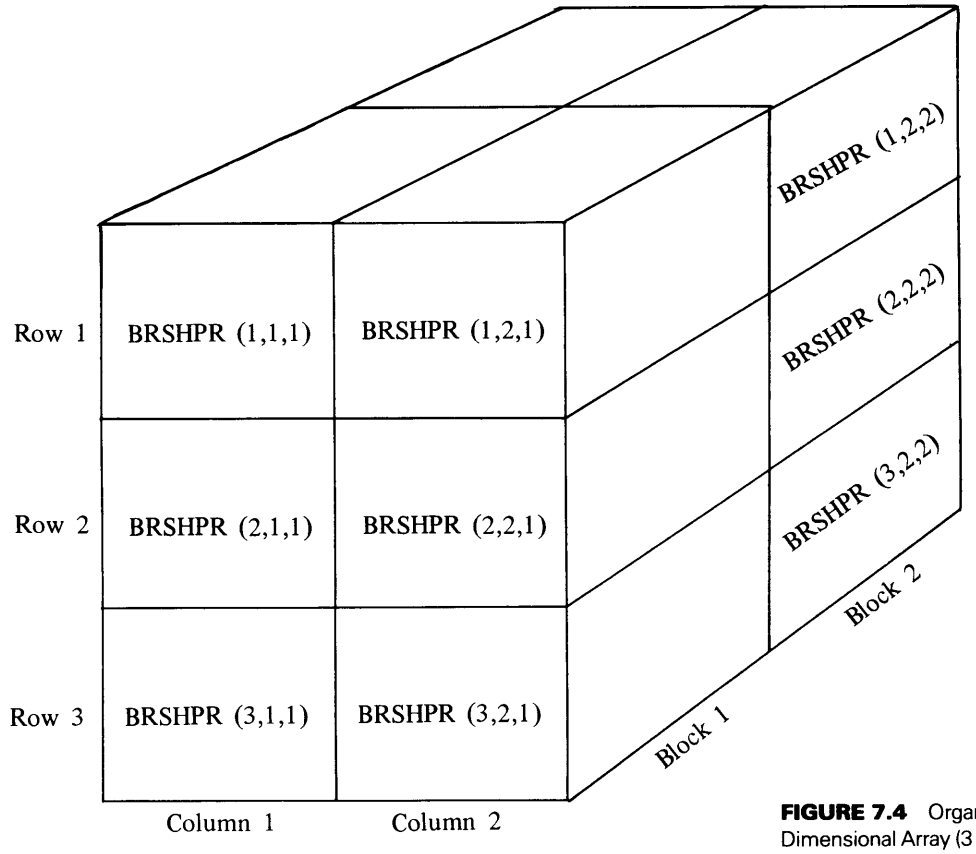



**FIGURE 7.4** Organization of BRSHPR as a Three-Dimensional Array (3 × 2 × 2).

a price of 4.15 to the size 2 grade 1 brush, we would say

```
BRUSH_PRICE(1,2) = 4.15
```

The organization in Figure 7.3 is not "better" or "worse," "milder" or "harsher" than the one in Figure 7.2; it is only different. The choice depends on the programmer's view of what will make the program simpler and clearer.

Thus, the *dimensionality* of an array indicates the number of pieces of information required to find one of its elements. The number of subscripts matches the dimensionality. Note also that the declaration indicates how many elements there are in the array. For example, our declaration of BRUSH_PRICE with 3 rows and 2 columns reserved room for (3 × 2) elements. An array X declared as

```
INTEGER X(10,15)
```

would have 10 × 15 or 150 elements.

More intricate situations may call for arrays with greater dimensionality. For instance, suppose there is great pressure from the marketplace so that now there are twelve different paintbrushes: Three sizes, two grades, and (by popular demand) two bristle lengths. This could be represented by a *three-dimensional* array:

```
REAL BRUSH_PRICE (3,2,2)
```

The organization of the array is shown in Figure 7.4. Note, as before, that the rows, columns, and *blocks* (as the third dimension is called) were selected as a matter of preference.

HP FORTRAN enables the programmer to declare arrays having as many as 32768 dimensions on the HP1000 and (2**32)−1 on the HP9000. (After rows, columns, and

blocks, many people refer to the fourth dimension as slices; beyond that, it is up to your imagination.)

### 7.1.2 Representation of Arrays in Storage

We have seen that we can find any array element by specifying the right subscript(s). The mechanism that does the searching and finding is automatic and, in many cases, we can let it operate without paying attention to it. However, there are certain situations where it is necessary to know the order in which the elements are stored. One type of instance stems from the fact that we can read in an entire array with a single statement. For example, if we write

```
REAL*4 WT (24)
READ *, WT
```

FORTRAN will go and read 24 values (consistent with WT's declaration as a 24-element array). Since WT is one-dimensional, there is no problem in analyzing what happens. The first input value is placed in WT(1), the second one in WT(2), and so on, until 24 values have been read and stored. But now let us look at the following statements:

```
REAL*4 WT2 (6, 4)
READ *, WT2
```

What happens here? We still have an array of 24 elements (6 rows × 4 columns) and the READ statement still drags in 24 values. But where are they stored? "Who cares?" is not the appropriate answer. The right input values will get to the right element only by the luckiest coincidence unless we know how FORTRAN handles this, and we arrange the input values using that knowledge.

To find out, let us start with WT2, declared above as a 6 × 4 array. We can think of that array's organization as the one shown in Figure 7.5. The way FORTRAN stores these elements is shown in Figure 7.6. Note that the elements are stored in order by column.

| | | | |
|---|---|---|---|
| WT2 (1,1) | WT2 (1,2) | WT2 (1,3) | WT2 (1,4) |
| WT2 (2,1) | WT2 (2,2) | WT2 (2,3) | WT2 (2,4) |
| WT2 (3,4) | WT2 (3,2) | WT2 (3,3) | WT2 (3,4) |
| WT2 (4,1) | WT2 (4,2) | WT2 (4,3) | WT2 (4,4) |
| WT2 (5,1) | WT2 (5,2) | WT2 (5,3) | WT2 (5,4) |
| WT2 (6,1) | WT2 (6,2) | WT2 (6,3) | WT2 (6,4) |

**FIGURE 7.5** Representation of the 6 × 4 Array WT2.

| WT2 (1,1) | WT2 (2,1) | WT2 (3,1) | WT2 (4,1) | WT2 (5,1) | WT2 (6,1) | WT2 (1,2) | . . . . . | WT2 (5,4) | WT2 (6,4) |
|---|---|---|---|---|---|---|---|---|---|

**FIGURE 7.6** Sequence of Storage for the 6 × 4 Array WT2.

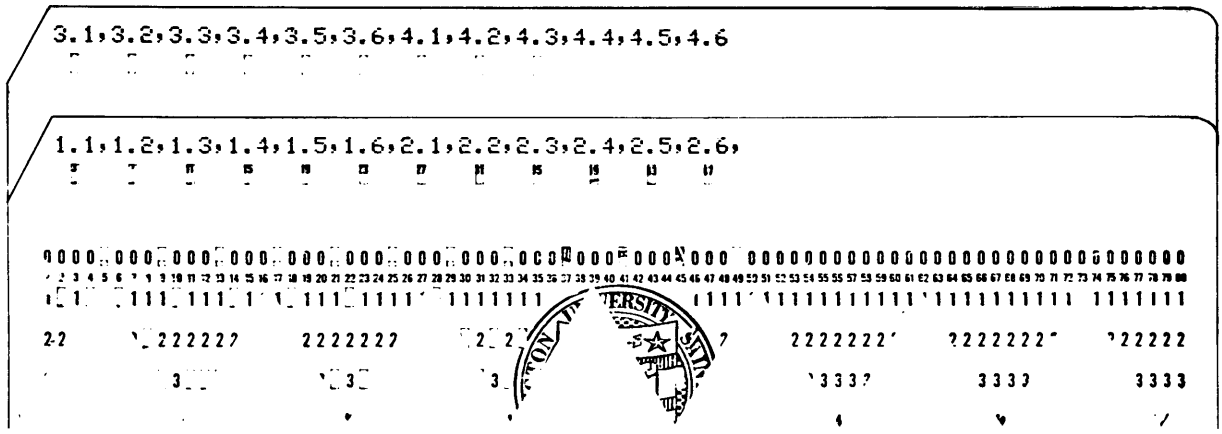



**FIGURE 7.7** Twenty-four Data Values (see Figures 7.5, 7.6, and 7.8) as They are Punched into a Data Card.

| | Col. 1 | Col. 2 | Col. 3 | Col. 4 |
|---|---|---|---|---|
| Row 1 | 1.1 | 2.1 | 3.1 | 4.1 |
| Row 2 | 1.2 | 2.2 | 3.2 | 4.2 |
| Row 3 | 1.3 | 2.3 | 3.3 | 4.3 |
| Row 4 | 1.4 | 2.4 | 3.4 | 4.4 |
| Row 5 | 1.5 | 2.5 | 3.5 | 4.5 |
| Row 6 | 1.6 | 2.6 | 3.6 | 4.6 |

**FIGURE 7.8** Stored Values in WT2 after Reading Data of Figure 7.7.

Going back to the READ statement for WT2, the first input value is stored in WT2(1,1) as expected. The next one is stored in WT2(2,1), the third one in WT2(3,1), and so on. Nothing is placed in the second column until FORTRAN gets to the seventh input value because it took the first six to fill the elements in the first column.

To make sure this is clear, we shall apply the previous READ statement to the 24 data values shown on the input lines in Figure 7.7. When that statement is executed using these values, the result will be as pictured organizationally in Figure 7.8.

### 7.1.3 A General Formula for Locating Array Elements

The order in which FORTRAN assigns storage for array elements follows a rule that can be described as follows:

> Array elements are stored so that the leftmost subscript varies first. The next one does not change until the leftmost subscript has gone through its entire range. This sequence proceeds from left to right, with the rightmost subscript changing least frequently.

We shall see how this works on a three-dimensional array declared as follows:

```
INTEGER*2 N(4,2,3)
```

The 24 elements (4 rows × 2 columns × 3 blocks) will be stored in the sequence N(1,1,1), N(2,1,1), N(3,1,1), N(4,1,1), N(1,2,1), N(2,2,1), etc. Note that the row subscript (the leftmost one) went through the complete range (from 1 to 4) before the next subscript moved from 1 to 2. Thus, the element N(1,2,1) is the fifth element in the sequence. In other words, the fifth input value brought in by a READ statement would be stored in N(1,2,1). If we look a little further in the sequence, .....N(2,2,1), N(3,2,1), N(4,2,1), N(1,1,2), N(2,1,2), N(3,1,2), ....., we see that the column subscript (second from left) went through its range (1–2) before the block subscript (the rightmost one) changed. Thus, in a three-dimensional array, we can think of the subscripts as the hands of a clock, with the leftmost subscript acting like a second hand, the middle one like a minute hand, and the right one like an hour hand.

This rule can be restated as a formula for computing the position of an element in a FORTRAN array. For a two-dimensional array, let

R = number of rows in the array,
C = number of columns in the array,
row = row number of the element whose position we want,
col = column number of the element whose position we want,
P(row,col) = position of an element with the subscript (row, col).

Then, the position of an element is

$$P(row,col) = row + R * (col-1)$$

Note that C, the number of columns in the array, does not enter into the calculations. For a three-dimensional array, we add a third dimension which we shall call B (the number of blocks), and a third subscript which we shall call blk. Thus, we are looking for P(row,col,blk), the position of the element with the subscript (row,col,blk):

$$P(row,col,blk) = row + R * (col-1) + R * C * (blk-1)$$

Note again that the rightmost dimension (B) is not used in the formula.

**Example 7.1 (a)** Find the position of element HT(7,4) in an array declared with the statement

```
REAL*4 HT(8,6)
```

Using our formula,

```
P(7,4) = 7 + 8 * (4-1)
 = 31
```

Thus, HT(7,4) is the 31st element in the 8 × 6 array HT.

**Example 7.1 (b)** Find the position of element PRS(3,5,4) in an array declared with the statement

```
INTEGER*2 PRS(5,7,7)
```

Using our formula,

```
P(3,5,4) = 3 + 5 * (5 - 1) + 5 * 7 * (4-1)
= 3 + 20 + 105
= 128
```

## 7.2 SPECIFICATION OF NUMERICAL ARRAYS

We have already seen that the declaration of an array is very similar to that for a single-valued variable. In this section we shall look at some additional features for array declarations, learn some more about subscripts, and take another brief journey to the past.

### 7.2.1 Declaration of Array Bounds

The statements we have been using for declaring arrays let FORTRAN make a standard assumption about how the element's subscripts will be specified. For example,

```
REAL AMT(15)
```

reserves room for a 15-element array of real values. Moreover, it implies that the first element's name is AMT(1), the second is AMT(2), and so on. There are occasions when it is more convenient to start with a subscript other than 1. FORTRAN 77 makes this possible by accepting a form of array declaration in which the programmer explicitly specifies the subscripts for the first and last elements in any or all of the array's dimensions. To illustrate, suppose we wanted to set up a one-dimensional real array TTL of 15 elements such that the first element will be named TTL(0). Since TTL is to have 15 elements, this means that the last (15th) element in the first (and only) dimension will have a subscript value of 14 (not 15, because we start from 0 rather than 1). The appropriate declaration would look like this:

```
REAL*4 TTL(0:14)
```

As a result, FORTRAN will set up room for 15 elements (14−0+1) named TTL(0), TTL(1), TTL(2), . . . . , TTL(14). Accordingly, the statement

```
READ *, TTL
```

would read 15 input values, with the first one being stored in TTL(0), etc. Now that we are acquainted with this form, it should be clear that the earlier declaration of AMT (in which we let FORTRAN use its standard assumption) represents the same thing as

```
REAL*4 ANT(1:15)
```

There is no restriction on the numbers one may use in this type of declaration as long as the one to the right of the colon is larger than the one to the left. Table 7.1 shows some examples.

Declaration of each dimension does not affect any of the others. For instance, the statement

```
INTEGER*2 PBX(10,5), SETNO(4,-1:10), CRNO(0:9,-2:2)
```

**Table 7.1** Array Declarations

| *Declaration* | *No. of Elements* | *Subscripts* | |
|---|---|---|---|
| INTEGER Y(107) | 107 | | 1 Through 107 |
| REAL X(0:10) | 11 | | 0 Through 10 |
| REAL WIDTH (−1:4) | 6 | | −1 Through 4 |
| CHARACTER LTRS *6(3,7) | 21 | Row: | 1 Through 3 |
| | | Column: | 1 Through 7 |
| INTEGER N2(15,0:3) | 60 | Row: | 1 Through 15 |
| | | Column: | 0 Through 3 |
| REAL GR(−6:6,10) | 130 | Row: | −6 Through 6 |
| | | Column: | 1 Through 10 |
| INTEGER HY(0:12,−4:0) | 65 | Row: | 0 Through 12 |
| | | Column: | −4 Through 0 |
| INTEGER WLS(6,4:0) | | Illegal—Improper column dimension | |
| REAL PTR(−5:5,4,0:6) | 308 | Row: | −5 Through 5 |
| | | Column: | 1 Through 4 |
| | | Block: | 0 Through 6 |
| CHARACTER WR *5(6,4:) | | Illegal—Improper column dimension | |
| REAL T(0,5,4) | | Illegal—Improper row dimension | |
| REAL ST(0:2,−1:1,5) | 45 | Row: | 0 Through 2 |
| | | Column: | −1 Through1 |
| | | Block: | 1 Through 5 |

reserves room for three integer arrays with the following characteristics:

1. A 10 × 5 array (PBX) in which the rows are numbered 1–10 and the columns are numbered 1–5.
2. A 4 × 12 array (SETNO) in which the rows are numbered 1–4 and the twelve columns are numbered −1, 0, 1, 2, . . . , 10.
3. A 10 × 5 array (CRNO) in which the rows are numbered 0, 1, 2, . . . , 9 and the columns are numbered −2, −1, −, 1, 2.

### 7.2.2 Initialization of Arrays

The DATA statement, introduced in Chapter 4 as a convenient way to initialize single-valued variables, serves equally well for arrays. As an example, the statements

```
REAL*4      TABL(10)
DATA      TABL/0.,1.,2.,3.,4.,5.,6.,7.,8.,9./
```

define an array and set its ten elements to values of 0.0 through 9.0. This is a concise way of building tables of constant values in a program where those values will be consulted again and again.

The same process can be applied to the initialization of arrays having more than one dimension. Here again, (as is true with input/output) it is necessary to know FORTRAN's sequence for storing array elements to make sure that the assignments are made as we want them. To review this sequence, let us assume the following declaration:

```
REAL*4 SPGRAV(3,2)
```

Now, convince yourself that the statement

```
DATA SPGRAV/0.682,0.691,0.708,0.719,0.732,0.771/
```

produces the same results as the sequence

```
SPGRAV(1,1) = 0.682
SPGRAV(2,1) = 0.691
SPGRAV(3,1) = 0.708
SPGRAV(1,2) = 0.719
SPGRAV(2,2) = 0.732
SPGRAV(3,2) = 0.771
```

**7.2.2.1 Using Repetition Factors to Initialize Arrays** When the elements of an array are to be initialized to a common value, we can use the DATA statement's repetition factor as was done in Chapter 4. Thus, the statements

```
REAL*4 SUMS(12)
DATA SUMS/12*0.0/
```

reserve storage for a 12-element array and set its elements to initial values of 0.0.

Multidimensional arrays can be initialized with the same technique. To illustrate, the statements

```
REAL*8 RAD(0:8,-5:5)
DATA RAD/99*0.0D0/
```

initialize RAD's 99 elements. Note that the repetition factor (99 in this example) must agree with the array size (9 rows × 11 columns).

**7.2.2.2 Initializing Parts of Arrays to Different Values** Rather than setting all of an array's element to the same value, or setting each individual element to a different value, there are situations calling for initialization to be done in groups. This can be handled quite easily in FORTRAN by adding a form of DO loop to the DATA statement. Such a loop is called an *implied* DO *loop*, primarily because the form in which it is specified is somewhat less complete than that for a more generalized loop (as discussed in Chapter 3). The basic form for the implied loop can be defined conveniently by looking at an example:

```
INTEGER*2 CT(20),I
DATA (CT(I), I=1,8)/8*0/(CT(I), I=9,20)/8*1/
```

we initialize the first eight elements to zero and the other twelve to 1. The variable I, called an *implied*-DO-*variable*, may be used more than once as shown. Its usage is basically the same as that of an index variable in a regular DO loop. Specifically, the assignments produced by the DATA statement are the same as those resulting from a sequence such as

```
DO I = 1, 8
     CT(I) = 0
END DO
DO I = 9, 20
     CT(I) = 1
END DO
```

### 7.2.3 Specification of Subscripts

Much of the power in using arrays stems from the fact that we can work with any element by referring to its subscript. This power is made even more flexible by allowing such references to be made in two basic ways: Either we know what an element's subscript is, or we know how to compute that subscript. To support this flexibility, FORTRAN accepts a variety of subscript expressions. For example, let us analyze the following group

of statements:

```
REAL*4 PRICE (20)
INTEGER*2 I
PRINT *, 'ENTER A VALUE FOR PRICE'
READ *, PRICE
   . . . . . .
   . . . . . .
   .
other statements
   . . . . . .
   . . . . . .
PRICE(20) = 1.08 * PRICE(20)
DO I=1,9
     PRICE(I) = 1.05 * PRICE(I)
     PRICE(I+10) = 1.06 * PRICE(I)
END DO
```

After declaring PRICE as a 20-element real array, we read 20 numbers and store them, respectively, in PRICE(1), PRICE(2), and so on. The single element PRICE(20) is adjusted by replacing its value with one that is 8% higher. Then, there is a loop that repeats exactly nine times. Note that each time through the loop, we affect the values of two elements: PRICE(I) is increased by 5%, and PRICE(I+10) is increased by 6%. In one case, the index variable I is used to indicate which element to adjust; in the other, the variable I is part of a subscript expression whose resulting value specifies the appropriate element. For instance, when I is 2 (the second time through the loop), PRICE(2)'s value will be increased by 5%, and the value in PRICE(2+10) or PRICE(12) will be increased by 6%. Consequently, when the loop completes its nine cycles, PRICE(1) through PRICE(9) will have new values, and PRICE(11) through PRICE(19) will have new values. PRICE(20) will have been changed before entering the loop, and PRICE(10) will remain unchanged.

In general, HP FORTRAN 77 allows any integer or real expression to be used for specifying a subscript. FORTRAN will evaluate the expression using its normal rules (as described in Section 5.2). If the expression produces a real value, HP FORTRAN 77 will truncate it. The resulting number will be used to help locate the desired element. For example, if PRICE is declared as shown before, and K1 and K2 are 8 and 3, the assignment statement

```
PRICE (4 * K1/K2 - 3) = 17.6
```

places the value 17.6 in PRICE(7).

#### 7.2.4 The DIMENSION Statement

FORTRAN 77 accepts the DIMENSION statement as another way of declaring arrays. This is a holdover from earlier versions of the language that can be used as follows:

```
DIMENSION name (specifications)
```

The specifications give the extent of each dimension, as they do in a regular declaration. For instance,

```
DIMENSION A2(10), INK(5, 7)
```

declares a one-dimensional real array A2 of 10 elements and a two-dimensional integer array INK consisting of 5 rows and 7 columns. Note that the automatic naming conventions (see 4.2.2.6) are in force here. (A2 is real and INK is integer because of their

respective first letters.) There is no opportunity to specify the data type, nor can we use the array bounds described in Section 7.2.1. Although the DIMENSION statement is an easy way to earmark array declarations, its limitations discourage its continued use. We mention it so that it will not appear strange when it is encountered.

## 7.3 PROCESSING OF ARRAYS

A major reason for organizing groups of data into arrays is that we expect to process those values systematically, performing the same kinds of operations on each one. FORTRAN makes it very easy to set up such computations. By combining the power of the DO loop with the capability to work with any element independent of the others in its array, we can deal with selected elements, parts of an array, or an entire array with equal convenience.

### 7.3.1 Array Elements in Expressions

The fact that we have used array elements in expressions and assignment statements without particular concern (as we did in the previous section) emphasizes the idea that when we refer to an array element, it is no different from an ordinary single-valued variable: It may be assigned a value, compared to some other value, read in, or printed. Moreover, an element may appear in an expression anywhere a constant or single-valued variable may appear.

**Example 7.2** To illustrate the fundamental aspects of array processing we shall design and implement a program to perform various operations on a group of arrays. The requirements are specified in the program to exemplify the use of comments for descriptive information about the program. (Such information is called *documentation.*) This practice places the statements and documentation in one inseparable place, adding to the program's clarity and ease of use. A structured flowchart for this program is shown in Figure 7.9(a) and a pseudocode representation is given in Figure 7.9(b). The corresponding program is given in Figure 7.10. Figure 7.11 shows a sample run.

In this version of the program, we have constructed a loop to handle each of the required computations. While this will get the job done, it does not take full advantage of the fact that once a loop is set up, the repeating mechanism can be used to control any number of activities which are to be done over and over.

The looping mechanism is exploited more fully in a second version of the program, where all of the processing is consolidated within a single loop that cycles 20 times. Each activity is regulated inside the loop by means of its own test. For instance, an IF-THEN-ELSE construct is included to determine how each of D's elements is computed. E's elements still are calculated in a separate loop because, for any element in E, we need an element of C that has not yet been computed. Consequently, the program waits till all of C's elements are available, and then it produces E. (By being more clever, we can arrange to express E's subscripts so that these representations can indeed be included in the same general loop. However, it is usually better to stay with the direct, straightforward approach.)

The structured flowchart and pseudocode for this second version are shown in Figure 7.12 and the body of the program is given in Figure 7.13. Note that the output is not affected at all by the change in loop organization.

### 7.3.2 Input/Output of Arrays

As the last example demonstrated, FORTRAN keeps enough information about declared arrays so that we can read or print an entire array with a simple statement. These bookkeeping services also make it possible to perform input/output operations on parts of arrays with only a little more work.

Suppose we have declared a one-dimensional integer array MRGN consisting of 12 elements and we wanted to read MRGN(3) through MRGN(8). One obvious way to handle this would be to list the elements:

```
READ *, MRGN(3), MRGN(4), MRGN(5), MRGN(6), MRGN(7), MRGN(8)
```

```
"Define SMDSQ, SUM1, SUM2."
"Define 20-element arrays A, B, C, D, and E."
"Set SMDSQ, SUM1 and SUM2 to zero."
"Read and echo arrays A and B."
do for all I, I = 1 to 20:
    "Compute C(I) = 2 × A(I) + B(I)."
enddo
do for the first eight elements:
    "Compute D(I) =

enddo
do for the last 12 elements:
    "Compute D(I) =

enddo
do for the first 5 elements:
    "Compute E(I) = A(I+5) + B(I+7) + C(2I+1)."
enddo
do for all 20 elements:
    "Add D(I)**2 to SMDSQ."
enddo
do for the last 10 elements:
    "Add A(I)*B(I) to SUM1."
enddo
do for odd-numbered values of I from 1-20:
    "Add A(I)*B(I) to SUM2."
enddo
 "Print C,D,E,SMDSQ,SUM1,SUM2."
 "Stop."
```

**FIGURE 7.9 (b)** Pseudocode for Example 7.2.

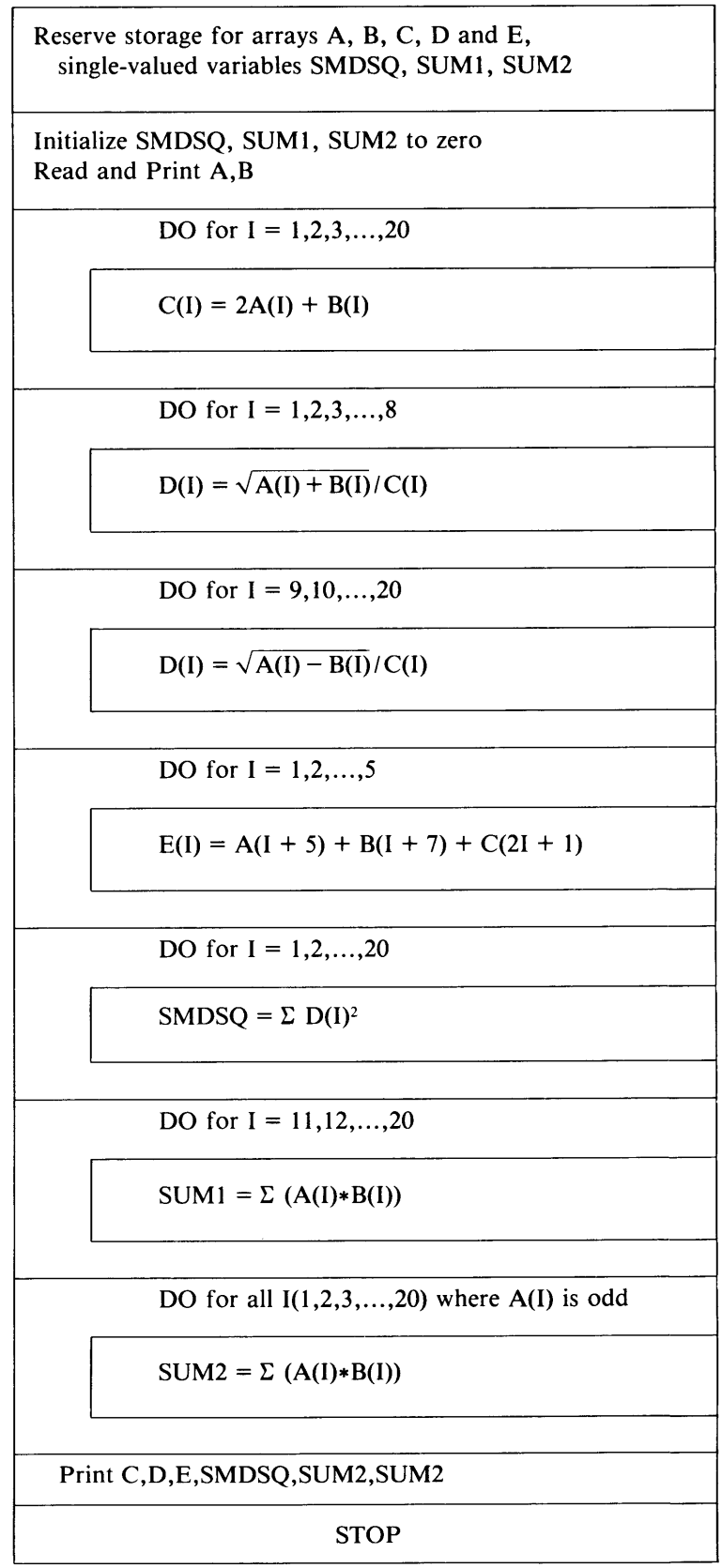



**FIGURE 7.9 (a)** Structured Flowchart for Example 7.2.

```
************************************************************
*                        EXAMPLE 7.2                       *
************************************************************
* THIS PROGRAM PERFORMS A VARIETY OF MANIPULATIONS TO      *
* ILLUSTRATE ARRAY PROCESSING:                             *
* A, B, C, AND D ARE 20-ELEMENT REAL ARRAYS;               *
* E IS A 5-ELEMENT REAL ARRAY;                             *
* AFTER READING THE VALUES FOR A AND B, THE FOLLOWING COM- *
* PUTATIONS ARE PERFORMED:                                 *
*   (1) ARRAY C IS DEVELOPED SUCH THAT EACH ELEMENT OF C   *
*       IS THE SUM OF THE CORRESPONDING ELEMENT IN B AND   *
*       TWICE THE CORRESPONDING ELEMENT IN A.              *
*    (2) ARRAY D IS DEVELOPED SO THAT EACH OF ITS FIRST    *
*        EIGHT ELEMENTS REPRESENTS THE SQUARE ROOT OF THE  *
*        SUM OF THE CORRESPONDING ELEMENTS IN A AND B, WITH*
*        THAT SUM DIVIDED BY THE CORRESPONDING ELEMENT IN  *
*        C. SPECIFICALLY,                                  *
*            FOR I <= 8:  D(I)=SQRT(A(I)+B(I))/*(I)        **
*        THEN,                                             *
*           FOR I > 8: D(I)=SQRT(A(I)-B(I))/C(I)           *
*    (3) THE ELEMENTS OF ARRAY E ARE COMPUTED AS FOLLOWS:  *
*                E(1) = A(6) + B(8) + C(3)                 *
*                E(2) = A(7) + B(9) + C(5)                 *
*                E(3) = A(8) + B(10)+ C(7)                 *
*                E(4) = A(9) + B(11) + C(9)                *
*                E(5) = A(10) + B(12) + C(11)              *
*     (4) A VALUE SMDSQ IS COMPUTED CONSISTING OF THE SUM  *
*         OF THE SQUARED VALUES IN ARRAY D                 *
*     (5) A VALUE SUM1 IS COMPUTED CONSISTING OF THE SUM   *
*         OF THE PRODUCTS OF THE LAST TEN PAIRS OF CORRES- *
*         PONDING A AND B ELEMENTS                         *
*     (6) A VALUE SUM2 IS COMPUTED CONSISTING OF THE SUM   *
*         OF THE PRODUCTS OF ALL CORRESPONDING PAIRS OF    *
*         A AND B ELEMENTS IN WHICH A'S VALUE IS ODD       *
*  THE PROGRAM IS SET UP SO THAT MESSAGES ASKING FOR INPUT *
*  ARE DISPLAYED ON THE USER'S TERMINAL, BUT THE COMPUTED  *
*  RESULTS ARE SENT TO THE STANDARD SYSTEM UNIT TO BE      *
*  DISPLAYED THERE. THE PRINT STATEMENT IS USED FOR THAT   *
*  PURPOSE.                                                *
************************************************************

      PROGRAM   EX702
      IMPLICIT  NONE
      REAL*4    A(20),B(20),C(20),D(20),E(5),SMDSQ,SUM1,SUM2
      INTEGER*2 I

      SMDSQ = 0.0
      SUM1 = 0.0
      SUM2 = 0.0
      PRINT *, 'ENTER VALUES FOR ARRAY A, FOLLOWED BY ARRAY B'
      READ *, A,B
      PRINT *, 'ARRAY A:'
      PRINT *, A
      PRINT *, 'ARRAY B:'
      PRINT *, B
      PRINT *, '       '

************************************************************
*               COMPUTE THE VALUES FOR ARRAY C             *
************************************************************
      DO I=1,20
         C(I) = 2.0*A(I) + B(I)
      END DO

************************************************************
* COMPUTE D'S FIRST 8 ELEMENTS: D(I)=SQRT(A(I)+B(I))/C(I)  *
* THEN ANOTHER LOOP FOR THE LAST 12: D(I)=SQRT(A(I)-B(I))/C(I)*
************************************************************
```

**FIGURE 7.10** FORTRAN Statements for Example 7.2. *(continued)*

```
      DO I=1,8
         D(I) = SQRT(A(I)+B(I))/C(I)
      END DO
      DO I=9,20
         D(I) = SQRT(A(I)-B(I)/C(I)
      END DO

*****************************************************************
*  FROM THE DESCRIPTION GIVEN AT THE BEGINNING OF THE PRO-      *
*  GRAM, WE SEE THAT THE SUBSCRIPTS IN A, B, AND C ARE SYS-     *
*  TEMATICALLY RELATED TO THOSE OF E:                           *
*    A'S SUBSCRIPT ALWAYS IS 5 GREATER THAN E'S SUBSCRIPT       *
*    B'S SUBSCRIPT ALWAYS IS 7 GREATER THAN E'S SUBSCRIPT       *
*    C'S SUBSCRIPT ALWAYS IS ONE MORE THAN TWICE E'S            *
*  THUS, EACH TIME WE COMPUTE AN ELEMENT OF E, SAY, E(I),       *
*  WE WOULD USE A(I+5). B(I+7) AND C(2*I+1) TO COMPUTE THAT     *
*  ELEMENT. HAVING WORKED THIS OUT, WE CAN USE A LOOP THAT      *
*  WILL GO THROUGH FIVE CYCLES, ONE FOR EACH OF E'S ELEMENTS*
*****************************************************************
      DO I=1,5
         E(I) = A(I+5) + B(I+7) + C(2*I+1)
      END DO

*****************************************************************
*  NOW, A LOOP TO PRODUCE SMDSQ. WE USE D(I)*D(I) INSTEAD OF*
*  D(I)**2 BECAUSE EXPONENTIATION IS UNNECESSARILY COMPLEX      *
*  FOR SOMETHING AS SIMPLE AS SQUARING A VALUE                  *
*****************************************************************
      DO I=1,20
         SMDSQ = SMDSQ + D(I)*D(I)
      END DO

*****************************************************************
*  NOW A SIMPLE LOOP TO COMPUTE SUM1                            *
*****************************************************************
      DO I=1,20
         SUM1 = SUM1 + A(I)*B(I)
      END DO

*****************************************************************
*  FINALLY, A LOOP FOR SUM2. HERE WE NEED A TEST FOR EACH       *
*  ELEMENT OF A TO SEE WHETHER IT IS ODD. THE MOD FUNCTION      *
*  WILL BE HELPFUL IN THIS TEST                                 *
*****************************************************************
      DO I=1,20
         IF (MOD(A(I),2.0) .NE. 0.0) SUM2=SUM2+A(I)*B(I)
      END DO
*
*****************************************************************
*  ALL COMPUTATIONS COMPLETE; READY TO PRINT THE RESULTS        *
*****************************************************************
      PRINT *, 'ARRAY C:'
      PRINT *, C
      PRINT *, '        '
      PRINT *, 'ARRAY D:'
      PRINT *, D
      PRINT *, '        '
      PRINT *, 'ARRAY E:'
      PRINT *, E
      PRINT *, '        '
      PRINT *, 'SMDSQ: ',SMDSQ
      PRINT *, 'SUM1: ',SUM1
      PRINT *, 'SUM2: ',SUM2

      PRINT *, 'RUN COMPLETED.'
      STOP
      END
```

**FIGURE 7.10** *(continued)*

```
ARRAY A:
   0.1000000E 01  0.2000000E 01  0.3000000E 01  0.4000000E 01  0.5000000E 01  0.6000000E 01  0.7000000E 01
   0.8000000E 01  0.9000000E 01  0.8000000E 01  0.7000000E 01  0.6000000E 01  0.5000000E 01  0.4000000E 01
   0.3000000E 01  0.2000000E 01  0.1000000E 01  0.3000000E 01  0.6000000E 01  0.9000000E 01

ARRAY B:
   0.0000000E 00  0.1000000E 01  0.2000000E 01  0.3000000E 01  0.4000000E 01  0.5000000E 01  0.6000000E 01
   0.7000000E 01  0.8000000E 01  0.7000000E 01  0.6000000E 01  0.5000000E 01  0.4000000E 01  0.3000000E 01
   0.2000000E 01  0.1000000E 01  0.0000000E 00  0.2000000E 01  0.5000000E 01  0.8000000E 01

ARRAY C:
   0.2000000E 01  0.5000000E 01  0.8000000E 01  0.1100000E 02  0.1400000E 02  0.1700000E 02  0.2000000E 02
   0.2300000E 02  0.2600000E 01  0.2300000E 01  0.2000000E 02  0.1700000E 02  0.1400000E 02  0.1100000E 02
   0.8000000E 01  0.5000000E 01  0.2000000E 01  0.8000000E 01  0.1700000E 02  0.2600000E 02

ARRAY D:
   0.5000000E 00  0.3464100E 00  0.2795085E 00  0.2405228E 00  0.2142857E 00  0.1950955E 00  0.1802775E 00
   0.1683905E 00  0.3846154E-01  0.4347826E-01  0.5000000E-01  0.5882353E-01  0.7142854E-01  0.9090906E-01
   0.1250000E 00  0.2000000E 00  0.5000000E 00  0.1250000E 00  0.5882353E-01  0.3846154E-01

ARRAY E:
   0.2100000E 02  0.2900000E 02  0.3500000E 02  0.4100000E 02  0.3300000E 02

SMDSQ:     0.9996974E 00
SUM1:      0.2200000E 03
SUM2:      0.2860000E 03
```

**FIGURE 7.11** Printout from Example 7.2.

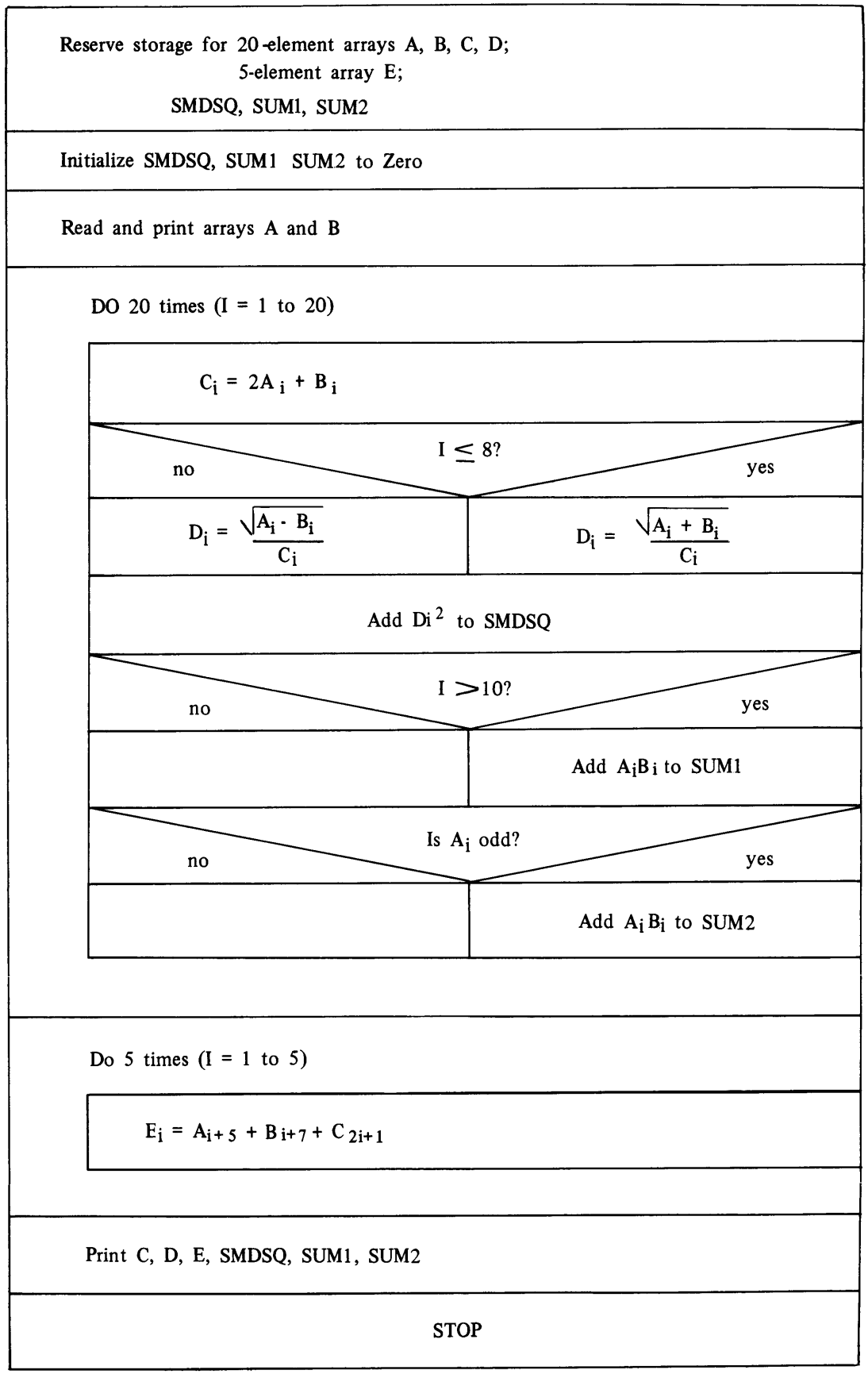



**FIGURE 7.12 (a)** Revised Flowchart for Example 7.2.

```
"Define 20-element arrays A, B, C, D,
        5-element array E,
        SMDSQ, SUM1, SUM2."
"Initialize SMDSQ, SUM1, SUM2 to zero."
"Read and echo A and B."
do for all 20 elements (I = 1 through 20):
    "Compute C(I) = 2A(I)+B(I)."
    if
        I is less than or equal to 8
    then
        "Compute D(I) = sqrt(A(I)+B(I))/C(I)."
    else
        "Compute D(I) = sqrt(A(I)-B(I))/C(I)."
    endif
    "Add D(I)**2 to SMDSQ."
    if
        I is greater than 10
    then
        "Add A(I)*B(I) to SUM1"
    else
    endif
    if
        A(I) is odd
    then
        "Add A(I)*B(I) to SUM2."
    else
    endif
enddo
do 5 times (I = 1 to 5):
    "Compute E(I) = A(I+5) + B(I+7) + C(2I+1)."
enddo
"Print C, D, E, SMDSQ, SUM1, SUM1."
"Stop."
```

**FIGURE 7.12 (b)** Pseudocode for Revised Example 7.2.

The shortcomings of this approach make themselves painfully felt as soon as we want to read a longer list of elements. Another way might be to set up a DO loop:

```
PRINT *, 'ENTER VALUES FOR MRGN(3) THROUGH MRGN(8)'
 DO I=3,8
     READ *, MRGN(I)
 END DO
```

This is straightforward enough: The loop will cycle six times, reading a single value each time. Unfortunately, it will not produce the same result as the previous READ statement in which the elements were listed explicitly. The reason is that the DO loop causes the READ statement to execute six times. Each time the statement executes, it reads a new line. Consequently, the results will be the same only if each of the input values happens to be on a separate line. However, we can handle this situation more concisely by using FORTRAN's feature for including groups of elements in a single statement.

Part of an array can be read or written with a READ or PRINT statement having a DO loop built into it. We shall see how this is done by using a 12-element integer array named MRGN. If we wanted to read values into the first five elements, we would say

```
READ *, (MRGN(I), I=1,5)
```

```
C*************************************************************
C                    EXAMPLE 7.2 (REVISED)                    *
C*************************************************************

      PROGRAM    EX702R
      IMPLICIT   NONE
      REAL*4     A(20),B(20),C(20),D(20),E(5),SMDSQ,SUM1,SUM2
      INTEGER*2  I

      SMDSQ = 0.0
      SUM1 = 0.0
      SUM2 = 0.0
      PRINT *, 'ENTER VALUES FOR ARRAY A, THEN FOR ARRAY B'
      READ *, A,B
      PRINT *, 'ARRAY A: '
      PRINT *, A
      PRINT *, 'ARRAY B: '
      PRINT *, B
      PRINT *, '         '

C*************************************************************
C  COMPUTE THE VALUES FOR C AND D, AS WELL AS THE SINGLE     *
C  VALUES FOR SMDSQ, SUM1 AND SUM2. THE SINGLE LOOP CYCLES   *
C  20 TIMES AND TESTS ARE ATTACHED TO THE APPROPRIATE PRO-   *
C  CESSES TO AVOID USING THOSE ARRAY ELEMENTS THAT DO NOT    *
C  BELONG IN THOSE PROCESSES.                                *
C*************************************************************

      DO I=1,20
         C(I) = 2.0*A(I)+B(I)

         IF (I .LE. 8) THEN
           D(I) = SQRT(A(I)+B(I))/C(I)
         ELSE
           D(I) = SQRT(A(I)-B(I))/C(I)
         END IF

         SMDSQ = SMDSQ+D(I)

         IF (I .GT. 10) SUM1=SUM1 + A(I)*B(I)
         IF (MOD (A(I),2.0) .NE. 0.0) SUM2=SUM2 + A(I)*B(I)

      END DO

C*************************************************************
C  SINCE ARRAY C IS AVAILABLE, WE CAN SET UP ANOTHER LOOP    *
C  NOW TO TAKE CARE OF COMPUTING THE ELEMENTS FOR ARRAY E.   *
C*************************************************************

      DO I=1,5
        E(I) = A(I+5) + B(I+7) + C(2*I+1)
      END DO
```

**FIGURE 7.13** Internal Processing for Revised Example 7.2.

The specification inside the parentheses is called an *implied* DO *loop*. Note that the index variable (I in this case) is no different in usage from the one in an ordinary DO loop. The concluding statement number is not specified in the implied loop because it is not needed since the loop can be only one statement long.

Implied DO loops also can be used to read or write parts of multidimensional arrays. However, this usage requires nested loops (loops inside each other), and we shall defer their discussion till later.

## 7.4 SUMMARY

An *array* is a collection of data values (*elements*) all of one type (e.g., integer) and all related by some common aspect (e.g., populations of a group of cities, batting averages for a team's players, or aluminum concentrations in a group of deodorants). Each of the elements in an array is identified by the array's (collective) name and one or more *subscripts* denoting that element's relative position in the array. A subscript is needed for each *dimension*.

Arrays are declared using the INTEGER, REAL, DOUBLE PRECISION, CHARACTER, or LOGICAL statements as for single-valued variables (Chapter 4). In addition to the name, the declaration includes a set of specifications showing the subscript range for each of the array's dimensions. Thus,

```
REAL*4 PLSRT (6, 5, 3)
```

declares a three-dimensional array with $5 \times 6 \times 3 = 90$ elements organized as 5 *rows*, 6 *columns* and 3 *blocks*. FORTRAN stores array elements in *column major order* i.e., the leftmost (row) subscript varying the most frequently.

An entire array can be read or written by specifying the collective name with no subscripts, e.g.,

```
READ *, PLSRT
```

Parts of arrays can be processed by setting up loops in which each cycle works on an element, with the subscript being systematically changed as part of the loop's control mechanism. Thus, the sequence

```
INTEGER*2       CRL (20)
PRINT *, 'ENTER 20 VALUES FOR CRL'
READ *, CRL
DO J = 12, 20
      CRL (J) = 2*CRL (J)
END DO
```

reads all 20 elements of array CRL and doubles the values in the last nine elements.

## PROBLEMS

1. For each of the following array declarations, specify the number of dimensions, the number of elements in each dimension, the lowest and highest subscript values for each dimension, and the total number of elements in the array:

   (a) REAL*4 ROSTER (12)
   (b) INTEGER*2 CNTRS (31)
   (c) REAL*4 STRS (4, 5)
   (d) REAL*8 LNGVAL (3, 2, 6)
   (e) INTEGER*2 EXTBL (2, 3, 2, 4, 3)
   (f) REAL*4 PRVAL (5: 10)
   (g) REAL*4 SMTVAL (6, 5: 10)
   (h) INTEGER*2 NDXES (0: 19, 0: 10)
   (i) REAL*8 CFNTS (−5: 5, 8, 0: 10)
   (j) INTEGER*2 NVNS (−10: 0, −10: −5, 6: 7)
   (k) DIMENSION XTRAS (5, 5, 2, 2)
   (l) DIMENSION INTRAS (4, 6, 4)
   (m) INTEGER*4 BIGNUM (3: 8, 4)
   (n) REAL*8 XTEND (0: 2, 2: 4: 6)

2. Write an appropriate FORTRAN statement for each of the following declarations:

   (a) A one-dimensional real array LIMITS having 18 elements with a lowest subscript of 1.
   (b) A one-dimensional integer array SUMS having 26 elements with a lowest subscript of zero.

(c) A one-dimensional real array ENTROP having 12 elements with highest subscript 12.
(d) A one-dimensional integer array NUMS having 12 elements with highest subscript 10.
(e) A two-dimensional real array TMPS having 12 rows and 6 columns with lowest subscripts being 1 for both dimensions.
(f) A two-dimensional real array PRES having 4 rows and 7 columns with lowest row subscript 1 and lowest column subscript 7.
(g) A two-dimensional double precision array PRECIS having 36 elements organized into an equal number of rows and columns with lowest subscript 0 for both dimensions.
(h) A $3 \times 8 \times 5$ real array named PRICES with highest subscripts 6, 8, and 0, respectively, for rows, columns and blocks.
(i) Two real 12-element arrays named EAST and WEST, each having a highest subscript value of 11.
(j) Three integer arrays: A 30-element one-dimensional array NMLST with highest subscript $-10$; a $4 \times 5$ array WDTHS with minimum row subscript 1 and maximum column subscript $-5$; a 27-element three-dimensional array QBC having an equal number of rows, columns and blocks with maximum subscript values of $-1$ in each dimension.

3. Shown below are the values assigned to an array declared as

```
REAL*4   ELEM(10)
```

Using this array, give the value for each of the following:

```
3.0    -6.0    4.5    2.2    6.0    0.0    5.2    4.0    1.5    -3.0
```

(a) ELEM(7)
(b) ELEM(4) * ELEM(9)
(c) ELEM(2) * (3*ELEM(3) + 3)
(d) The name of the smallest element
(e) The name of the element whose value is equal to ELEM(8) *ELEM(9)
(f) The name of the second largest element
(g) The sum of all the elements with odd-numbered subscripts
(h) Assume that we have declared a real variable named SMVAL and an integer variable named I. Show that value that is printed as a result of the following sequence of statements:

```
SMVAL = 0.0
DO I = 1,10
     SMVAL = SMVAL + ELEM(I) - I
END DO
PRINT *, SMVAL
```

(i) Using SMVAL and I as in part (h), show that value that is printed by the following sequence of statements:

```
   SMVAL = 0.0
   DO 22 I = 2,10
        SMVAL = SMVAL + (I-1)*ELEM(I-1)
22 CONTINUE
   PRINT *, SMVAL
```

(j) Assume the following declarations:

```
REAL*4     NEWARY(10), SMVAL
INTEGER*2  I
```

show the values that are printed as a result of the statements given below:

```
SMVAL = 0.0
DO I=1,10
     NEWARY(I) = ELEM(I)+1
     IF (I .LT. 6) THEN
       SMVAL = SMVAL+ELEM(I)
     ELSE
       SMVAL = SMVAL-NEWARY(I)
     END IF
END DO
```

```
PRINT *, NEWARY(1), NEWARY(2), NEWARY(3), NEWARY(4), NEWARY(5)
PRINT *, NEWARY(6), NEWARY(7), NEWARY(8), NEWARY(9), NEWARY(10)
PRINT *, SMVAL
```

4. Given below are the values assigned to an array declared as

```
INTEGER LTS(3, 6)
```

Answer the following questions involving this array:

| | | | | | |
|---|---|---|---|---|---|
| 24 | −4 | 10 | 0 | 8 | −12 |
| 6 | 20 | 2 | 32 | 4 | 5 |
| −16 | 1 | 25 | 54 | 3 | 0 |

(a) Give the sum of the elements in Column 3.
(b) Write an expression for the product of the three largest elements.
(c) Give the value of LTS(1, 2)+LTS(2, 1)−(LTS(3, 4)−LTS(1, 6)).
(d) Specify which elements contain the same value.
(e) Give the sum of all the elements having the same row and column number.
(f) Which element has the value MAX(LTS)?, 0.25*MIN(LTS)?
(g) Specify the location of the largest element having an even row number and odd column number.
(h) Evaluate the expression DIM(LTS(3, 1), LTS(1, 6)).
(i) Let us say that we wish to refer to a pair of values in this array as LTS(I1, J1) and LTS(I2, J2). List the locations of all pairs of elements in this array such that LTS(I2, J2) = 2*LTS(I1, J1).
(j) Suppose that variables VAL, R and C are declared as integers. Show the value that is printed as a result of the following statements:

```
      VAL = 0
      R = 2
      DO 10 C=1, 6
           IF (C .EQ. R) THEN
             VAL = VAL + LTS(R, C−1) − C
           ELSE
             VAL = VAL + LTS(R, C)
           END IF
10    CONTINUE
      PRINT *, VAL
```

(k) Using VAL, R and C as in part (j), show the value that is printed as a result of the following statements:

```
VAL = 0
DO C = 1, 6
   R = MOD(C−1, 3) + 1
   VAL = VAL + R*LTS(R, 7−C)
END DO
PRINT *, VAL
```

5. Assume the following declaration:

```
REAL*4 BXT(24)
```

write the appropriate FORTRAN statement (or statements) for each of the following:

(a) Use a loop to initialize all of BXT's elements to 1.0.
(b) Perform the same initialization as in (a) using a DATA statement.
(c) Use a loop to initialize all of BXT's odd-numbered elements to 0.0 and all of its even-numbered elements to 1.0.
(d) Set up the same initialization as in (c) using a DATA statement.
(e) Use a loop to assign 0.0 to all but the last four elements of BXT. Assign 4.0 to each of those last four elements.
(f) Duplicate the assignments in (e) using a DATA statement.

(g) Use an appropriate method to assign −1.0 to BXT's first element, −2.0 to the second element, ....., −24.0 to BXT's last element.
(h) Use an appropriate method to assign 1.0 to the first four elements, 2.0 to elements 5–8, 3.0 to elements 9–12, and so on, so that elements 21, 22, 23, and 24 each will have a value of 6.0.
(i) Use an appropriate method to assign a value of 1.0 to BXT(1), BXT(5), BXT(9), BXT(13), BXT(17), and BXT(21); a value of 2.0 to BXT(2), BXT(6), BXT(10), . . . , BXT(22); a value of 4.0 to BXT(3), BXT(7), . . . , BXT(23), and a value of 8.0 to BXT(4), BXT(8), . . . , BXT(24).
(j) Read BXT's values in from a line.
(k) Initialize BXT's values to 0.0 and read BXT(5) and BXT(8) from a line.
(l) Initialize BXT's values to 0.0 and read BXT(17) through BXT(22) from a line.
(m) Initialize BXT's values to 0.0 and read the first 12 elements from a line.

6. Using the following declaration,

```
REAL*4      BXT(-5:18)
INTEGER*2   J
```

show the values in BXT as a result of each of the following activities. Treat each problem independently:

(a)
```
DO J=1,24
    BXT(J-6)=3.0+2.0*J
END DO
```

(b)
```
BXT(-5)=1.0
DO 10 J=2,24
    BXT(J-6)=J-BXT(J-7)
10 CONTINUE
```

(c)
```
BXT(-5)=2.0
DO J=-4,18
    BXT(J)=3.0+2.0*J-BXT(J-1)
END DO
```

(d)
```
DATA BXT/24*4.0/
```

(e)
```
DATA BXT/6*0.0,12*1.0,6*2.0/
```

(f)
```
DATA (BXT(J), J=-5,0)/6*8.0/(BXT(J), J=1,18)/18*7.7/
```

(g)
```
DATA BXT/24*0.0/
READ *, BXT(1), BXT(5)
```
(use this line):
```
12.0  -2.4  54.0  72.0  5.5  21.6  -3.3
```

(h)
```
DATA BXT/24*0.0/
READ *, (BXT(J), J=-1,3)
```
(use the line in part (g))

(i)
```
DATA BXT/12*4.0,12*2.0/
READ *, BXT(-3), (BXT(J), J=16,18)
```
(use the line in part (g))

7. Given the following declaration,

```
INTEGER*2   BVAL(5,3)
```

write the appropriate FORTRAN statement (or statements) to perform each of the following activities:

(a) Use a data statement to initialize the entire array to zeros.
(b) Use a DATA statement to set BVAL's first three rows to zeros and the last two rows to 7s.
(c) Use a DATA statement to set all but the last column to 5s, with the last column being set to 6s.
(d) Use a DATA statement, along with appropriate assignments, to set all but the eleventh element to 4, with the eleventh element being set to 8.
(e) Use a DATA statement, along with other statements as appropriate to set the elements in the second column to 4, 5, and 6, and all the other elements to 3.
(f) Use a DATA statement, along with other statements as appropriate to set the first column to 7, the second column to 8, and the third column, with the exception of element BVAL(2,3) to −3. BVAL(2,3) is to be given a value equal to the sum of the eighth and fourteenth elements.

8. Using the following declaration,

```
INTEGER*2   VAL2(4,3),R,C
```

draw the two-dimensional representation of VAL2 and fill in the values resulting from each of the activities specified below. Treat each problem independently:

(a)
```
DATA VAL2/12*6/
```
(b)
```
DATA VAL2/0,1,2,0,1,2,0,1,2,3*4/
```
(c)
```
DATA VAL2/5*-3,4,4,8,9,3,2*9/
```

(d)
```
DATA VAL2/12*4/
C=3
DO R=2,4
    VAL2(R,C)=2*VAL2(R,C)
END DO
```

(e)
```
READ*, VAL2
```
(use the line shown below);
```
4 0 9 8 4 7 2 12 74 5 -3 11
```

(f)
```
READ*, VAL2(1,2),VAL2(2,3)
```
(use the line in problem (e))

(g)
```
READ*, VAL2(1,3),VAL2(1,3)
```
(use the line in problem (e))

9. Change the program in the revised Example 7.2 so that it will process any number of arrays. Stop the run when A(1) and B(1) have the same value.

10. The Worldwide Rainfall Survey gathers and publishes monthly rainfall data for a large number of cities. Data for each city are prepared on a line containing the name of the city (up to 15 letters) and blanks (e.g., Des Moines), and the twelve monthly figures, the latter given to the nearest centimeter. Write a program that processes any number of these rainfall input lines. For each one, the program is to print the name of the city on a separate line, followed by a line for each of the twelve months showing two values: The rainfall as reported, and the rainfall as a fraction (to the nearest thousandth) of the city's maximum value for the year. After that, the program is to print the average rainfall (to the nearest tenth of a centimeter) and the month number in which the maximum rainfall occurred. Leave a blank line between sets and stop the run with a city named $$$$$$$. The following input may be used:

```
'ARID CITY'   3   2   12   14   18   16   14   13   11   9   6   4
'DRYBURGH'    8   12   15   18   22   31   43   44   41   32   38   38
'DAMPTOWN'   22   43   31   87   101   97   123   85   77   75   65   78
'HUMID POINT'   76   82   101   118   144   187   167   171   87   84   43   56
'WETVILLE'   89   87   74   129   156   202   256   313   329   280   211   198
```

11. Expand the program in Problem 10 so that, in addition to the output described there, it prints the following: After the last city's data have been processed, the program prints the average rainfall (to the nearest tenth of a centimeter) over all the cities for each of the twelve months.

12. Professor Zoozl's exam in Extraterrestrial Literature consists of fifteen questions worth 30 points apiece. Each student's scores are recorded on a separate line containing the student's name (up to 15 letters) and the points given for each of the questions. No fractional points are given. Write a program that processes a succession of such input lines and, for each one, prints the raw score (total number of points earned) and the score expressed as a percentage (to the nearest tenth of a percent) of the total possible score (e.g., a raw score of 315 is equivalent to 315/450 or 70.0%). After the last student's data have been processed, leave a blank line and print three lines: The first one shows the number of students processed, the second one shows the minimum and maximum scores (expressed as raw scores and percents), and the last line shows the average score, expressed both ways. Express the average to the nearest tenth of a point and stop the run with a student named LASTFELLOW.

13. Another grading problem: This time, it is the final exam in Professor Mandible's course in New Words from American Advertisers. The test consists of 25 multiple choice questions, each answerable by 1, 2, 3, 4, or 5. A student's answer contains his or her name (up to 12 letters) and the 25 answers. The correct answers, each worth one point, are entered on a separate line along with the "name" KEYCARD and that line is submitted ahead of the student data. The last student's input line is followed by a fake input line with student name NOMORE. Write a program that processes an arbitrary number of input lines. For each student line, the program prints a line with the student's name and the number of questions answered correctly. After the last student line, the program leaves a blank line and prints three summary lines showing the number of tests processed, the highest and lowest scores, and the average score to the nearest tenth of a point. The following data may be used:

```
'KEYCARD'      1,1,2,3,5,4,3,2,4,3,2,1,2,2,3,2,1,4,3,2,3,4,5,3,4
'W. JOOJIK'    1,2,4,3,5,4,3,2,3,2,3,1,3,1,1,2,3,4,3,4,2,2,2,1,1
'O. GOODBOY'   1,1,2,3,5,4,3,2,4,3,2,1,2,2,2,2,1,4,3,2,3,4,5,3,4
'L. ZUMZUM'    1,1,5,5,4,3,2,3,4,3,2,3,4,3,2,3,4,3,2,3,4,3,2,3,4
'B. DIGRUT'    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
'M. BACKWD'    1,1,3,2,4,5,3,2,1,2,3,4,3,3,2,3,1,4,3,2,3,4,5,3,4
'NOMORE'       5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5
```

14. Professor Yehyeh also uses a multiple choice exam for his class in Casual Living, except that he assigns different point amounts to each of his 25 questions. Accordingly, the KEYCARD data are followed by a

second line with the "name" WEIGHTS containing the point values for each of the questions. Output is to be the same as for the last problem with the added requirement that each student's score also be expressed as a percent (to the nearest tenth of a percent). Here are weighting data you can use:

```
'WEIGHTS'  1,1,1,2,4,3,1,6,3,3,2,2,7,1,1,1,5,8,2,1,1,3,3,2,1
```

15. After some thought, Professor Yehyeh has decided that he is interested in the relative "difficulty" of each question, based on how many people answered correctly. Thus, you are asked to process the test data as in the previous problem and, in addition, print the number of correct answers for each of the 25 questions.

16. In Problem 21 of Chapter 5, formulas were given for computing the slope (A1) and intercept (A0) for a straight line that provides the best fit to a set of X–Y data. This was called the least squares line because of the measure that is used as an indicator of how good the fit is. This indicator, D, is computed as the sum of the squared differences between each observed Y value and the one computed from the straight line equation at the same X value where the experimental Y was measured. Mathematically, D can be expressed as follows:

$$D = \sum_{i=1}^{n} (Y_i - A0 - A1X_i)^2$$

Anyway, the least squares line is that line for which D is the smallest it can be. Write a program that reads a set of 12 X–Y data points (one point per line) and prints three lines of output: The slope, the intercept, and D.

17. Expand the program in Problem 16 so that it processes any number of 12-point output sets, producing three lines of input for each set, separated by a blank. Stop the run with a data point of (−100., −100).

18. A1 and A2 each are 8-element one-dimensional real arrays and EX is an 8-element one-dimensional integer array. Each input set consists of 8 lines with each line containing four values: Corresponding values for an element in each of the three arrays and an integer (from 1–8) indicating which three elements they are. For instance, a line that looks like this

```
-22.7     303.9     3     4
```

contains a value of −22.7 for A1(4), 303.9 for A2(4) and 3 for EX(4). The fourth value on the line (i.e., the 4) indicates that the first three values are intended for the fourth elements of their respective arrays. Write a program that reads and processes sets of these lines as follows: For each set, the program is to print the following:

(a) Eight lines, each one showing corresponding elements of A1, A2 and EX.
(b) Eight lines, each one showing corresponding elements of arrays B, C, D, and E where:

$$B_i = A1_i + A2_i - EX_i$$

$$C_i = \frac{A1_i}{A2_i}^{EX_i}$$

$$D_i = A2_i^{EX_i} - A1_i^{EX_i - 1}$$

$$E_i = \frac{EX_i(A1_iA2_i)^{EX_i+1}}{\sqrt{A1_i^{EX_i} + A2_i}}$$

19. BINSTR is a 16-element integer array in which each value is either a 1 or a 0. Write a program that reads in a succession of values for BINSTR and processes each set as follows: For each set, the program prints the length of the longest string of identical values in that set. If the longest string is a string of ones, the program prints the word ONES after the length; if the longest string is a string of zeros, the program prints the word ZEROS after the length. For example, if the input set looks like this,

```
1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 1
```

the output would say

```
LONGEST STRING:   6   ONES
```

Assume that there will be no more than one longest string in a set. Stop the run with an input set whose first value is 2.

20. Write the same program as in the previous problem with one important difference: Instead of being an array, BINSTR is an ordinary single-valued integer variable.

21. Using the same requirements given in the Problem 19, write a program in which you cannot make the assumption stated in that problem. If there is more than one longest string, base your message on the first one you find. For instance, if the input set looks like this,

    0 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0

    the output would say

    LONGEST STRING: 7 ZEROS

22. Using the same requirements given in either Problem 19, 20, or 21, include in your output the starting position of the longest string. For the example input set in Problem 21, then, the output would say

    LONGEST STRING: 7 ZEROS STARTS AT POSITION 6

23. A and B are 15-element one-dimensional integer arrays. Write a program that reads a set of values for A followed by a single integer SHIFT which may have a value from −10 to +10 (including zero). Using these data, develop values for array B such that its elements are the same as those in A, but their positions are shifted by an amount equal to SHIFT. For example, if SHIFT's value is +2, then B(3) has the same value as A(1), B(4) has the same value as A(2), B(5) has the same value as A(3), and so on. In order to make the computations complete, the shifting process "wraps around" the arrays so that, in this example, B(1) would receive A(14)'s value and B(2) would receive A(15)'s value. A negative shift works the same way: If SHIFT were −4, for instance, A(1)'s value would be copied into B(12), A(2)'s into B(14), A(3)'s into B(15), and so on. For each set processed, the program is to print an output line showing SHIFT's value, followed by 15 lines with each line showing a corresponding pair of elements from A and B. A SHIFT of −100 stops the run.

# 8
# *Character Data*

If we relied solely on the popular press for our technical information about computers, it would be easy to form the idea that computers deal only with numbers. However, we have already seen (Chapter 4) that FORTRAN takes a wider view of "computing" by recognizing characters and logical signals as legitimate types of data. In this chapter we take a closer look at character data and their processing.

## 8.1 SPECIFICATION OF CHARACTER STRINGS

The basic character string declarations are familiar by now, so that we need only review them briefly. Once that is done, we can consider some additional features.

### 8.1.1 Declaration of Character Strings

Recall that the length of a character variable must be declared along with its name and type. Thus, the statements

```
CHARACTER*9 W1, W2, W4
CHARACTER*12 W3
```

define four character variables: Three of them (`W1`, `W2` and `W4`) can accommodate nine characters each and the fourth (`W3`) can hold 12 characters. Note that the length given with the word `CHARACTER(9` in this example) serves as the default for the entire statement.

Although it is an entire character string that generally means something to the programmer using it, there are times when it is also desirable to deal with part of a string. Features in FORTRAN 77 enable us to identify such parts, called *substrings*, and manipulate them without affecting the rest of the string from which they are isolated.

### 8.1.2 Defining Substrings

To understand how a substring is identified, we need to know that the individual characters in a string have numbered positions in that string, with the numbering starting at `1` and going from left to right. For example, if we said

```
CHARACTER*6 WRD1
DATA WRD1/'FOREST'/
```

the six characters `FOREST` would occupy positions `1`–`6` of the string `WRD1`. (As a specific illustration, position `4` of `WRD1` would have the character `E` in it.) Now, with this rule defined, it becomes an easy matter to describe a substring: We simply specify the name of the string and the starting and ending positions of the part we want to use. With `WRD1` as declared and initialized before, the specification

```
WRD1(1:3)
```

**Table 8.1** Specification of Substrings
C is declared as CHARACTER C*12; M, N are declared as INTEGER M, N

| *Value in C* | *Substring* | *Specified Value* |
|---|---|---|
| STRATOSPHERE | C(1:4) | STRA |
| | C( :4) | STRA |
| | C(10:12) | ERE |
| | C(10: ) | ERE |
| | C(4:7) | ATOS |
| | N=3; M=4 | |
| | C(N:M) | RA |
| | C(7:M) | Illegal; 7 M M |
| ARRANGEbbbbb | C(3:5) | RAN |
| | C(3:7) | RANGE |
| | C(3: ) | RANGEbbbbb |
| | C(8: ) | bbbbb |
| | M=3; N=M+4 | |
| | C(2*M:N+3) | GEbbb |

describes the substring consisting of the first three positions of WRD1 (i.e., positions 1 through 3). The value currently in that substring consists of the characters FOR. Similarly, if we say

```
WRD1(1:1)
```

we are referring to the substring consisting of WRD1's first character, and

```
WRD1(6:6)
```

isolates WRD1's final (sixth) character. Of course, it is up to the programmer to make sure that the substring description makes sense. For instance, a specification of

```
WRD1(5:7)
```

will be rejected by FORTRAN because WRD1 was declared with a length of 6, and there is no seventh position. Similarly, something like

```
WRD1(3:1)
```

would be unacceptable because it violates FORTRAN 77's left-to-right position numbering. Additional examples of substring identification are shown in Table 8.1.

Once a substring is specified, we can think of it as having an existence of its own: We can read into it, print from it, compare it to other strings, and change its value by assignment without changing the rest of the string. The versatility of substrings is extended even further by the fact that the beginning and end of the substring, shown as constant values in the previous examples, also may be variables. We shall make use of this flexibility later on.

### 8.1.3 EQUIVALENCE and Character Strings

The naming capabilities provided by FORTRAN's EQUIVALENCE statement (Chapter 4) are available for character strings as well as numerical data. At the simplest level, this means that we can assign several names to the same character string. For instance, the statements

```
CHARACTER*7 WRIT, SYN
EQUIVALENCE (WRIT, SYN)
```

declares that the 7-character string WRIT also may be referred to as SYN. In other words, WRIT and SYN refer to the same storage locations.

EQUIVALENCE also enables us to establish associations between parts of character strings. One convenient use for this is to define a separate name for a single character in a string. To illustrate, let us say that we intend to store somebody's last name in a 15-character string called LNAME. At the same time, we would like to deal separately with the first letter of LNAME. Of course, we have seen in the previous section that we can refer to that position by identifying the substring LNAME(1:1). However, it may be more convenient to be able to use a distinct name for the first letter. If FLETR is the desired name for that position, we can set this up by writing

```
CHARACTER LNAME*15, FLETR*1
EQUIVALENCE (LNAME, FLETR)
```

This says that the first position of LNAME and the first (and only) position of FLETR refer to the same storage location. Since FLETR has no second position, the association goes no further.

The same kind of association can be defined using any substring. Suppose it turned out to be useful to refer to the last three letters of LNAME with a separate name. The following statements

```
CHARACTER LNAME*15, FLETR*1, LAST3*3
EQUIVALENCE (LNAME, FLETR), (LNAME(13:15), LAST3)
```

establish:

1. The location of the first position of LNAME to be the same as the first (and only) position of FLETR;
2. The location of the first position of the specified substring (i.e., the 13th position of LNAME) to be the same as the first position of LAST3. Since LAST3 is declared with a length of 3, the EQUIVALENCE also establishes the sequence of positions. Thus, once we see that the first position of LAST3 has the same location as the 13th position of LNAME, we know that LAST3's second position has the same location as LNAME's 14th, and LAST3(3:3) and LNAME(15:15) are two names for the same location.

This is shown in the diagram of Figure 8.1.

This gives us a useful clue with regard to the way EQUIVALENCE works with character strings:

> Each pair of positions in a character string corresponds to one HP1000 storage location. For HP 32-bit computer systems like the HP9000, a location has a capacity of four characters. When two character strings are EQUIVALENCEd to each other, this specifies that the first positions of those strings refer to the same storage location. (Please pardon the atrocious English. EQUIVALENCEd is part of FORTRAN's jargon and the fight was lost years ago.)
>
> The statements above represent more than a description; they are rules that FORTRAN obeys when it responds to an EQUIVALENCE statement. As a result, we can look at more complicated situations and work out what happens simply by following the same rule FORTRAN follows. We shall put these rules to work now.

The EQUIVALENCE statement is a general resource. It may be used to define associations among any number of character strings having arbitrarily different lengths. To make sure this ability is clear, we shall look at several specifications and analyze their effects:

1. The statements

```
CHARACTER W*4, X*5, Y*3, Z*10
EQUIVALENCE (X,Z), (Y,W), (W,Z(4:7))
```

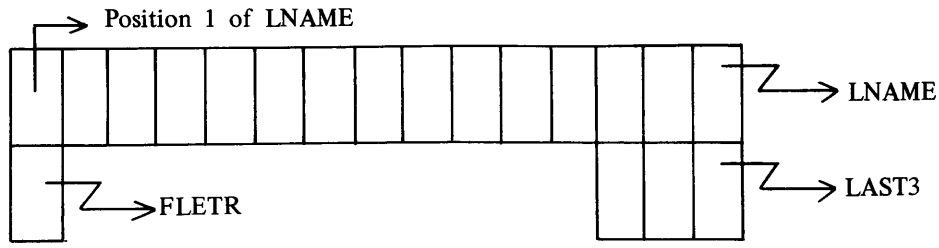



FIGURE 8.1

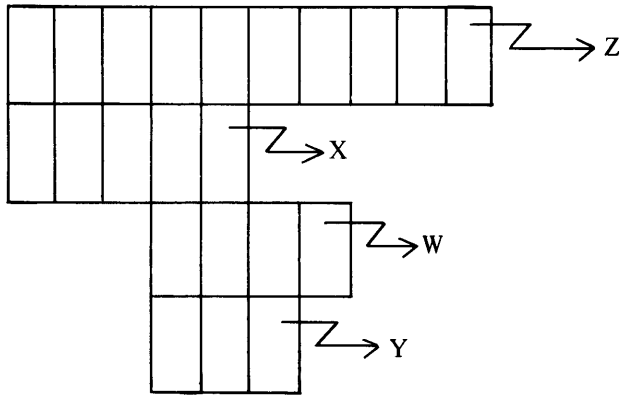



FIGURE 8.2

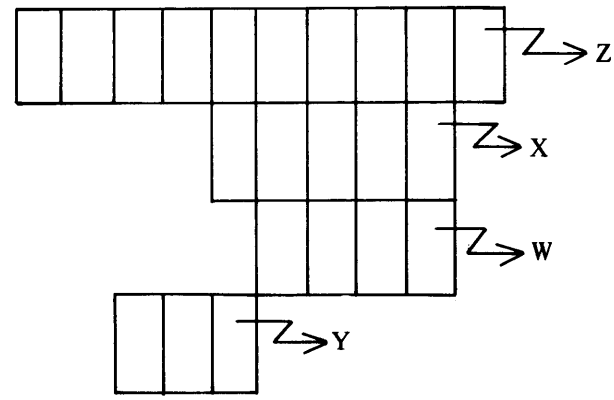



FIGURE 8.3

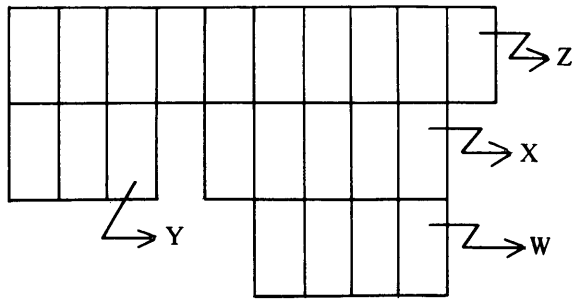



FIGURE 8.4

produce a set of associations as shown in Figure 8.2. X and Z both start at the same location, as do W and Y. W's first position is the same as Z's fourth position. Finally, since W and Y are associated with each other, that automatically associates Y with Z, even though the EQUIVALENCE statement does not actually say so.

2. The statements

```
CHARACTER W*4, X*5, Y*3, Z*10
EQUIVALENCE (X,Z(5:9)), (W,Z(6:9)), (Y,Z(3:5))
```

specify the same strings but with different associations. Now, (Figure 8.3) X, W and Y all are associated with Z and, therefore, with each other.

3. Again, we shall use the same declaration and extend one of the associations beyond Z's boundaries (Figure 8.4):

```
CHARACTER W*4, X*5, Y*3, Z*10
EQUIVALENCE (X,Z(5:9)), (W,Z(6:9)), (Y(2:),Z)
```

The notation Y(2:) refers to the substring of Y beginning at position 2 and ending at the end of the string.

### 8.1.4 Characters Strings and Numerical Data

HP FORTRAN 77 goes beyond the standard language by allowing character strings to be EQUIVALENCEd with numerical data. Since a unit of HP1000 storage generally is used to hold two characters, the declarations

```
INTEGER*2       NUMC
CHARACTER*2     PVAL
EQUIVALENCE     (NUMC, PVAL)
```

in HP1000 FORTRAN 77 produce a straightforward association between NUMC and PVAL. Similarly, the statements

```
INTEGER*2       VCOMP
CHARACTER*4     STWD
EQUIVALENCE     (VCOMP, STWD(3:))
```

indicate that the word in which an integer value for VCOMP is to be stored is the same word (i.e., has the same address) as that associated with the third and fourth characters of STWD. When applied to the HP9000, where each word has a capacity of 32 bits, the first sequence of declarations still has the same effect: Both NUMC and PVAL refer to the same word, but only half of it is used. The second sequence of declarations reserves one 32-bit word of HP9000 storage. The full capacity is used for STWD, but only half of it is considered when the name VCOMP is used as the reference.

The programmer must use caution in defining these associations to make sure they line up. For instance, the statements

```
CHARACTER*2     PVALUE
INTEGER*2       VCOMP
EQUIVALENCE     (PVALUE(2:),VCOMP)
```

is likely to cause the compiler to deliver an error message because it would appear that we were forcing HP FORTRAN 77 to start an integer variable in the middle of a word. The same type of error would result on the HP9000 with the sequence

```
CHARACTER*4     QVALUE
INTEGER*4       WCOMP
EQUIVALENCE     (QVALUE(3:),WCOMP
```

### 8.1.5 Character Arrays

Character arrays are declared like any other arrays. For instance, the statement

```
CHARACTER VOCAB(18)*6, WLIST(4,6)*8, WORD*12
```

sets up an 18-element one-dimensional array of 6-characters, a 24-element (4 rows by 6 columns) array of 8-character strings, and a 12-character string named WORD.

Since array elements may be treated independently, we can expect to do anything with such an element that we can do with a separately declared string. For one thing, we can identify substrings of array elements. The form for doing this is necessarily a little long-winded, but it is simple enough: First we specify the subscript(s) to identify the element of interest, and then we describe the desired part of that element. For example, using VOCAB, WLIST and WORD as declared before, the specification

```
VOCAB(4)(3:5)
```

identifies a string consisting of the contents of positions 3, 4 and 5 of VOCAB's fourth element. Similarly, the specification

```
WLIST(1,5)(1:4)
```

refers to the first four characters of the element in WLIST's first row, fifth column. Note that FORTRAN 77's substring notation allows us to say the same thing by specifying

```
WLIST(1,5)(:4)
```

That is, if the leftmost substring boundary is not given, FORTRAN assumes it to be 1.

## 8.2 PROCESSING OF CHARACTER STRINGS

Although the types of processes normally associated with character strings do not involve the arithmetic operations performed on numerical data, the general events are basically the same: Manipulations are performed in response to the program's instructions. A result is produced, in the form of a new character string, and that string is placed in (assigned to) a specified destination. Accordingly, it is both logical and simple for FORTRAN 77 to handle character string processing by means of the regular assignment statement. The evaluation of a character string expression proceeds according to a set of rules that are as consistent for those operations as the arithmetic rules are for numerical computations. Consequently, we can become familiar with character manipulations easily by building on what we already know about assignments.

### 8.2.1 Basic Character String Assignments

Direct assignment works for character strings just as it does for numerical data. To get things started, consider the following declaration:

```
CHARACTER*5 R, S, T
```

At this point, of course, the values in R, S, and T are irrelevant because they have not been initialized. We shall change that by assigning a cosntant to R:

```
R='ROUND'
```

Now, we can write

```
S=R
```

and both variables (R and S) will have the value ROUND in them. (T's value still is undefined.) Now, we thicken the plot a little by assigning a value to T as follows:

```
T='CUBE'
```

The result is that T will have the value CUBEb stored in it. (Recall that the b denotes a blank.) Confusing? Not at all. T, after all, was declared with a length of 5 and the = operation specifies *replacement.* Thus, when a value is assigned with T as its destination, it must replace all of T. This is no different from numerical assignment. Since we have not given T enough characters to fill it (obviously, the constant CUBE is one character short), FORTRAN supplies the necessary character, for which purpose it uses a blank. This process is called *padding* (a reasonable enough name) and it occurs at the right. That is, FORTRAN will do the replacement going from left to right. If the destination is longer than the value assigned to it, the remainder (on the right) is padded with blanks. Later on in this section we shall learn how to avoid padding when we want to.

**8.2.1.1 Assignments with Different String Lengths** FORTRAN 77 is designed to allow the processing of character string assignment statements with no particular length restrictions. We have already seen that a string assigned to a larger destination will be padded with blanks to fill it out. When the opposite is true, FORTRAN simply does the best it can: Starting at position 1, it fills the destination string until there is no more room. Then it

stops. for example, the statements

```
CHARACTER W*5, X*6, Y*3
W = 'BAGEL'
X = W
Y = W
```

will place BAGEL in W, BAGELb in X, and BAG in Y. Note that if we were to follow these statements with

```
W = Y
```

W would lose its BAGEL. Since Y has only three characters (BAG), all of W is replaced by BAG, padded with two blanks.

**8.2.1.2 Assignments with Substrings** Character assignments become much more powerful with the use of substrings. To illustrate, we shall use the variables from the previous section with slightly different assignments:

```
CHARACTER W*5, Y*3
W = 'BAGEL'
Y = W(2:4)
```

Y still has only three characters in it (after all, that is all that it can hold), but this time these characters are AGE because we copied them from positions 2–4 of W.

A substring also can be used as a destination. This is the way to change part of a string without affecting the rest of it. To take a closer look, we shall use more variables:

```
  CHARACTER W*5, X*6, Y*3, Z*5
1 W = 'BAGEL'
2 X = W
3 Z = W
4 Y = W(2:4)
5 W(3:5) = Y
6 X(6:) = Z(5:)
7 next statement
```

Statements 1, 2, and 3 are self-evident. In statement 4, the three positions in Y are filled with the three-character substring starting with position 2 of W. Consequently, Y will contain the value AGE. Now, those three characters will replace the three in W (statement 5) starting with position 3. As a result, W, which had BAGEL in it just prior to statement 5, now will have the GEL replaced with AGE, thereby producing BAAGE (a famous Norwegian luggage company). Statement 6 shows substrings being used on both sides of the replacement operator. Here the sixth (and final) position of X (which has a blank in it from statement 2) is replaced with the L from Z's fifth (and final) position, producing BAGELL in X. These actions are summarized in tabular form below:

| *just prior to* | *W* | *X* | *Y* | *Z* |
|---|---|---|---|---|
| statement 1 | ? | ? | ? | ? |
| statement 2 | BAGEL | ? | ? | ? |
| statement 3 | BAGEL | BAGELb | ? | ? |
| statement 4 | BAGEL | BAGELb | ? | BAGEL |
| statement 5 | BAGEL | BAGELb | AGE | BAGEL |
| statement 6 | BAAGE | BAGELb | AGE | BAGEL |
| statement 7 | BAAGE | BAGELL | AGE | BAGEL |

### 8.2.2 The Concatenation Operator

So far, the expressions we have used in character assignments have been as basic as possible: simple character strings or substrings. In addition, we can build more involved expressions by combining strings using an operation designed for that purpose. That operation is *concatenation*. The symbol for concatenation is // and the action it represents can be illustrated by the statements

```
CHARACTER*7 VRB
VRB = 'CATER' // 'ED'
```

In the second statement the constant CATER is concatenated with the constant ED to form a new string CATERED which then is stored in VRB. Of course, concatenation may be used with any form of character string. For instance, in the following sequence

```
CHARACTER UTL*8, WRD*5, P*2, VR*8, NM*7
WRD = 'PLACE'
P = 'AT'
UTL = WRD // 'M' // P
```

FORTRAN 77:

1. Takes the five characters PLACE from WRD and attaches an M at the end to form PLACEM. This string is stored someplace (call it T1).
2. The two characters AT are copied from P and attached to T1, producing PLACEMAT. This is stored someplace (call it T2).
3. The eight characters from T2 are copied into (i.e., assigned to) UTL.

As we can see, it is possible to build some long strings by concatenating on and on, into the sunset.

Since a substring is a legitimate term in a character expression, it can be used as an ingredient to be concatenated. Using the declarations and assignments from the previous four statements, we shall add another statement and analyze what happens:

```
VR = WRD(1:4) // P // WRD(5:5) // 'D'
```

The process is orderly enough:

1. WRD's first four characters (PLAC) are concatenated with AT (from the variable P) to form PLACAT. This intermediate value is stored someplace (T1).
2. The string grows as the E from WRD's fifth character is tacked onto it. PLACATE, the new string thus formed, is stored in another temporary place (T2).
3. The final growth occurs by extending the string from T2 with a D, and the result (PLACATED) is held somewhere (T3).
4. T3 is assigned to VR.

One more example using WRD = 'PLACE' will illustrate the use of the same string more than once in an expression:

```
NM = WRD(2:3) // WRD
```

This produces a value of LAPLACE in NM. The only restriction FORTRAN places on such usage is that overlapping substrings cannot appear on both sides of the same assignment statement. For example, suppose we have a 5-character string named WRD5 and we wanted to copy the characters from its third, fourth, and fifth positions into its second, third, and fourth positions. We *cannot* do it by saying

```
WRD5(2:4) = WRD5(3:5)
```

Instead, we have to get at the same result indirectly. A simple technique is to reserve a separate variable (we shall call it CHR3) of length 3 and use that for the transfer:

```
CHR3 = WRD5(3:5)
WRD5(2:4) = CHR3
```

Concatenation is the only explicit character string operator in the language. However, we should recognize that when we identify a substring, we are implying another type of operation: Part of a designated character string is being extracted and isolated for further processing. Consequently, between these two activities, we have a surprisingly powerful set of tools for building character strings, taking them apart, and putting them back together in different ways. Once we examine the kinds of decisions we can make with character strings, as we shall do in the next section, we shall put these capabilities to work on some more interesting problems.

## 8.3 DECISION PROCESSES AND CHARACTER STRINGS

The basic decision mechanism for character strings is the same as it is for numerical data: We perform a test and select a particular activity based on the outcome. Moreover, the test consists of a comparison between two expressions in which the type of comparison is described by a relational operator.

Such tests, when applied to character strings, make perfect sense for certain comparisons. To illustrate, the following statements surely present no serious problem to us:

```
CHARACTER TRDMK*6
READ *, TRDMK
PRINT *, TRDMK
IF (TRDMK(1:1) .EQ. 'S') THEN
    TRDMK(1:1) = ' '
ELSE
    TRDMK(1:1) = '$'
END IF
PRINT *, TRDMK
```

In this little sequence we read a 6-character input value into TRDMK and display it. Then we perform the following decision operation: If the first character of TRDMK is an S, we replace it with a blank; if not, we replace it with a dollar sign. After making the appropriate replacement, we display the new value in TRDMK.

### 8.3.1. Comparisons Between Character Strings

Comparing character strings seems like a perfectly reasonable thing to do and, expectedly, it is. By the same token, it makes equally good sense to be able to compare two character strings to see if they are *not* the same. Thus, the .NE. relational operator also is useful for comparing two character strings. (We should point out, though it may seem obvious, that when we talk about comparing "character strings" we include anything that produces a character string, i.e., a character expression.) However, when it comes to FORTRAN's other relational operations (such as .LT. and .GE.), things appear to make less sense. What does it mean to test whether the string GRAND is "less than" the string HUGE? And if we can figure out what it means, why would we want to know? Why indeed? It's funny you should ask.

Use of relational operations for comparing character strings with each other is based on the fact that each type of character is represented in the processor by its own numerical code. The collection of such codes is called the *collating sequence* to indicate that the numerical values, built into the machine's circuitry, are assigned to the various characters in a systematic order. For many types of processors, the collating sequence is set up as

follows:

1. The number representing the letter 'A' is smaller than that representing 'B', whose numerical code is smaller than that for 'C', and so on.
2. The number representing the blank character 'b' is smaller than that for the letter 'A'.

Thus, a comparison between two characters translates, in the machine's terms, to a comparison between the corresponding numerical codes. To us this means, for example, that when a character C1 is "less than" some other character C2, it is like saying that C1 is alphabetically before C2 (in the same sense that the letter M is before the letter R).

Appendix B tabulates the collating sequence for HP systems. The exact same principle applies to strings of characters. When instructed to compare two strings, FORTRAN 77 works from left to right. For example, the test

```
('CK' .LT. 'CR')
```

would turn out to be .TRUE. because FORTRAN, after comparing 'C' against 'C' and finding them to be equal, would go on and compare 'K' against 'R'. Since K is "less than" R, that establishes the entire string 'CK' to be less than 'CR' and FORTRAN marks the result "true." Similarly, the test

```
('DAX' .GT. 'CAP')
```

produces a result of .TRUE. because FORTRAN, having found the 'D' from 'DAX' to be greater than the 'C' from 'CAP', needs to look no further. It is clear that 'DAX' is alphabetically "after" 'CAP'. Once the relationship between the initial characters is established, of course, it does not matter how the remaining characters compare with each other.

Since the numerical code assigned to the blank places that character below the digits and letters in the collating sequence, this provides an easy way to take care of comparisons between character strings of unequal length. When directed to perform such a test, FORTRAN pads the shorter string with blanks (on the right) to produce a temporary string having the same length as the other one. For instance, the test

```
('CAP' .GE. 'CAPTAIN')
```

will turn out to be a test between the two 7-character strings 'CAPbbbb' and 'CAPTAIN'. Since blanks are lower than letters in the collating sequence, 'CAPTAIN' is greater than (i.e., alphabetically after) 'CAP' and, therefore, the outcome of the test will be false. Additional examples are shown in Table 8.2.

**Table 8.2** Decisions with Character Strings

```
P = 'CARNIVALS'; E='CARNIVORE'
```

| *Test Condition* | *Outcome When Internal Code is ASCII** | *Outcome When Internal Code is EBCDIC*** |
|---|---|---|
| (P .EQ. E) | .FALSE. | .FALSE. |
| (P .LT. E) | .TRUE. | .TRUE. |
| (P .LE. E) | .TRUE. | .TRUE. |
| (P(1:6) .EQ. E(1:6)) | .TRUE. | .TRUE. |
| ('34' .LT. '35') | .TRUE. | .TRUE. |
| ('E34' .LT. 'E35') | .TRUE. | .TRUE. |
| ('WZ' .LT. 'W4') | .FALSE. | .TRUE. |
| ('AB' .GT. '28') | .TRUE. | .FALSE. |
| (P(2:4)//'33' .LE. E(2:4)//'AA') | .TRUE. | .FALSE. |

* American Standard Code for Information Interchange; see Appendix B.
** Extended Binary Coded Decimal Interchange Code; see Appendix B.

FORTRAN 77 also provides four separate built-in functions for performing such comparisons. In each case, the function compares two character strings (which may be constants, variables, or expressions) in accordance with the rule implied by the particular function. If CHR_STR1 and CHR_STR2 are the two character strings being compared, we can characterize the functions' behavior as follows:

1. The expression

   LLT (CHR_STR1, CHR_STR2)

   produces a value of .TRUE. if CHR_STR1 precedes CHR_STR2 in the collating sequence.
2. The expression

   LLE (CHR_STR1, CHR_STR2)

   produces a value of .TRUE. if CHR_STR1 equals CHR_STR2 or if CHR_STR1 precedes CHR_STR2 in the collating sequence.
3. The expression

   LGT (CHR_STR1, CHR_STR2)

   produces a value of .TRUE. if CHR_STR1 follows CHR_STR2 in the collating sequence.
4. The expression

   LGE (CHR_STR1, CHR_STR2)

   produces a value of .TRUE. if CHR_STR1 equals CHR_STR2 or if CHR_STR1 follows CHR_STR2 in the collating sequence.

**Example 8.1** We shall apply our newly acquired character handling powers to an important piece of research in paleoanthropology—an exciting investigation to establish a link between people of today and the ancient Minoans. Recent evidence hints that the Minoans, an unusually tall people, favored the use of the letters O–Z in their surnames. Newly unearthed trade and religious documents containing long lists of names and transactions show that a good Minoan surname contained at least four letters in the 'O' to 'Z' part of the alphabet.

Scientists have long suspected that descendants of the Minoans, after working their way around the Fertile Crescent and across the forbidden Hills of Endless Sneezings, settled in parts of Idaho and built a life there. To find out more about this possibility, a crack team of researchers went into Idaho and gathered sample data, writing down names (up to 15 letters long) and heights (in centimeters, to the nearest tenth of a centimeter). Now what is needed is a FORTRAN 77 program that separates the data into two groups based on the names: Suspected Minoans (with at least four letters between O and Z), and suspected non-Minoans. The program is to report the number of people in each group, as well as the average heights.

We can develop this program systematically by starting with a bare outline (Figure 8.5) that shows the basic components without giving any details about how they will be expressed:

1. A set of declarations and initializations;
2. A loop that reads and processes each line;
3. An output component that reports the findings.

(These components are shown in Figure 8.5.)

Let us deal with component 1 first. The requirements of the problem give us a direct indication of the variables we shall need:

1. A 15-character string for the last name (LSTNAM);
2. A real variable for the height (HT);
3. Counters (integer variables) for the number of suspected Minoans (NUMMIN) and the number of suspected non-Minoans (NUMOTH). These need to be initialized to zero;

"Declare and initialize variables for
names, heights, counters."
**while** there are input data:
"Process the current input card."
**endwhile**
"Print the summary report."
"Stop."

**FIGURE 8.5 (a)** Basic Pseudocode Representation of the Program for Example 8.1.

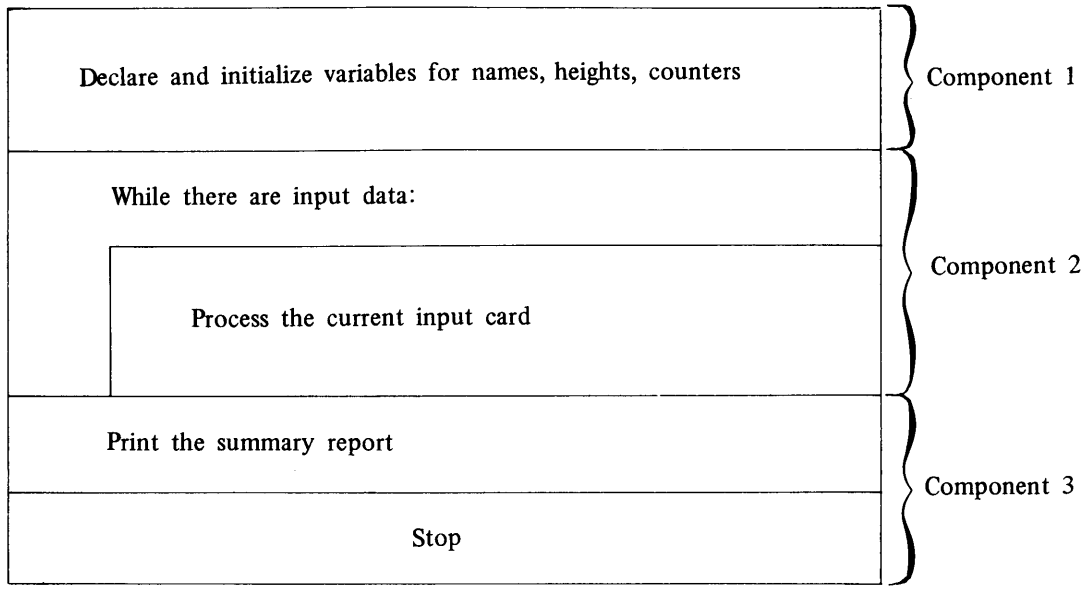



**FIGURE 8.5 (b)** Basic Description of the Program for Example 8.1: N-S Diagram.

4. Real variables for each of the average heights (AVGHTM, AVGHTO);
5. Since each average is computed as a sum divided by a count, we shall need a real variable in which to accumulate the sum of the heights for each group (SUMHTM, SUMHTO);
6. Finally, since we shall have to count the number of letters between O and Z in each name, a counter (NUMLET) will be needed for that purpose. As part of that process, we shall be looking at each of LSTNAM's letters. A DO-variable (I) will help keep track of that activity.

This gives us a more precise picture of the program's first component, and we can represent it in more detail, as shown in Figure 8.6.

Component 3 can be examined next. Since the required results include the average heights for the two groups, we know that this activity cannot begin until all the input has been read and processed. Once we are assured that this has happened, we can compute AVGHTM and AVGHTO from SUNHTM and NUMHTM, and from SUMHTO and NUMHTO, respectively. This is reflected in the more detailed pseudocode and diagram for component 3 (Figure 8.7).

This leaves us with the heart of the program to consider. Our original view of the program (Figure 8.5) represented component 2 as a DO-WHILE structure in which each cycle performed all the necessary processing for a single input line. Since this structure starts (by definition) with a test to determine whether the loop is to be entered, we need something on which to base this test. Consequently, just before the program enters this component, there must be a set of input data in storage, all ready to be examined. This tells us that component 1 needs to read the first set of input values as part of the initialization process. (Accordingly, this will be added to component 1.)

Having provided a safe and secure entry to component 2, we can use the requirements to help describe the component's activities in more detail. The loop itself, then, includes:

1. A test to determine whether there is more input to process. We shall use a height of 0.0 as an end-of-data signal. This not only controls the use of the loop; it also makes sure that component 3 is entered only at the proper time, i.e., when all input has been processed;

"Declare real variables HT, SUMHTM, SUMHTO, AVGHTM, AVGHTO,
integer variables NUMMIN, NUMOTH, NUMLET, I,
15-character string LSTNAM."
"Initialize NUMMIN, NUMOTH, SUMHTM, SUMHTO, to zero."

**FIGURE 8.6 (a)** Expansion of Component 3 from Figure 8.5 (a).

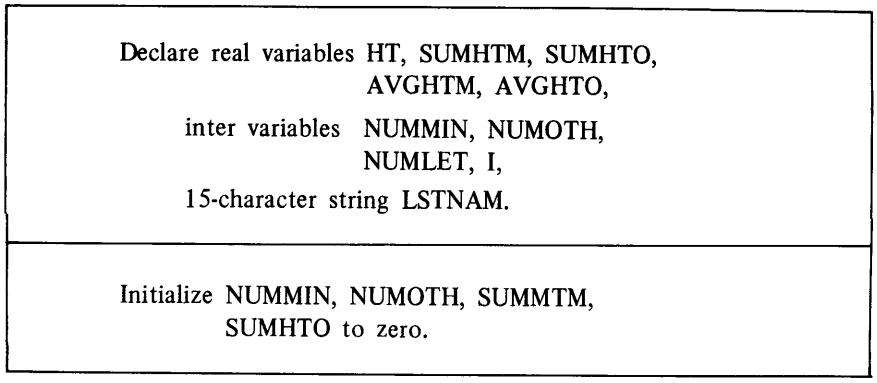



**FIGURE 8.6 (b)** Expansion of Component from Figure 8.5 (b): N-S Diagram.

"Compute AVGHTM <-- SUMHTM/NUMMIN."
"Compute AVHGTO <-- SUMHTO/NUMOTH."
"Print NUMMIN, NUMOTH, AVGHTM, AVGHTO."
"Stop."

**FIGURE 8.7 (a)** Expansion of Component 3 from Figure 8.5 (a).

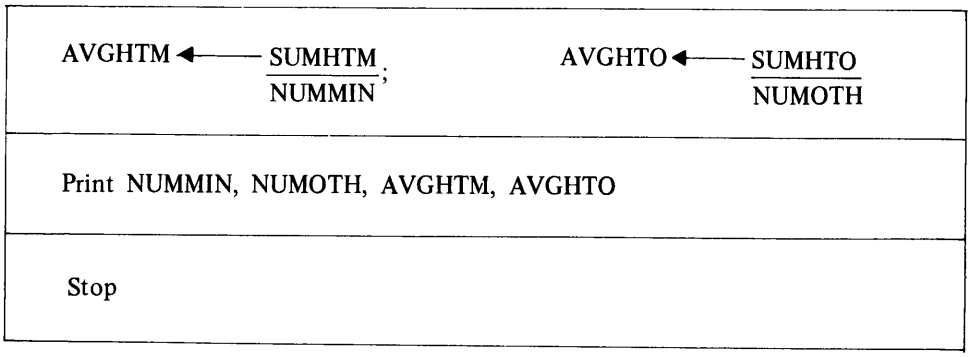



**FIGURE 8.7 (b)** Expansion of Component 3 from Figure 8.5 (b): N-S Diagram.

2. A letter-by-letter examination of LSTNAM to find out how many letters between O and Z there are in that name;
3. Addition of this input information to that being accumulated for suspected Minoans (or for suspected non-Minoans), based on the results obtained in (2) above;
4. Input of the next set of values in preparation for another (potential) cycle through the loop.

As a result of this analysis, we can expand our description of component 2 so that it gives more detail (Figure 8.8).

Now we can work with the description in Figure 8.8, concentrating this time on how these activities will be specified:

1. The first activity is straightforward enough and needs no further explanation;
2. The letter-by-letter examination immediately suggests a DO-loop in which the location of a letter from O through Z is recorded by adding 1 to NUMLET. (To make sure this works properly, we remind ourselves to initialize NUMLET before the loop is started.) The loop uses the index (I) to step through LSTNAM's letters. For example, when I is 6 (the sixth time through the loop for that name), we are looking at LSTNAM(6:6), the sixth position in LSTNAM;
3. NUMLET now can be used to identify the proper group for the set of data. If NUMLET is at least 4, then NUMMIN needs to be incremented, HT needs to be added to SUMHIM, and an appropriate message produced. If NUMLET is less than 4, corresponding activities are to be performed for NUMOTH and SUMHTO;

```
while HT is not equal to zero:
    "Determine how many letters between O and Z
        there are in LSTNAM."
    if
        LSTNAM indicates a suspected Minoan:
    then
        "Update NUMMIN and SUMHTM."
        "Print input with Minoan message."
    else
        "Update NUMOTH and SUMHTO."
        "Print input with non-Minoan message."
    endif
    "Read the next input card."
endwhile
```

**FIGURE 8.8 (a)** Expansion of Component 3 from Figure 8.5 (a).

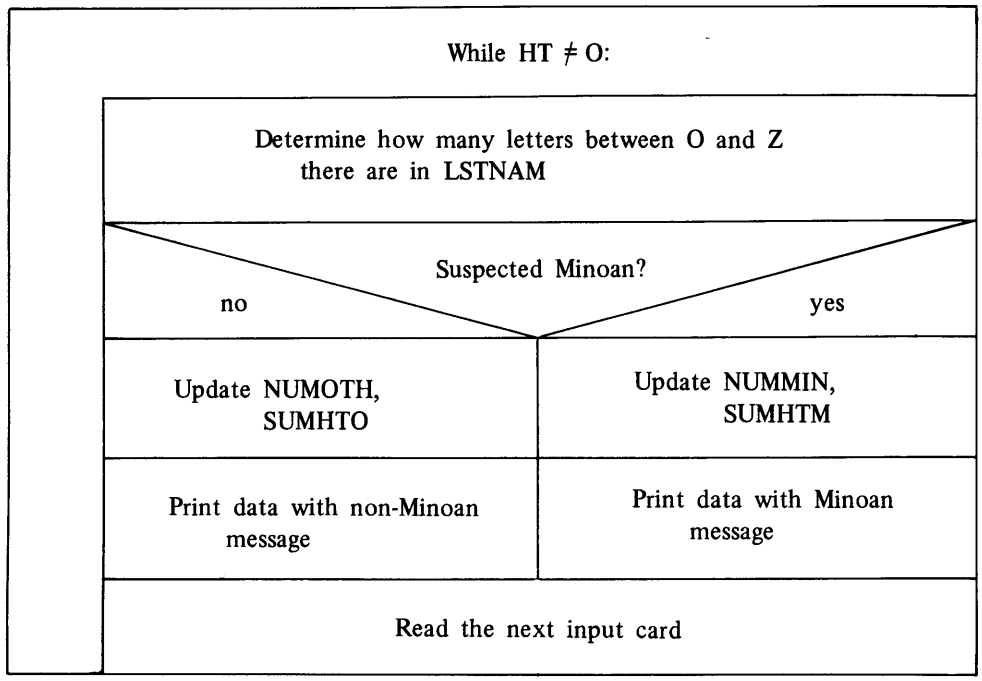



**FIGURE 8.8 (b)** Expansion of Component 2 from Figure 8.5 (b): N-S Diagram.

4. The final activity is straightforward.

This additional detail is shown in Figure 8.9, which also includes the revision for component 1 indicated earlier.

At this point, it is clear that our understanding of the program's requirements (and how to meet them) is sufficiently well developed so that writing the actual statements becomes almost automatic. We can see that this is not exaggerated by examining the statements in Figure 8.10. This is one of the major ideas behind this kind of systematic development, and it works no matter how complicated the problem might be.

Sample input for the program is given in Figure 8.11 and the results are seen in Figure 8.12.

### 8.3.2 Searching for Character Strings— The INDEX Function

A process often used in work with character strings is one in which a string is searched to see if it contains some other one. The FORTRAN compiler is an immediate example of this: When it analyzes our programs, a considerable part of its time is spent in searching our statements, which it treats as character strings, to find certain words and other

```
"Declare real variables HT, SUMHTM, SUMHTO, AVGHTM, AVGHTO,
        integer variables NUMMIN, NUMOTH, NUMLET, I
    15-character string LSTNAM."
"Initialize NUMMIN, NUMOTH, SUMHTM, SUMHTO to zero."
"Read the first input card."
while HT is not equal to zero:
    "set NUMLET to zero."
    do for I = 1 to 15:
        if
            the Ith letter in LSTNAM is between O and Z:
        then
        "Add 1 to NUMLET."
        else
        endif
    enddo
    if
        NUMLET is equal to or greater than 4:
    then
        "Add 1 to NUMMIN and add HT to SUMHTM."
        "Print input with Minoan message."
    else
        "Add 1 to NUMOTH and add HT to SUMHTO."
        "Print input with non-Minoan message."
    endif
    "Read the next input card."
endwhile
"Compute AVGHTM <-- SUMHTM/NUMMIN."
"Compute AVGHTO <-- SUMHTO/NUMOTH."
Print NUMMIN, NUMOTH, AVGHTM, AVGHTO."
"Stop."
```

**FIGURE 8.9 (a)** Final Expansion of Program Description for Example 8.1.

symbols that will help determine what it will do next. In a way, that is what we were doing in Example 8.1, where we examined one character at a time. FORTRAN 77's INDEX built-in function gives us a more general facility that allows convenient searches for strings of any length. This function has the form

```
INDEX(arg1, arg2)
```

where `arg1` is the string to be examined and `arg2` is the string we are trying to match. For example, the expression

```
INDEX ('KANKAKEE', 'AK')
```

is set up to search the character constant KANKAKEE for an occurrence of the constant AK. INDEX returns an integer whose value describes the outcome of the search. If there is no match, the value is zero; on the other hand, if the search is successful, the integer shows at which position the match begins. We can see how this works by using the previous expression in an assignment statement:

```
N = INDEX('KANKAKEE', 'AK')
```

N will have a value of 5 in it because the string AK, in fact, is found in KANKAKEE, starting at position 5.

INDEX finds only a single occurrence, regardless of how many times the smaller string may occur in the larger one. INDEX searches from left to right, stopping after it finds the first match. Consequently, it takes a little programming to arrange for a string to be searched for all matches with another string.

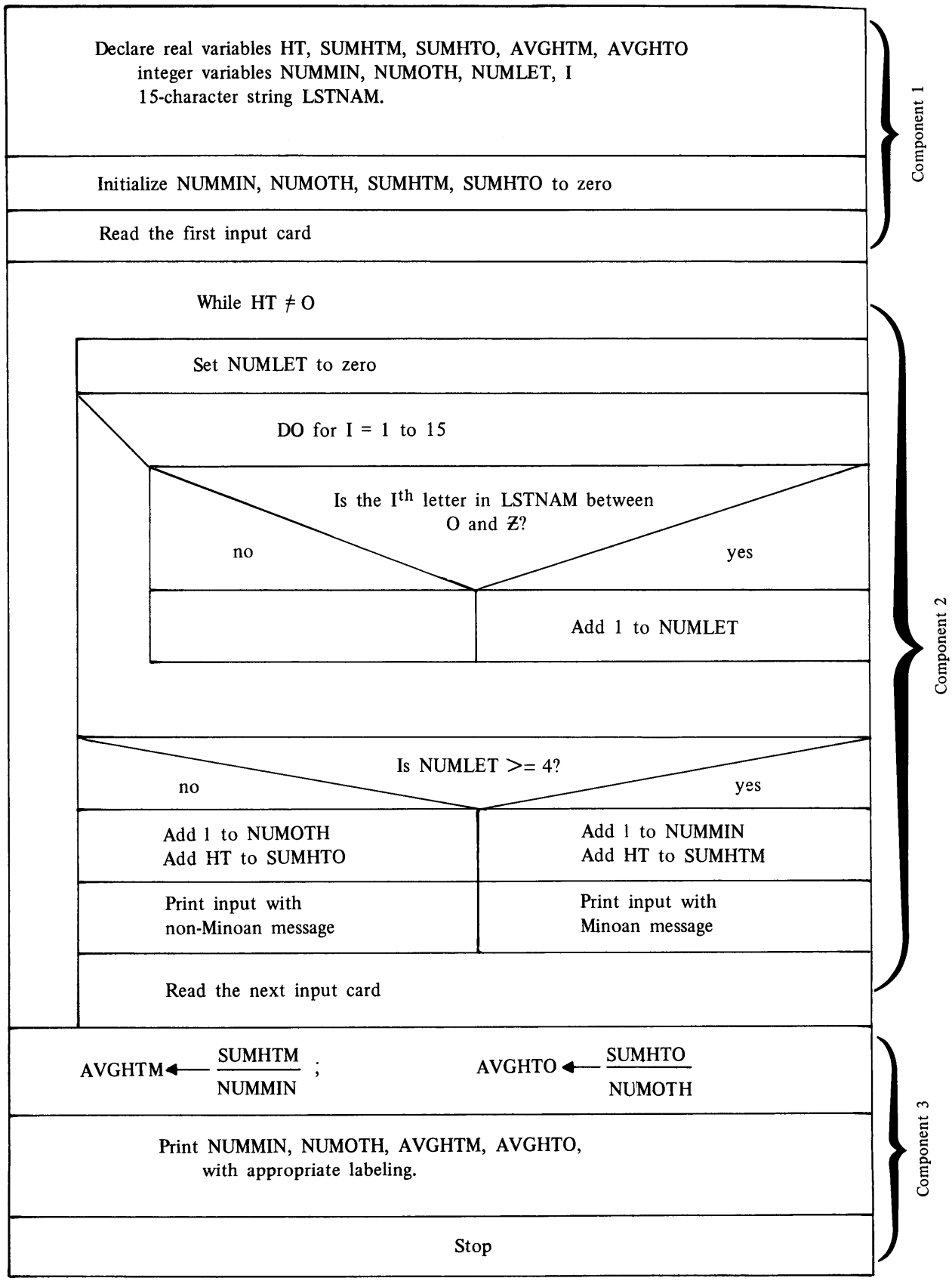



**FIGURE 8.9 (b)** Final Expansion of Program Description for Example 8.1: N-S Diagram.

```
C*************************************************************
C                        EXAMPLE 8.1                          *
C*************************************************************
      PROGRAM       EX801
      REAL          SUMHTM,SUMHTO,AVGHTM,AVGHTO,HT
      INTEGER       NUMMIN,NUMOTH,NUMLET,I
      CHARACTER     LSTNAM*15
C
      SUMHTM = 0.
      SUMTHO = 0.
      NUMMIN = 0
      NUMOTH = 0
      PRINT *, 'ENTER VALUES FOR LSTNAM AND HT'
      READ *, LSTNAM,HT
C
      DO WHILE (HT .NE. 0)
C
C*************************************************************
C  THIS LOOP EXAMINES EACH OF LSTNAM'S 15 POSITIONS TO SEE  *
C  WHETHER IT CONTAINS A LETTER FROM O-Z. WE DO THIS BY DE- *
C  TERMINING WHETHER THE LETTER IN THAT POSITION IS GREA-   *
C  TER THAN OR EQUAL TO 'O'. THE ASSUMPTION IS THAT THERE   *
C  ARE NO OTHER CHARACTERS IN LSTNAM EXCEPT LETTERS OR      *
C  BLANKS. IF WE DID NOT ASSUME THAT, WE ALSO WOULD TEST FOR*
C  THE CHARACTER BEING LESS THAN OR EQUAL TO 'Z'.           *
C*************************************************************
         NUMLET = 0
C
         DO 20 I=1,15
            IF (LSTNAM(I:I) .GE. 'O') NUMLET=NUMLET+1
   20    CONTINUE
C
         IF (NUMLET .GE. 4) THEN
            NUMMIN = NUMMIN+1
            SUMHTM = SUMHTM+HT
            PRINT *, LSTNAM,HT,' POSSIBLE MINOAN'
         ELSE
            NUMOTH = NUMOTH+1
            SUMHTO = SUMHTO+HT
            PRINT *, LSTNAM,HT, 'POSSIBLE NON-MINOAN'
         END IF
C
         PRINT *, 'ENTER ANOTHER SET OF VALUES FOR LSTNAM AND HT'
         READ *, LSTNAM, HT
      END DO
C
      AVGHTM = SUMHTM/NUMMIN
      AVGHTO = SUMHTO/NUMOTH
      PRINT *, 'NO. OF POSSIBLE MINOANS: ',NUMMIN
      PRINT *, 'NO. OF POSSIBLE NON-MINOANS: ',NUMOTH
      PRINT *, 'AVG. HT. OF POSSIBLE MINOANS: ',AVGHTM
      PRINT *, 'AVG. HT. OF POSSIBLE NON-MINOANS: ',AVGHTO
      STOP
      END
```



**FIGURE 8.10** FORTRAN Statements for Example 8.1.

```
ROSOSTOMOS          183.6
AKLAMANSA           151.3
COLAMBOLANA         168.0
KUSUTUVOS           168.2
BIDIBIM             157.4
DLAZOVA             154.6
ABDABILAD           156.5
VUTOVSYZY           179.9
FILADKY             161.1
KROTKO              177.7
BLEEBIK             159.8
```

**FIGURE 8.11** Input Data for Example 8.1.

```
ROSOSTOMOS          183.6     POSSIBLE MINOAN
AKLAMANSA           151.3     POSSIBLE NON-MINOAN
COLAMBOLANA         168.0     POSSIBLE NON-MINOAN
KUSUTUVOS           168.2     POSSIBLE MINOAN
BIDIBIM             157.4     POSSIBLE NON-MINOAN
DLAZOVA             154.6     POSSIBLE NON-MINOAN
ABDABILAD           156.5     POSSIBLE NON-MINOAN
VUTOVSYZY           179.9     POSSIBLE MINOAN
FILADKY             161.1     POSSIBLE NON-MINOAN
KROTKO              177.7     POSSIBLE MINOAN
BLEEBIK             159.8     POSSIBLE NON-MINOAN
NO. OF POSSIBLE MINOANS:        4
NO. OF POSSIBLE NON-MINOANS:    7
AVG. HT. OF POSSIBLE MINOANS:       0.1823500E 03
AVG. HT. OF POSSIBLE NON-MINOANS:     0.1583857E 03
```

**FIGURE 8.12** Output for Example 8.1.

**Example 8.2** These are exciting times for believers in Eastern gurus and their spiritual prowess. Certain researchers, claiming to be on the verge of a breakthrough, feel that the concentration of power is linked, somehow, with a guru's name. There is great significance, they say, in the number of appearances of the letter-group ANA and in the placement of these appearances. A guru named DANAA, for instance, is no big deal in the magic department (brief local rainfalls, etc.). On the other hand, somebody like the Swami MENANANDAPRASHANANARANANA would be a major force to be reckoned with.

Since the exact relationship still is not known, the researchers need a FORTRAN 77 program to develop crucial data for their continued work: For each guru's name submitted as input, the program is to produce a table showing the starting position of each occurrence of ANA. Each table is preceded by the name and the number of occurrences.

As expected, the program is built around a search of each name, and the search is built around the INDEX function. However, we must get around the fact that INDEX will keep finding the first (leftmost) occurrence unless we avoid repeated examinations of the same part of the name. For example, in examining the name BANDANANARANJANAN, INDEX will find an occurrence of ANA starting in position 5:

```
BAND|ANA|NARANJANAN
     ↑
     └──── Position 5
```

In order to allow INDEX to find the next match (which happens to start in position 7), we want to search only that part (substring) of the name that has not been searched yet:

```
BANDAN|ANA|RANJANAN
       ↑
       └──── Position 7
```

Accordingly, after finding a match at position 5, we want to look at the rest of the name, starting in position 6. Now, the leftmost match for that string begins in the second position of the shortened string, which corresponds to position 7 of the entire name:

```
BANDA|NANARANJANAN
     ↑
     └──Position 6
```

Then, if we want to look further, we need to start at position 8:

```
BANDANA|NARANJANAN
```

The next (and last) match occurs at position 7 from the beginning of the new substring, or position 14 from the beginning of the entire string:

```
BANDANA|NARANJANAN
```

Now, let us restate this in terms of FORTRAN statements: if GURU is the character variable containing the name, and POSN is the position of a match, then our first search would say

```
POSN = INDEX (GURU, 'ANA')
```

As seen before, this would produce a value of 5 in POSN. Now, suppose we stored that away someplace, and we are ready to continue the search. Regardless of how we manage it, we want the next search to do the following:

```
POSN = INDEX (GURU(6:), 'ANA')
```

In other words, "please search the characters in that part of GURU starting with position 6 and going out to the end of the string, if necessary, and report the result in POSN." For our medium-powered guru BANDANANARANJANAN, this would produce a value of 2 in POSN, since the match begins at the second position of GURU(6:). In order to report this as a position relative to the beginning of the entire string, we must adjust the value in POSN by adding 5, the starting position of the first match. Once this is done, the new value (7) shows the actual point at which the second match starts.

Now, we can generalize this process by observing the following:

1. Whenever we search a substring successfully (i.e., we find a match), the position at which that match begins (with the beginning of that substring being position 1) can be adjusted to the position from the beginning of the entire string by adding the location of the previous match. Thus, if POSN is the position of the match just found, and MARK shows the beginning of the previous match, the adjusted position is obtained with a statement like

```
POSN = POSN+MARK
```

2. After a successful search, the starting position of the next search can be set by adding 1 to the adjusted position of the successful search. For instance, if START is the starting position of the reduced substring about to be searched, and we have already adjusted POSN by adding MARK to it, then START is reset by the statement

```
START = POSN+1
```

Since the first search covers the entire string (i.e., from position 1 on), our "substring" is really GURU(1:). Consequently, that tells us that START needs to be initialized to 1. Moreover, since it *is* the first search, we cannot adjust for a previous one, so that MARK requires initialization to zero.

That leaves us with the job of storing the search information someplace, as we generate it. The easiest thing to do is to set up an array of integers where each element specifies the beginning position of a match. Since we cannot know how many matches a particular name will produce, we shall allow enough room to handle even the most powerful guru. Our crack researchers are convinced that anything beyond ten matches is out of the question. Being a little on the conservative side, we shall allow for twelve, which is the largest possible number of matches in a 25-character name. (We shall leave it up to you to figure out what that highly powerful name must be in order to have twelve occurrences.) Initially, all twelve elements will be set to zero before we look at each name. Then, as the search progresses, the zeros will systematically be replaced with integers indicating the position of each match. An indicator, which we shall call LATEST, will keep track of the array element just filled, so that we avoid storing information in a

place already containing a search result. Here again, LATEST will be initialized to zero before we look at a new name, thereby indicating that the array (which we shall call PSNTBL) is empty.

Now we have defined all the major ingredients and we can step through the processing described by the statements in Figure 8.13. We shall use our good friend Swami BANDANANARANJANAN (the first input name from Figure 8.14).

```
C*************************************************************
C                        EXAMPLE 8.2                          *
C*************************************************************
C  GURU IS A 25-CHARACTER STRING CONTAINING AN INPUT NAME;    *
C  PSNTBL IS AN ARRAY WHICH WILL BE FILLED WITH THE LOCA-     *
C  TIONS OF EACH MATCH FOUND IN A GIVEN NAME;                 *
C  START IS AN INDICATOR THAT DEFINES THE STARTING POINT FOR*
C  EACH SEARCH THROUGH AN INPUT NAME;                         *
C  MARK INDICATES THE LOCATION OF THE MOST RECENTLY FOUND     *
C  MATCH AND IS USED TO ADJUST THE POSITION COMPUTED BY THE  *
C  INDEX FUNCTION;                                            *
C  POSN IS THE LOCATION OF THE MATCH FOUND BY THE INDEX       *
C  FUNCTION (OR 0 IF A MATCH IS NOT FOUND);                   *
C  NUMLET COUNTS THE NUMBER OF MATCHES FOUND IN A NAME.       *
C*************************************************************
      PROGRAM         EX802
      IMPLICIT        NONE
      INTEGER*2       PSNTBL(15),START,MARK,POSN,NUMLET,I
      CHARACTER*25    GURU
      CHARACTER*7     BLANK
      PARAMETER       (BLANK='       ')
      DO I=1,15
         PSNTBL(I) = 0
      END DO
      PRINT *, 'ENTER THE FIRST VALUE FOR GURU'
      READ *, GURU

      DO WHILE (GURU .NE. BLANK)
         START = 1
         MARK = 0
         POSN = 0
         NUMLET = 0
         POSN = INDEX(GURU(START:),'ANA')
         DO WHILE (POSN .NE. 0)
            NUMLET = NUMLET+1
            POSN = POSN+MARK
            PSNTBL(NUMLET) = POSN
            MARK = POSN
            START = START+1
            POSN = INDEX(GURU(START:),'ANA')
         END DO

         PRINT *, BLANK
         PRINT *, 'SWAMI ',GURU,' HAS ',NUMLET,' OCCURRENCES OF ANA'
         I = 1

         DO WHILE (PSNTBL(I) .NE. 0)
            PRINT *, 'MATCH ',I,' STARTS AT POSITION ',PSNTBL(I)
            I = I+1
         END DO

         PRINT *, BLANK
         PRINT *, 'ENTER THE NEXT VALUE FOR GURU'
         READ (*,END=99) GURU
      END DO

   99 PRINT *, BLANK
      PRINT *, 'END OF RUN'
      STOP
      END
```

**FIGURE 8.13** FORTRAN Statements for Example 8.2.

```
'BANDANANARANJANAN'
'JANHA'
'RANHA'
'RAMANANASHANANATANATANANA'
'HARANA'
'       '
```

**FIGURE 8.14** Input for Example 8.2.

```
SWAMI   RANDANANARANJANAN            HAS   3   OCCURRENCES OF ANA
MATCH   1     STARTS AT POSITION       3
MATCH   2     STARTS AT POSITION       7
MATCH   3     STARTS AT POSITION       14

SWAMI   JANHA                        HAS   0   OCCURRENCES OF ANA

SWAMI   RAMANANASHANANATANATANANA    HAS   7   OCCURRENCES OF ANA
MATCH   1     STARTS AT POSITION       4
MATCH   2     STARTS AT POSITION       6
MATCH   3     STARTS AT POSITION       11
MATCH   4     STARTS AT POSITION       13
MATCH   5     STARTS AT POSITION       17
MATCH   6     STARTS AT POSITION       21
MATCH   7     STARTS AT POSITION       23

SWAMI   HARANA                      HAS   1   OCCURRENCES OF ANA
MATCH   1     STARTS AT POSITION       4

END OF RUN
```

**FIGURE 8.15** Output for Example 8.2.

1. The first time we execute statement 10, the variable START has a value of 1, so that the string being searched by INDEX is, in fact, the entire name. There is a match starting at position 5, so that POSN is set to 5, the IF test passes, and the associated action is performed. Accordingly, LATEST is increased from 0 (its initial value) to 1; since MARK is at zero, its addition to POSN does not affect POSN's value this time. PSNTBL(1) is given a value of 5 (indicating that the first match starts at position 5), MARK is given a new value (5, the position of the most recently found match), and START, the leftmost position of the substring to be searched next, is set to 6 (i.e., 5+1), and we are ready to look for a second possible match.

2. The second use of statement 10 on this name limits the search to GURU(6:), i.e., the string NANARANJANAN. Another match is found, starting at the second position of this string. Thus, POSN will have a value of 2 as we leave statement 10. LATEST is increased from 1 to 2, and POSN is adjusted from 2 to 7 (by adding MARK's current value of 5 to it) to indicate that the second match starts at position 7 when we measure from the beginning of the entire string. MARK is brought up to date (so that it now is 7) after PSNTBL(2) is assigned a value of 7. Finally, START is set to 8 (i.e., one position after the starting point of the match just found), and we are ready to try a third search.

3. Statement 10 now searches GURU(8:), i.e., the string NARANJANAN. Once again the search succeeds and POSN receives a value of 7, indicating that the match begins in position 7 from the beginning of this substring. LATEST is increased from 2 to 3, showing that this is the third match. POSN is adjusted again by adding MARK's current value (7+7), so that the new value (14) gives the position of the third match from the very beginning of the original string. This value is stored in PSNTBL(3), MARK is set to 14, START is set to 15, and we try again.

4. The fourth attempt turns out to be the final one as far as this name is concerned. Statement 10 searches GURU(15:), a greatly reduced string consisting of the three letters NAN. Since there is no match, the IF test fails and the action is bypassed. As a result, only the first three entries in PSNTBL are filled in (with the rest having the zeros we put there during initialization). Incidentally, since the rest of the activity was bypassed when the IF test failed, LATEST still has a 3 in it, which happens to be the number of successful matches.

The results corresponding to Figure 8.14's input are shown in Figure 8.15.

### 8.3.3 Conversions Between Characters and Integer Values

FORTRAN includes two functions based on the idea that each character, when stored internally, has a unique numerical representation (i.e., a place in the collating sequence): The ICHAR function examines a single character and produces the corresponding position of that character in the collating sequence. For example,

```
ICHAR ('W')     and     ICHAR ('w')
```

produce respective integer values of 87 and 119. (Check these values against the information in Appendix B to make sure the relationship is clear.) The CHAR function operates in the opposite direction by producing the character corresponding to the integer value given to it. For instance, the sequence

```
CHARACTER*1     NEWLTR
NEWLTR = CHAR (80)
```

will store the character 'P' in NEWLTR. (Again, check Appendix B.)

## 8.4 ANOTHER EXAMPLE PROGRAM

Since the design and implementation of character-handling algorithms is likely to be less familiar than the idea of "computing" with numbers, we shall look at another, more ambitious example. This time, instead of dealing with input consisting of individual words or names, each entire input line will contain several words, and all the words on a line will be read as a single character string. There is no reason to expect the number of words on a line, or the lengths of these words, to be uniform. Other information about the input can be summarized as follows:

1. A word always will end on the same line on which it starts.
2. Words are separated by *at least* one blank.
3. Every input line will have *at least* one word.
4. Columns 79–80 will never be part of an input character string; those columns are to be reserved for a data line sequence number (in the same way that columns 73–80 on FORTRAN statement lines are reserved).
5. The longest word will be 12 letters.
6. While there may be any number of blanks between words, there never is a blank before the first word on a line.
7. The last word on a line always will be followed by at least one blank.
8. There are no periods (.) in the entire input.

A typical input line is shown in Figure 8.16.

The required program is to produce the following output:

1. A count of the total number of words in the input.

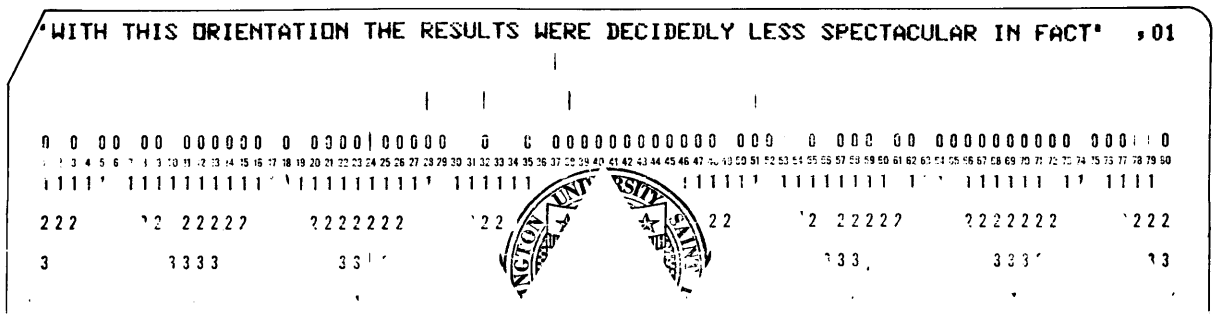



**FIGURE 8.16** Input Card Containing Several Words.

2. Counts of the number of words for each word length (1 to 12).
3. The average word length.

The crucial processing component in solving this problem is to separate each individual word from the rest of its input line so that we can determine its length and use that information for the statistics that we are required to develop. Before we look at the detailed operations needed to do this, we shall note that the job of processing a word can be made simpler by dealing with a copy of the word, once it has been identified, instead of processing the word as it sits inside the larger input string. Consequently, we shall use a 12-character string named WORD for this purpose. Originally, (i.e., before we start to look for a new word), WORD will be emptied (initialized to blanks), and an indicator named WRDPTR will be set to 1 to indicate that we are about to look for the first letter of a new word as we find it in the input string (called LINE) and rebuild it in WORD. Similarly, as we work our way through LINE, another indicator (called CRDPTR) will keep track of where we are on the line, so that we do not look at the same part more than once, and we do not miss anything either.

Once everything has been declared, the program's major activities divide fairly naturally into four sections:

1. An initialization section during which the array of counters (NWDS) is set to zeros, the first input line is read, and the indicators are set to begin building the first word.
2. A simple loop that builds an individual word (in WORD) by copying successive characters from the input line until a blank is found.
3. A process that handles the bookkeeping required to add information about another word to the statistics, and resets the program to get the next word which includes finding and bypassing blanks between words), bringing in another line as needed.
4. A final, one-time activity that computes the average word length and displays the results.

The first three activities are built within a DO-WHILE construct that allows the cycle to repeat as long as the concluding asterisk has not been reached. Figure 8.17 shows a flow diagram and pseudocode, and the program is given in Figure 8.18. Operation of the program is illustrated by the sample run in Figure 8.19.

## 8.5 SUMMARY

A character string may consist of any combination of letters, numerical digits, and special characters, including blanks. Character string variables are named like any other variables and are declared with a statement having the form

```
CHARACTER  name*length, name*length, etc.
```

where each length specifies the number of characters the particular string can store at one time. Arrays of character strings follow the same organizational rules that apply to other data types (as described in Chapter 7).

Part of a string (i.e., a *substring*) can be described by specifying the starting and ending positions of the string in which it is contained. For example,

```
SVAR(2:8)
```

identifies that part of the string SVAR extending from positions 2 through 8.

Assignment statements for character strings work the same way they do for numerical values: A character expression is evaluated and the result (a character string) is stored in (assigned to) a specified destination. Expressions may contain combinations of character

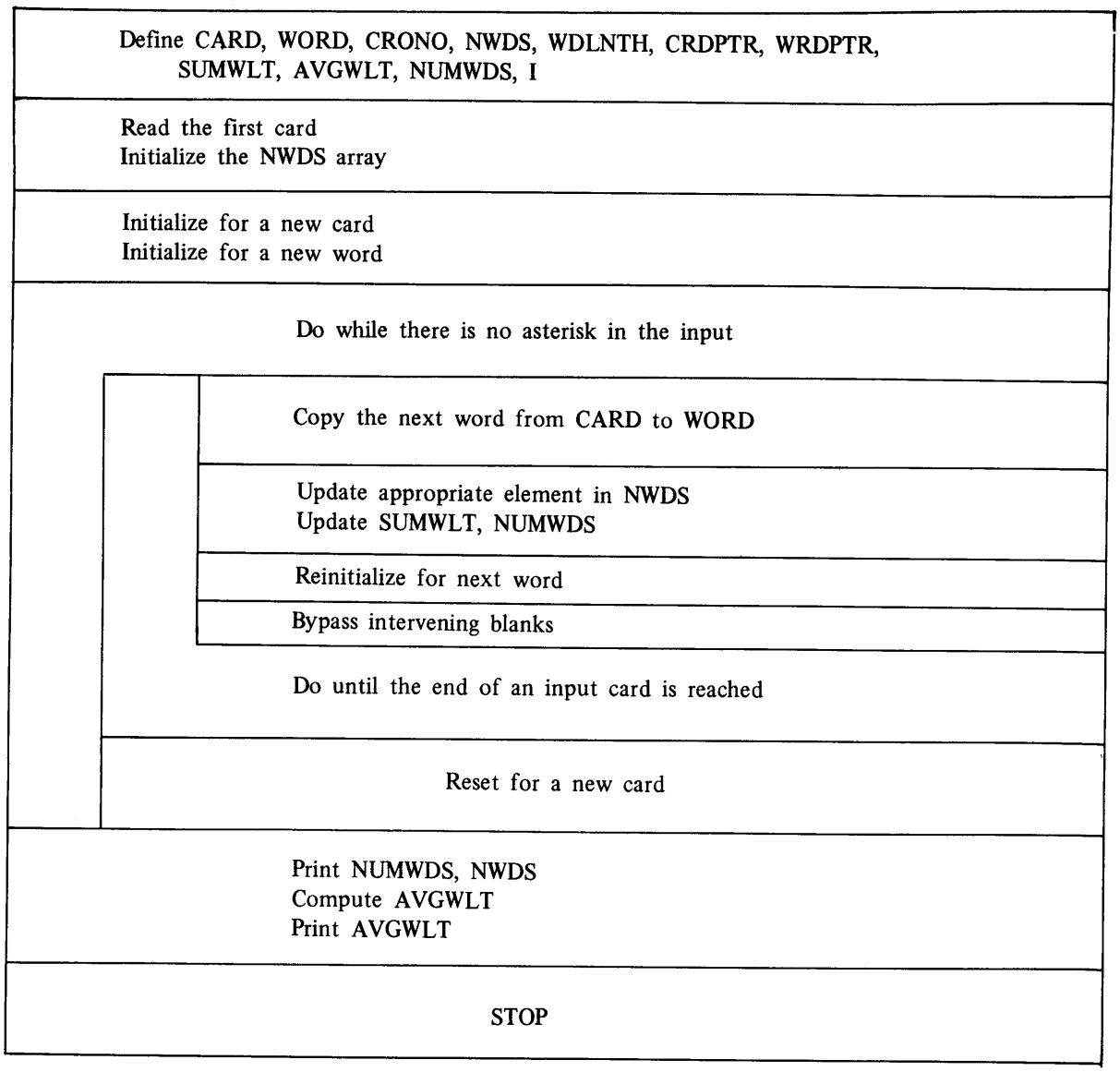



**FIGURE 8.17 (a)** Structured Flowchart for Example 8.3.

"Define CARD, WORD, CRDNO, NWDS, WDLNTH, CRDPTR,
WRDPTR, SUMWLT, AVGWLT, NUMWDS, I."
Read the first card."
Initialize the NWDS array, initialize for a new
card, and initialize for a new word."
**while** there is no asterisk in the input:
**do** until the end of an input card is reached:
"Copy the next word from CARD to WORD."
"Update the appropriate element in NWDS."
"Update SUMWLT, NUMWDS."
"Reinitialize for the next word."
"Bypass intervening blanks."
**enddo**
"Reset for a new card."
**endwhile**
"Print NUMWDS, NWDS."
"Compute and print AVGWLT."
"Stop."

**FIGURE 8.17 (b)** Pseudocode for Example 8.3.

```
C*************************************************************
C                          EXAMPLE 8.3                       *
C*************************************************************
C  LINE:       A 75-CHARACTER STRING FOR AN INPUT LINE       *
C  CDNO:       THE LINE NO., FROM COLS. 79-80 OF A LINE      *
C  WORD:       A 12-CHARACTER STRING FOR AN INPUT WORD       *
C  CRDPTR:     INDICATES THE CURRENT POSITION BEING EXAMINED*
C              IN LINE                                       *
C  WRDPTR:     INDICATES THE NEXT AVAILABLE POSITION IN WORD*
C  NUMWDS:     THE NUMBER OF WORDS IN THE INPUT              *
C  NWDS:       A 12-ELEMENT ARRAY FOR OCCURRENCES OF EACH    *
C              WORD LENGTH                                   *
C  WDLNTH:     THE LENGTH OF THE INPUT WORD BEING EXAMINED   *
C  SUMWLT:     TOTAL NUMBER OF LETTERS IN THE INPUT WORDS    *
C  AVGWLT:     THE AVERAGE WORD LENGTH                       *
C*************************************************************

      PROGRAM      EX803
      IMPLICIT     NONE
      INTEGER*2    CDNO,CRDPTR,WRDPTR,NUMWDS,NWDS(12),WDLNTH,I
      CHARACTER    LINE*75,WORD*12,BLANK*1,STAR*1
      REAL*4       SUMWLT,AVGWLT
      PARAMETER    (BLANK=' ',STAR='*')

      DO 1 I=1,12
         NWDS(I) = 0
    1 CONTINUE
      NUMWDS=0
      SUMWLT=0
      PRINT *, 'ENTER VALUES FOR LINE and CDNO'
      READ *, LINE,CDNO
      CRDPTR=1
      WRDPTR=1
      WDLNTH=0
      WORD=BLANK


      DO WHILE (LINE(CRDPTR:CRDPTR) .NE. STAR)
         WORD(WRDPTR:WRDPTR) = LINE(CRDPTR:CRDPTR)
         DO WHILE (LINE(CRDPTR:CRDPTR) .NE. BLANK)
            WDLNTH = WDLNTH+1
            WRDPTR = WRDPTR+1
            CRDPTR = CRDPTR+1
            WORD(WRDPTR:WRDPTR) = LINE(CRDPTR:CRDPTR)
         END DO

C----------UPDATE THE VARIOUS COUNTERS----------

         NUMWDS = NUMWDS+1
         NWDS(WDLNTH) = NWDS(WDLNTH)+1
         SUMWLT = SUMWLT+WDLNTH
         WDLNTH = 0
         WRDPTR = 1
         WORD = BLANK
```

**FIGURE 8.18** Statements for Example 8.3. *(continued)*

```
C----------BYPASS BLANKS AND CHECK FOR END OF LINE----------

         DO WHILE (LINE(CRDPTR:CRDPTR) .NE. BLANK)
            IF (CRDPTR .EQ. 75) THEN
               PRINT *, 'ENTER ANOTHER INPUT LINE AND CDNO VALUE'
               READ *, LINE, CDNO
               CRDPTR = 1
            ELSE
               CRDPTR = CRDPTR + 1
            END IF
         END DO
      END DO

C----------FINAL PROCESSING----------

      PRINT *, 'NUMBER OF WORDS THIS RUN: ',NUMWDS
      PRINT *, BLANK
      PRINT *, 'WORD DISTRIBUTION:'
      DO I=1,12
         PRINT *, 'THERE ARE ',NWDS(I),'WORDS WITH LENGTH ',I
      END DO
      PRINT *, BLANK
      AVGWLT = SUMWLT/NUMWDS
      PRINT *, 'AVG. WORD LENGTH: ',AVGWLT

      STOP
      END
```

**FIGURE 8.18** *(Continued)*

```
'WITH THIS ORIENTATION THE RESULTS WERE DECIDEDLY LESS SPECTACULAR IN FACT'  .01
'THE  BENEFITS     OF THIS PRACTICE VARIED WIDELY AMONG COMPUTER'            .02
'   CENTERS    THAT FOLLOWED IT SURVEYS OF SUCH USAGE REVEALED A           '.03
'SURPRISINGLY HIGH INCIDENCE OF CASES IN WHICH PROGRAM MODULES WERE CRAFTED '.04
'AS POTENTIAL LIBRARY COMPONENTS ONLY TO END UP BEING USED IN SINGLE'        .05
'APPLICATIONS THE RESULTING LIBRARIES WERE FILLED WITH A GREAT NUMBER OF'    .06
'SUBPROGRAMS THAT COLLECTED DUST ****                                       '.07
```

**FIGURE 8.19 (a)** Output from Sample Run of Program to Determine Average Word Length in a Sample: Total Words.

```
NUMBER OF WORDS THIS RUN:         68

WORD DISTRIBUTION:
THERE ARE      2     WORDS WITH LENGTH      1
THERE ARE     11     WORDS WITH LENGTH      2
THERE ARE      4     WORDS WITH LENGTH      3
THERE ARE     16     WORDS WITH LENGTH      4
THERE ARE      6     WORDS WITH LENGTH      5
THERE ARE      5     WORDS WITH LENGTH      6
THERE ARE      7     WORDS WITH LENGTH      7
THERE ARE      5     WORDS WITH LENGTH      8
THERE ARE      6     WORDS WITH LENGTH      9
THERE ARE      1     WORDS WITH LENGTH     10
THERE ARE      3     WORDS WITH LENGTH     11
THERE ARE      2     WORDS WITH LENGTH     12

AVG. WORD LENGTH:        0.5441176E 01
```

**FIG. 8.19 (b)** Output from Computation of Program to Determine Average Word Length in a Sample.

string constants, variables, and substrings connected to each other in any desired order by means of the *concatenation operator* //. Either a character string name or a substring may be specified as a destination. When the value of a character expression is assigned to a substring, only that part of the string is affected. The positions outside the specified boundaries of the substring retain their contents.

Character strings may be compared with each other using the same relational operators available for numbers. The comparisons .EQ. and .NE. are self-evident with regard to character strings; use of the others is based on each character's relative position in the computer's *collating sequence*. For most machines the collating sequence corresponds to alphabetical order, so that 'A' .LT. 'B', 'B' .LT. 'C', and so on.

Strings may be searched for other strings with the INDEX function:

```
INDEX   (s1, s2)
```

searches string s1 for an occurrence of s2. If the search succeeds, the function returns the starting position of the first match; otherwise it returns a zero.

**PROBLEMS**

1. Assume we have the following declarations:

```
CHARACTER       DSC1*12, DSC2*7
INTEGER*2       LOC1, LOC2
DATA            DSC1, DSC2/'THOROUGHFARE', 'SUMMARY'/
LOC1 = 1
LOC2 = 4
```

   Indicate the value for each of the following:
   (a) DSC1(1:10)
   (b) The character in position 4 of DSC1
   (c) The positions of the letters in DSC2 that also appear in DSC1
   (d) DSC2(3:4)
   (e) DSC1(6:6)
   (f) DSC2(4:)
   (g) DSC1(LOC2:LOC2)
   (h) DSC2(:LOC1)
   (i) DSC1//'S'
   (j) 'CAR'//DSC1(9:)
   (k) 'C'//DSC2(5:6)//DSC1(2*LOC2+LOC1:3*LOC2)
   (l) DSC1(7:LOC2)  (be careful)
   (m) DSC2(4:9)  (again, be careful)
   (n) DSC1(1,8)/'GOING'

2. Assume the following declarations:

```
CHARACTER*8     CH1, WD1*4, WD2*4, CH2, CH3, K1*1, K2*1
```

   Specify the values for all of the variables as a result of each of the following sequences of statements:

   (a)
```
DATA CH1, CH3/'PLATFORM', 'BASEBALL'/WD1, K1/'MOAT', '6'/
EQUIVALENCE (CH2, CH1), (WD2, WD1), (K2, WD1)
```
   (b)
```
DATA CH1, CH2, K2, WD2/'FOOTBALL', 'KINGDOMS', '*', 'SEAT'
EQUIVALENCE (K1, CH1(4:4)), (WD1, CH2)
CH3 = WD1//WD2
```
   (c)
```
DATA CH3/'BESOTTED'/WD1/'NOTE'/
EQUIVALENCE (WD2, CH3(2:)), (K1, WD2), (K2, K1)
CH1 = K1//CH3
CH2 = K2//K1//WD2//WD1
```

3. Assume the following declaration

```
CHARACTER*8     PREP(4, 3)
```

and some type of processing that fills the array with the values shown below:

```
AT          TO          BY
WITH        UNTIL       IN
FROM        WITHIN      WITHOUT
BEYOND      BELOW       OVER
```

Specify the value or write the FORTRAN description for each of the following:

(a) The element containing the fewest blanks.
(b) The value in PREP(3,2).
(c) The value in PREP(1,2)(:2).
(d) The value in PREP(2,2(3:).
(e) The value in PREP's ninth element.
(f) The location of the element containing the most vowels.
(g) The string WITH occurs in three different places; describe each one.
(h) The value formed by the expression PREP(1,1)//PREP(1,2).
(i) The value formed by the expression PREP(2,3)(:3)//PREP(1,3)(1:2).
(j) An expression that forms the string ROVER using only the characters in a single element of PREP.
(k) An expression that forms the string FORMAT using only those elements in PREP's first column.
(l) Declare a 4-character string named FIRST and write a sequence of statements that will fill FIRST with the string BIWO.

4. We have the following declarations:

```
CHARACTER       WD1*4, CH*8, VOCAB*10, PEARL*6, UTTER*12, L1*1
INTEGER*2       POS
DATA            WD1, CH, L1, POS/'BEAM', 'CONDUCTS', 'R', 4/
```

Show the outcome of each of the following assignments, treating each part independently. When the destination of a particular assignment statement is expressed as a substring, show the resulting value in that entire string:

(a) VOCAB = WD1
(b) VOCAB = CH(:7)//WD1(POS/2:2)//L1//WD1(8:)
(c) PEARL = CH(3:)
(d) PEARL = WD1//WD1
(e) WD1(POS:) = L1
(f) UTTER = WD1(:2)//CH//WD1
(g) UTTER = CH(:3)//CH(2*POS:)//CH(7:7)//'ITIO'//CH(3:POS-1)
(h) Show the value in UTTER after the following sequence:

```
UTTER = 'EXASPERATION'
UTTER(POS:5) = CH(6:7)
UTTER(6:6) = ' '
```

(i) Show the value in PEARL after the following sequence:

```
UTTER = 'STRATOSPHERIC'
DO 9 I=1,6
    PEARL(I:I) = UTTER(2*I:2*I)
9 CONTINUE
```

(j) Show the value in PEARL after the following sequence:

```
IF (CH(1:) .EQ. CH(6:6) THEN
    PEARL = 'SPHERE'
ELSE
    PEARL = 'OVOID'
END IF
```

(k) Show the value in PEARL after the following sequence

```
IF (WD1 .GE. L1//WD1(2:)) THEN
    PEARL = R1
ELSE
    PEARL = WD1(3:)
END IF
```

(l) Show the value in PEARL after the following sequence:

```
PEARL = '        '
IF (WD1(2:) .LE. WD1(2:3)//CH(7:) THEN
    PEARL (2:4) = CH(6:)
    PEARL(5:) = PEARL(2:3)
ELSE
    PEARL(3:) = CH(4:7)
    PEARL(:2) = PEARL(5:)
END IF
```

5. Assume the following sequence of statements:

```
CHARACTER*15      DUM, DEE
DUM = 'KOFEIUYT9Y6**$Q'
```

Write a sequence of FORTRAN 77 statements that will result in DEE containing DUM's characters in reverse order. Make any additional declarations you may need.

6. Using the same declarations as in the previous problem, write a sequence of statements that will produce a copy of DUM in DEE with all vowels replaced by dollar signs.

7. Using the same declarations as in Problem 5, write a sequence of statements that will place all of DUM's vowels in DEE's leftmost positions (in the same order in which they appear in DUM) followed by enough blanks to fill the rest of the string. Make any additional declarations you may need.

8. Using the same declarations as in Problem 5, write a sequence of statements that will place all of DUM's vowels in the rightmost positions of DEE, padded to the left with the appropriate number of blanks. Make any additional declarations you may need.

9. Assume the following sequence of statements:

```
INTEGER*2         N1, N2, N3
CHARACTER         W1*1, W4*4, W2*2, W10*10
W10 = 'MONOTONOUS'
W1 = 'N'
```

solve each of the independent problems given below:

(a) Indicate the value in N1 after the statement

```
N1 = INDEX(W10, W1)
```

(b) Indicate the value in N1 after the statement

```
N1 = INDEX(W10(4:), W1)
```

(c) Indicate the value in N2 after the following sequence:

```
W2 = W10(2:3)
N1 = INDEX(W(5:), W2)
IF (N1 .GE. 5) THEN
    N2 = 7
ELSE
    N2 = 8
END IF
```

(d) Indicate the value in W2 after the following sequence:

```
N1 = INDEX(W10, W1)
N2 = N1 + INDEX(W10, 'T') + 1
W2 = W10(N1:N1)//W10(N2:N2)
```

(e) Indicate the value in W4 after the following sequence:

```
N1 = 5
N2 = INDEX(W10(N1:), 'ON')
N3 = N1/N2
W4 = W1
W4(3:) = W10(N3:N2)//W10(N1:)
```

10. The simplest type of cryptogram is one in which each type of character in the original text is replaced by another to form the coded message. For example, in a particular system, all B's might be replaced by W's, C's by Y's, blanks by T's, and so on. If we know what this relationship is for a particular cryptogram, we can convert that cryptogram easily to its original form.

As it turns out, the Blazvoolean intelligence agency uses just such a system, and the key has fallen into our hands: Their system uses only 27 characters—the 26 letters and a blank. In terms of "alphabetical order," the blank is placed ahead of the A. With this as a basis, the scheme replaces each character with the one that is two positions further up in the "alphabet." Thus, when a message is converted to code form, C replaces A, D replaces B, X replaces V, Y replaces W, Z replaces X. Since there is no more alphabet, the system "wraps around" at Z, so that Y is replaced by a blank, and Z is replaced by A.

Write and run a program that reads a message already presented in this code. The program is to display the message as submitted, followed by a blank line and a display of the same message, this time in decoded (human readable) form. Input is organized as described for Example 8.3. Run your program using the data shown in Figure 8.20.

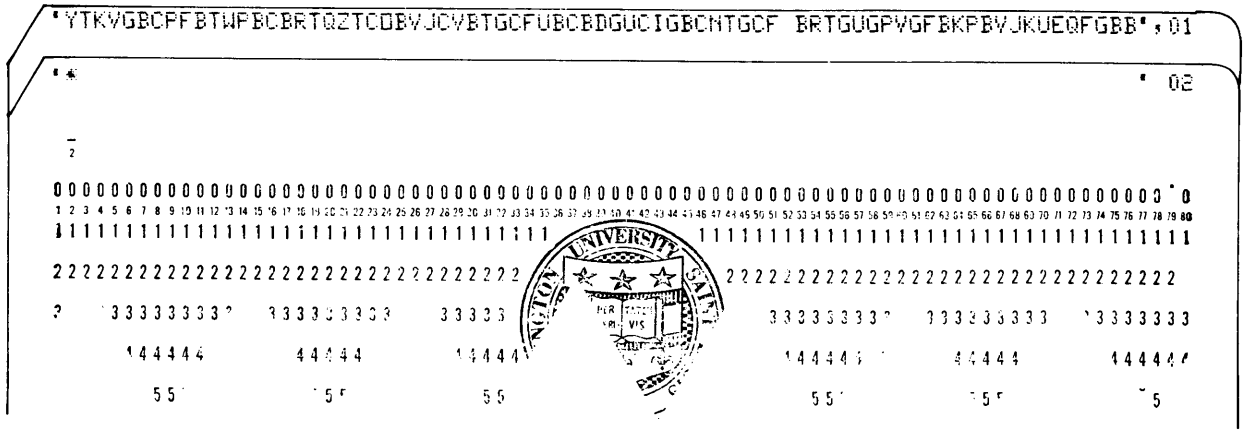


**FIGURE 8.20** Input Showing Blazvoolean Cryptogram.

11. Alert work and clever investigation has revealed that the Procrustean intelligence organization uses the same basic type of coding system as in Problem 10, but with some added flexibility: Instead of staying with one type of shift, they use a different shift with each message. Specifically, when a message is prepared, it is preceded by an integer value specifying how far up or down the alphabet each replacement character sits in relation to the character it is replacing. For instance, the replacement rule in the previous problem corresponds to a shift value of +2. A shift of +4 would mean that D replaces a blank, E replaces A, F replaces B, and so on. Similarly, a shift of −3 would mean that X replaces blank, Y replaces A, Z replaces B, blank replaces C, and so on.

    Write a program that meets the same basic input/output requirements as described for the previous problem. The difference here is that the input character data are preceded by a single line that specifies the shift. Input for a suggested test run is given in Figure 8.21.

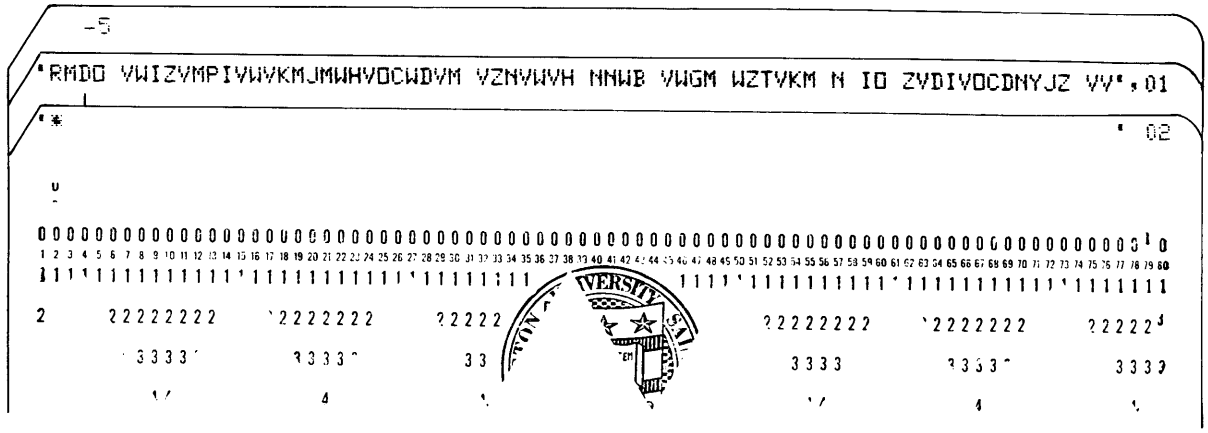


**FIGURE 8.21** Input for a Test Run for Decoding the Procrustean Cipher.

12. The J. P. Middleman Company is a wholesale distributor dealing in a wide variety of manufactured parts and components. For years they managed their business in a haphazard way, but now the competition is killing them with computers: Everything is systematized and Middleman must mend their ways. One of the areas that needs tightening stems from the fact that the various parts are identified using a scheme invented decades ago by the president's brother-in-law William E. Nilley, Jr. Some parts were identified by numbers, some by names containing only letters, others by combinations of letters and numbers—all in all a mess. This is to be straightened out by a simple system that assigns an all-numerical identifier to each part:

    1. Identifiers containing only numbers are left alone.
    2. Identifiers containing only letters are converted with each letter being replaced by a number corresponding to that letter's position in the alphabet. For instance, A is replaced by 1, E by 5, X by 24, and so on.

3. Identifiers containing letters and numbers are handled by leaving the numbers alone and converting the letters as described for all-letter identifiers. In addition, a 0 (zero) is placed at the beginning of the converted identifier. (It was found that there are no part identifiers beginning with zero in the current system.)

A few examples will clarify: A part identifier of 62755 will remain 62755; an identifier of FIZZLR would be converted to 6926261218; an identifier of 21GJ7B would convert to 02171072. Careful examination of the inventory established the fact that this conversion would guarantee against producing the same identifier for more than one type of item.

Write a program that reads a succession of old identifiers (one to a line) and displays that identifier, along with its new version. An old identifier is never more than six characters long, no matter what type it is. (Of course, some new ones will have to be longer.) Stop the run with an identifier of XXXXXX. Each line of output is to look like this:

```
PART NAME     FIZZLR     CHANGED TO     6926261218
```

After all the input has been processed, the program is to display four additional lines showing how many all-digit part identifiers were processed, how many all-letter identifiers were processed, how many mixed identifiers were processed, and the total number of identifiers processed (not counting the XXXXXX). One more thing: Old identifiers consist only of letters, numbers and blanks, and the blanks always are at the right. Here are some suggested input values:

```
'FIZZLR'
'320311'
'BJ033'
'2W'
'ZX3XZ'
```

13. Sir Melville Thrinckleton left a peculiar will: He was sure that the village of Threckfordshirehamburghville was the cradle of his family, and that many of his distant relatives still lived there. They didn't know they were descended from the rich and powerful Duke of Thrinckleton, but Sir Melville had a theory. He believed that his relatives could be traced through their names. If the last name had a TH in it, followed anywhere by a CK, that family (or so Sir Melville believed) was related. Accordingly, the will set up a fund that would pay for a crack team of genealogists to travel to Threckfordshirehamburghville and check the population. Every family with the appropriate type of last name would then receive a handsome simulated vinyl wall sculpture of the Thrinckleton coat of arms (at no charge) and a fine certificate attesting to the fact that the family had indeed received a handsome simulated vinyl wall sculpture, etc. The sleepy village of (no, we are not going to do that again) will never ever be the same after that.

    Luckily, the village is considerably more modernized than was first thought so that much of the civic data already existed in computer-compatible form. Thus, for each household, the genealogists were able to acquire an input line showing the last name (up to 25 letters) and a six-digit identification number. Now they need a program to process the lines and display the name and i.d. number for each household found to be eligible for a handsome, simulated, etc. End the run with a dummy last name of blanks. After the last line has been processed, the program is to display a line indicating the total number of eligible households. (Note that a name like THROCKLEY is eligible while a name like BRACKTHISTLE is not; reread the problem if you are not sure why this should be so.) Here are some suggested input values:

```
'THROCKMORTON'     324655
'SNAPTHICKET'      801742
'BILGEWEATHER'     280997
'THIMBLEORRICK'    324488
'FRANCKMISTLE'     798632
'WACKTHRESHER'     099898
'THWEBWYTHE'       792123
```

14. Rewrite the program in Example 8.3 (Section 8.4) so that the input no longer is arranged with complete words on each line. Columns 79–80 still are used for sequence numbers but a word may start on one line, go up through column 76 (column 77 still contains an apostrophe to finish the character constant) and continue in column 2 of the next line. (Column 1 still contains an apostrophe to start the new character string.) In other words, if we refer to the description of the example in Section 8.4, the ground rules numbered 1 and 7 no longer are true for this problem; all the rest still hold. Some suggested input is given in Figure 8.22.

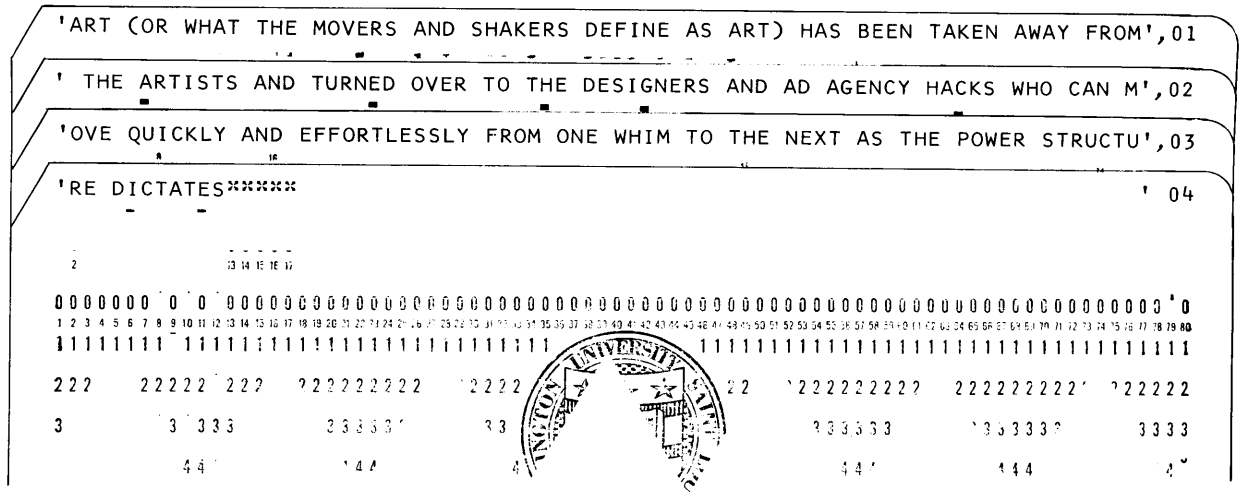


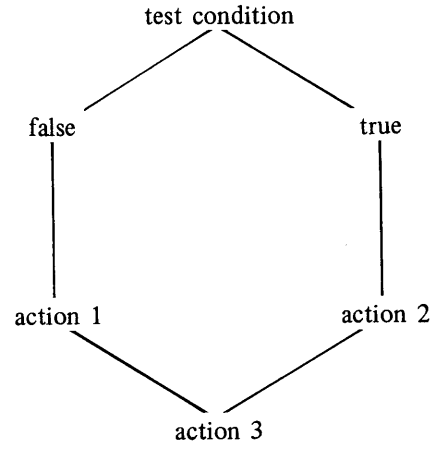

**FIGURE 8.22** Input for Problem 14, Chapter 8.

# 9
# *Control Structures and Decision Making*

The power and convenience of FORTRAN 77 should be quite apparent by now: Without much fanfare, we have worked our way to some fairly intricate procedures using but a few simple features. Now we are ready to tackle an even wider spectrum of decision-making possibilities by learning and applying FORTRAN 77's full range of facilities for specifying tests and using their outcomes to dictate further action. These specifications, called *control structures*, will be examined systematically, within the effective, orderly framework of structured programming.

## 9.1 THE IF-THEN-ELSE CONSTRUCT

It will not hurt to restate the basic decision mechanism that forms one of the cornerstones of structured programming:




We can summarize the behavior of this construct as follows:

> A test is performed in accordance with the rules defined by the specified test condition. Regardless of how complicated that test may be, its construction is such that it will have one of only two possible outcomes: either .TRUE. or .FALSE.. If the outcome is .TRUE., the program will carry out the processing indicated by "action 1," ignoring "action 2." An outcome of .FALSE. will produce the opposite effect, i.e., performance of "action 2." In *either case* the program continues with "action 3."

A lot can happen within this simple construct, and we shall explore the possibilities in two ways: First, we shall examine the kinds of test conditions that can be specified. Then, we shall look at ways of organizing and describing increasingly extensive actions to be triggered by these tests.

### 9.1.1 Specification of Test Conditions

We have set up situations in which tests have been based on a wide variety of numerical and character expressions. Yet, despite their diversity, these conditions all have had one thing in common: In each test, a single requirement had to be met in order for the outcome to be .TRUE.. Of course, that requirement could be quite complicated. For example, the condition described by

```
(X**1.8/(2.34*LOG(X+SQRT(3.0-.7*X))-Y) .LE. 3.38*(Z+SQRT(Y*Z)))
```

is by no means simple; implied in it is a request for a fair amount of processing. Yet, it still describes a *single* "true or false" condition, one comparison whose outcome will determine whether the THEN action or ELSE action will be followed.

This is not always adequate. There are many everyday situations where an outcome of .TRUE. is based on meeting several requirements at the same time, or meeting a choice of requirements. It is not necessary to reach far to find situations like this. For instance, it is perfectly reasonable for an accounting supervisor in a business to say, "Let me see the invoice summaries for all customers in Illinois who owe us at least one thousand dollars and have not made a payment in the last thirty days." Under these specifications, no less than three conditions have to be met in order for a customer record to reach the supervisor's eagle eye. Just being from Illinois is not enough (although that alone might be worthy of attention in some other situation); nor is a debt of at least a thousand dollars sufficient to have the record singled out. The same thing applies to the time elapsed since the last payment. A record with two out of three still does not qualify.

It certainly is possible to represent such multiple conditions in FORTRAN by using an IF statement for each condition and letting the program work its way through such a combination. However, the resulting construction is awkward and generally disorderly. The next few sections will introduce additional features and develop simple but effective techniques for implementing these types of tests.

**9.1.1.1 Combinations of Comparisons: the AND Operation** FORTRAN provides a natural way to express multiple comparisons: They are built simply by connecting individual comparisons with *AND operators*. The general form for this construction is as follows:

```
IF (compar1 .AND. compar2 .AND. compar3. . . . AND. comparn) THEN
    action 1
ELSE
    action 2
END IF
```

In this general example, compar1, compar2, compar3, ..., comparn represent an arbitrary collection of $n$ comparisons linked together by the .AND. operations. (Note that the periods are required on either side of the AND, just as they are for .EQ., .LT., and the other relational operators.) Each comparison is of the type already familiar to us, producing an outcome that is either .TRUE. or .FALSE.. These individual outcomes are submerged within FORTRAN, so that the programmer views the multiple conditions as one grand and glorious comparison whose final, *single* outcome is .TRUE. only if all $n$ comparisons turn out to be .TRUE..

**Example 9.1** People at the Ipicoopchick County Health Authority are excited. After three frustrating weeks without a clue, there appears to be a positive break in their efforts to track down a sudden increase in the occurrence of fainting spells among the women. Examination of the hospital records has revealed that all of the victims were over 28, had brown eyes, and lived in dwellings built by Hovel Construction Co. While trying to determine exactly what is happening, the County now has an opportunity to warn other residents that they may be potential targets of this mysterious attack. In addition to newspaper ads and

television announcements, the authorities have decided to send individual letters. Accordingly, we would like to design a program that looks through the County data, identifies the appropriate residents, and prints their names and addresses. In addition, the program is to count the number of people to whom letters need to be sent, as well as the total number of residents.

Ipicoopchick County has computerized for its residents. Information for each resident consists of an input line containing a name (20 characters), street number (4 digits), street name (8 characters), township (8 characters), zipcode (5 characters), year of birth (4 digits), sex code (M or F), eye color code (1=blue, 2=brown, 3=green, 4=gray, 5=hazel, 6=other), and first year in the County (4 digits). The state is not recorded since all of the data pertain to the same state (NY).

It is obvious from the description of the recorded data that we can do nothing about determining the resident's type of dwelling; the information simply is not there. Consequently, this is not part of the computer's job and will be left, instead, to the letter that is sent. By the same token, each record contains information that is not needed, i.e., the resident's first year in the County. The unformatted **READ** statement forces us to read and store it anyway, since it appears as part of the data. Figure 9.1 shows the diagram and pseudocode for the program, and the statements themselves are in Figure 9.2. Note that parameters are put to good use to increase the program's clarity. Also note that the **ELSE** was omitted since there was no alternative action. This is a direct reflection of what is seen in Figure 9.1

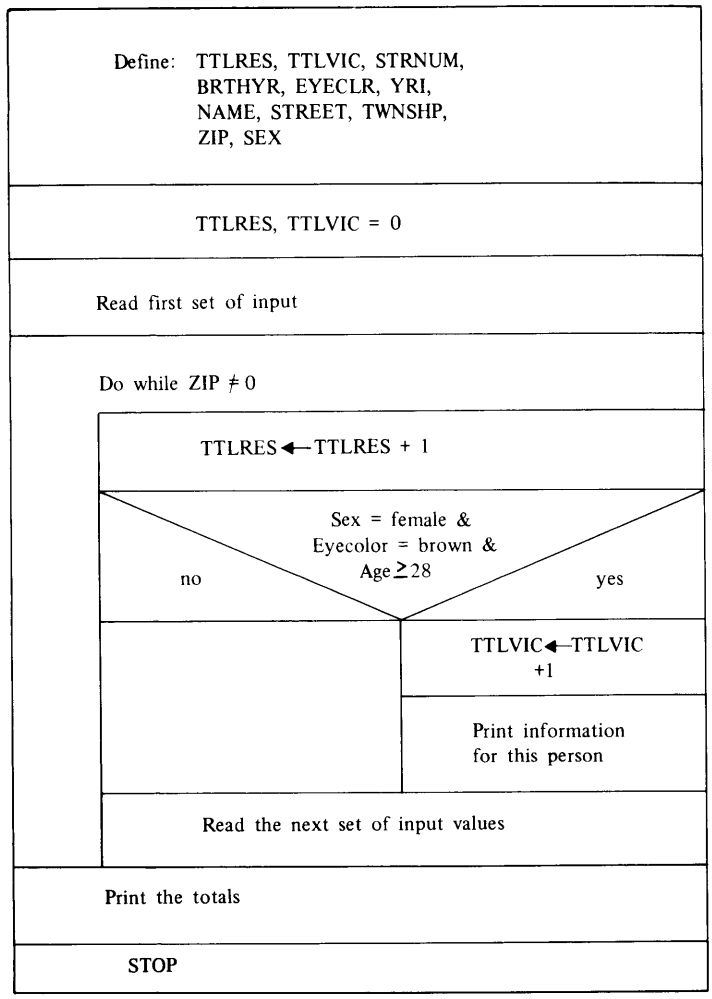



**FIGURE 9.1 (a)** Diagram for Example 9.1

```
"Define  TTLRES,TTLVIC,STRNUM,BRTHYR,EYECLR,
         YR1,NAME,STREET,TWNSHP,ZIP,SEX."
"Initialize TTLRES and TTLVIC to zero."
"Read the first set of input." while ZIP is not equal to zero:
    "Add 1 to TTLRES."
    if
         "SEX is female, eye color is brown and age at least 28"
    then
         "Add 1 to TTLVIC."
         "Print i.d. and address information for this resident."
    endif
    "Read the next resident's input data."
endwhile
"Print the totals."
"Stop."
```

**FIGURE 9.1 (b)** Pseudocode for Example 9.1

```
************************************************************************
*                          EXAMPLE 9.1                                 *
************************************************************************
*    VARIABLE NAMES:                                                   *
*         NAME:          RESIDENT'S NAME:                             *
*         STRNUM:        STREET NUMBER                                *
*         STREET:        STREET NAME                                  *
*         TWNSHP:        TOWNSHIP                                     *
*         ZIP:           ZIPCODE                                      *
*         BRTHYR:        YEAR OF BIRTH                                *
*         SEX:           RESIDENT'S SEX                               *
*         EYECLR:        EYECOLOR                                     *
*         YR1:           FIRST YEAR IN THE COUNTY                     *
*         TTLRES:        TOTAL NUMBER OF RESIDENTS                    *
*         TTLVIC:        TOTAL NUMBER OF POTENTIAL VICTIMS            *
*---NOTE THAT ZIPCODE WILL BE TREATED AS A 5-CHARACTER STRING         *
*---SINCE THAT IS WHAT IT REALLY IS. NO COMPUTING IS DONE ON          *
*---ZIPCODES.                                                         *
*---THE SYMBOL $ IS USED HERE TO INDICATE A CONTINUED STATEMENT       *
*---(RECALL THAT FORTRAN REQUIRES A NON-BLANK IN COLUMN 6)            *
************************************************************************

      PROGRAM        EX901
      IMPLICIT       NONE
      INTEGER*2      TTLRES,TTLVIC,STRNUM,BRTHYR,EYECLR,YR1,BROWN
      CHARACTER      NAME*20,STREET*8,TWNSHP*8,ZIP*5,SEX*1,
     1               STATE*2,FEMALE*1
      PARAMETER      (STATE='NY',BROWN=2,FEMALE='F')
      TTLRES = 0
      TTLVIC = 0
      PRINT *, 'ENTER THE DATA FOR THE FIRST INDIVIDUAL'
      READ *, NAME,STRNUM,STREET,TWNSHP,ZIP,BRTHYR,SEX,EYECLR,YR1
*-------------A ZIPCODE OF ZERO WILL STOP THE RUN.----------------
      DO WHILE (ZIP .NE. '00000')
         TTLRES = TTLRES + 1
         IF (SEX .EQ. FEMALE .AND. 1979-BRTHYR .GE. 28
     1                       .AND. EYECLR .EQ. BROWN) THEN
           TTLVIC=TTLVIC+1
           PRINT *, NAME,STRNUM,STREET,TWNSHP,STATE,ZIP
         END IF
         PRINT *, 'ENTER THE DATA FOR THE NEXT INDIVIDUAL'
         READ *, NAME,STRNUM,STREET,TWNSHP,ZIP,BRTHYR,SEX,EYECLR,YR1
      END DO

*-------------NO MORE DATA; TIME TO SUM UP.-----------------------

      PRINT *, '  '
      PRINT *, 'TOTAL NO. OF RESIDENTS: ',TTLRES
      PRINT *, 'TOTAL NO. OF POTENTIAL VICTIMS: ',TTLVIC
      STOP
      END
```

**FIGURE 9.2** FORTRAN Statements for Example 9.1.

**9.1.1.2 A Choice of Conditions: The OR Operation** There are times when only one of several conditions needs to be true in order for a particular action to be taken. For instance, a sales manager might want a list of all people assigned to any of three geographic regions. This type of situation can be specified directly by connecting the choices with the *OR operator*. The general form for this construction is

```
IF (compar1 . OR. compar2 . OR. .....comparn) THEN
   action 1
ELSE
   action 2
END IF
```

Thus, if REGION is the variable denoting the geographic area, a statement selecting personnel from regions 2, 4, and 7 would read

```
IF (REGION . EQ. 2 . OR. REGION . EQ. 4 . OR. REGION . EQ. 7) THEN
   action 1
ELSE
   action 2
END IF
```

**9.1.1.3 Combined Conditions** By using the AND and OR operations in various combinations, it is possible to describe an endless variety of tests, thereby producing programs that are powerful decision-making mechanisms. In the next chapter we shall look at the rules that govern the construction of these combinations. Prior to that, however, it will be helpful to introduce these techniques informally so that their ease of use is established. The next example will serve that purpose.

**Example 9.2** The Pompeii Realty Company is preparing to computerize its operations. (Understandably, there is great joy at Pompeii.) Their plans are to record information about each house on an input line. Included will be the listing number (HLIST), street number (STRNUM), street name (STREET), township (TWNSHP), number of bedrooms (NBDRMS), number of full baths (NFBTHS), number of half baths (NHBTHS), square feet of living area (AREA), lot size in square feet (LOTSZ), and price to the nearest cent (PRICE). With this information available, the program can read an input line containing the client's requirements. The program then would read a line for each house, comparing its characteristics with those specified by the buyer. For each match, the program would print the information about the house, including its identification.

For purposes of illustration, we shall look at a simplified version of such a program, one with a restricted range of comparisons. Specifically, we shall allow a customer to specify a choice of two townships (TWN1 and TWN2), a minimum number of bedrooms (BDRMIN), and a price range (i.e., the house could cost at least PR1 dollars and cents but no more than PR2). The program will print all of the available information for each house meeting the buyer's requirements. Then, after all the houses have been processed, the program is to print the number of houses that matched the requirements and the number that did not.

The crucial part of the program, of course, is the test that determines whether the house currently being "examined" is a match or not. We shall build this test systematically by recognizing that there are three major types of comparisons: First of all, there is price. For a buyer to be selected, its cost must be equal to or greater than the buyer's bottom limit PR1 and, at the same time, it must be equal to or less than PR2. So our price comparison, when written as part of a FORTRAN decision statement, would read

```
(PRICE . GE. PR1 . AND. PRICE . LE. PR2)
```

Our second comparison is between the township in which the house is located (TWNSHP) and the two choices TWN1 and TWN2. A match with either one would be acceptable. This is stated as

```
(TWNSHP . EQ. TWN1 . OR. TWNSHP . EQ. TWN2)
```

The third comparison matches the number of bedrooms against the buyer's minimum:

```
(NBDRMS . GE. BDRMIN)
```

Now, since all three of these comparisons must be true in order for the house to be selected as a possible candidate for sale, we can produce the complete matching test by combining the three components with AND operators and placing the result within the context of an IF statement:

```
   IF       ( (NBDRMS . GE. BDRMIN)
1  . AND.   (PRICE . GE. PR1 . AND. PRICE . LE. PR2)
2  . AND.   (TWNSHP . EQ. TWN1 . OR. TWNSHP . EQ. TWN2) ) THEN
     action 1
   ELSE
     action 2
   END IF
```

Note that the additional parentheses are legal, as they are in arithmetic or character expressions, and they help clarify exactly what comparisons are taking place.

Now, with the heart of the decision mechanism defined, we can turn our attention to the complete program. A flowchart and pseudocode are shown in Figure 9.3 and the statements are given in Figure 9.4.

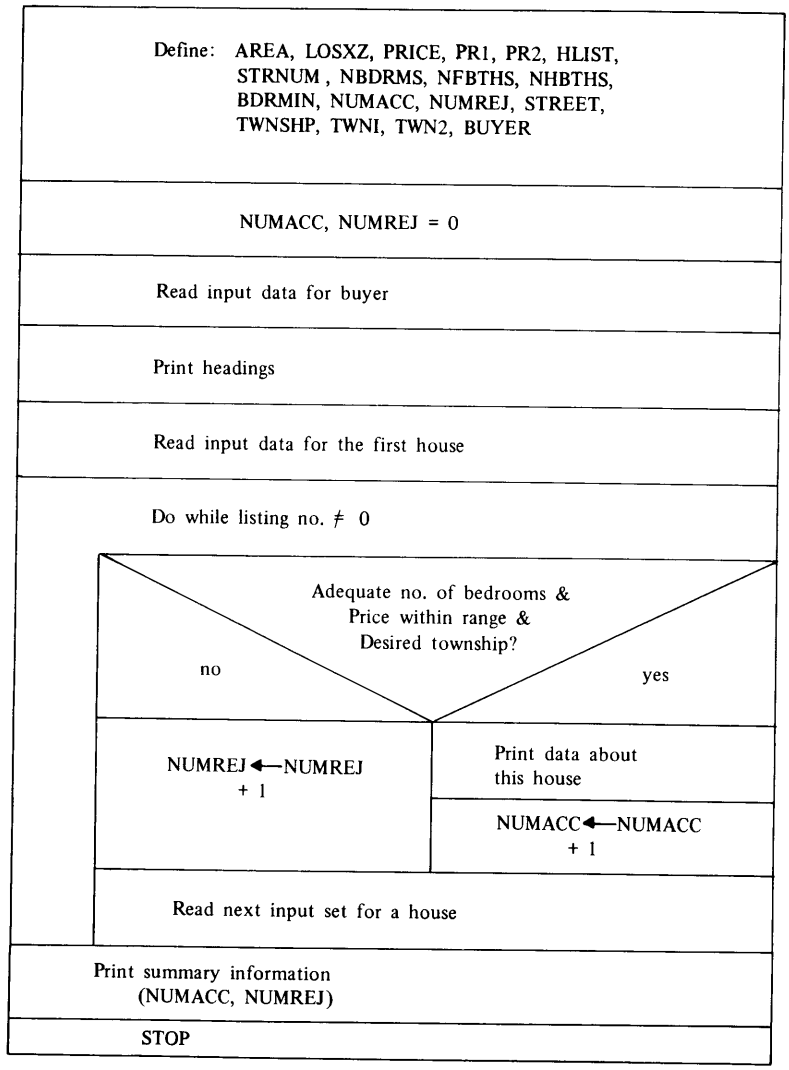



**FIGURE 9.3 (a)** Diagram for Example 9.2.

```
"Define  AREA,LOTSZ,PRICE,PR1,PR2,HLIST,
         STRNUM, NBDRMS,NFBTHS,NHBTHS,
         BDRMIN,NUMACC,NUMREJ,
         STREET,TWNSHP,TWN1,TWN2,BUYER."
"Set NUMACC and NUMREJ to zero."
"Read the buyer's input data."
"Print the output headings."
"Read input data for the first house."
while the listing no. is not equal to zero:
        if
        there is a sufficient no. of bedrooms
            and
        the price is in the buyer's range
            and
        the house is in the desired township
    then
        "Print the data about the house."
        "Add 1 to the number of acceptable houses."
    else
        "Add 1 to the number of rejected houses."
    endif
    "Read the input data for the next house."
endwhile
"Print the output summary."
Stop."
```

**FIGURE 9.3 (b)** Pseudocode for Example 9.2.

```
************************************************************************
*                          EXAMPLE 9.2                                 *
*     HLIST,BUYER:     HOUSE I.D. NUMBER, BUYER'S NAME                 *
*     STRNUM,STREET:   STREET NUMBER, NAME                             *
*     TWNSHP:          TOWNSHIP                                        *
*     NBDRMS,BDRMIN:   NO. OF BEDROOMS,  MIN. REQ'D NO. OF BEDROOMS    *
*     NFBTHS,NHBTHS:   NO. OF FULL BATHS, HALF BATHS                   *
*     AREA,LOTSZ:      LIVING AREA, LOT SIZE, SQ. FT.                  *
*     PRICE:           PRICE OF THE HOUSE  ($XXXXXX.XX)                *
*     TWN1,TWN2:       BUYER'S FIRST AND SECOND TOWNSHIP CHOICES       *
*     PR1,PR2:         BUYER'S MINIMUM AND MAXIMUM ACCEPTABLE PRICES   *
*     NUMACC:          NO. OF HOUSES MATCHING BUYER'S REQUIREMENTS     *
*     NUMREJ:          NO. OF HOUSES NOT MATCHING REQUIREMENTS         *
************************************************************************
      PROGRAM          EX902
      IMPLICIT         NONE
      REAL*4           AREA,LOTSZ,PRICE,PR1,PR2
      INTEGER*2        HLIST,STRNUM,NBDRMS,NFBTHS,NHBTHS,BDRMIN,
     1                 NUMACC,NUMREJ
      CHARACTER        STREET*10,TWNSHP*8,TWN1*8,TWN2*8,BUYER*20
*-----INITIALIZE THE PROCESSING BY SETTING THE COUNTERS TO ZERO,
*-----READING THE BUYER'S INPUT DATA, AND PRINTING IT, ALONG
*-----WITH A GENERAL HEADING. THEN, READ THE FIRST HOUSE'S DATA.
      NUMACC=0
      NUMREJ=0
      PRINT *, 'ENTER THE DATA FOR THE BUYER'
      READ *, BUYER,TWN1,TWN2,BDRMIN,PR1,PR2
      PRINT *, 'POMPEII REALTY, INC.'
      PRINT *, 'SPECIAL ANALYSIS PREPARED FOR ',BUYER
      PRINT *, ' '
      PRINT *, 'ENTER THE DATA FOR THE FIRST HOUSE'
      READ *, HLIST,STRNUM,STREET,TWNSHP,NBDRMS,NFBTHS,NHBTHS,
     1          AREA,LOTSIZ,PRICE

      DO WHILE (HLIST .NE. 0)
         IF        ( (NBDRMS .GE. BDRMIN)
     1       .AND.    (PRICE .GE. PR1 .AND. PRICE .LE. PR2)
     2       .AND.    (TWNSHP .EQ. TWN1 .OR. TWNSHP .EQ. TWN2)) THEN
            PRINT *, HLIST,STRNUM,STREET,TWNSHP,NBDRMS,NFBTHS,
     1                NHBTHS,AREA,LOTSZ,PRICE
            NUMACC = NUMACC+1
         ELSE
            NUMREJ = NUMREJ+1
         END IF
         PRINT *, 'ENTER THE DATA FOR THE NEXT HOUSE'
         READ *, HLIST,STRNUM,STREET,TWNSHP,NBDRMS,NFBTHS,NHBTHS,
     1      AREA,LOTSZ,PRICE
       END DO

       PRINT *, '   '
       PRINT *, 'NO. OF HOUSES SELECTED: ',NUMACC
       PRINT *, 'NO. OF HOUSES REJECTED: ',NUMREJ
       STOP
       END
```

**FIGURE 9.4** Statements for Example 9.2.

```
*                          EXAMPLE 9.3                                  *
*     HLIST,STRNUM:         HOUSE I.D. NUMBER, STREET NUMBER            *
*     STREET,TWNSHP:        STREET,TOWNSHIP                             *
*     TWN1,TWN2:            BUYER'S TOWNSHIP CHOICES                    *
*     NBDRMS,BDRMIN:        NO. OF BEDROOMS,  MININUM NO. REQ'D         *
*     NFBTHS,NHBTHS:        NO. OF FULL BATHS,  HALF BATHS              *
*     BFBMIN,BHBMIN:        MIN. REQ'S FULL BATHS,   HALF BATHS         *
*     AREA,AREAMN:          LIVING AREA,   BUYER'S MIN. AREA            *
*     LOTSZ,LOTMIN:         LOT SIZE,   BUYER'S MIN. LOT SIZE           *
*     PRICE,PR1,PR2:        PRICE, BUYER'S MIN., MAX PRICES             *
*     NUMACC,NUMREJ:        NO. OF HOUSES SELECTED, NO. REJECTED        *
*     NMATCH:               NO. OF MATCHES FOR A GIVEN HOUSE            *

      PROGRAM               EX903
      IMPLICIT              NONE
      REAL*4                AREA,AREAMN,LOTSZ,LOTMIN,PRICE,PR1,PR2
      INTEGER*2             HLIST,STRNUM,NBDRMS,BDRMIN,NFBTHS,NHBTHS,
     1                      BFBMIN,BHBMIN,NUMACC,NUMREJ,NMATCH
      CHARACTER             STREET*10,TWNSHP*8,TWN1*8,TWN2*8,BUYER*20
      NUMACC = 0
      NUMREJ = 0
      PRINT *, 'ENTER THE BUYER'S DATA'
      READ *, BUYER,TWN1,TWN2,BDRMIN,BFBMIN,BHBMIN,AREAMN,LOTMIN,
     1        PR1,PR2
      PRINT *, 'POMPEII REALTY, INC.'
      PRINT *, 'SPECIAL ANALYSIS PREPARED FOR ',BUYER
      PRINT *, '   '
      PRINT *, 'ENTER THE DATA FOR THE FIRST HOUSE'
      READ *, HLIST,STRNUM,STREET,TWNSHP,NBDRMS,NFBTHS,NHBTHS,
     1        AREA,LOTSZ,PRICE

      DO WHILE (HLIST .NE. 0)
         NMATCH = 0
         IF (TWNSHP .EQ. TWN1 .OR. TWNSHP .EQ. TWN2) NMATCH=NMATCH+1
         IF (NBDRMS .GE. BRDMIN)                     NMATCH=NMATCH+1
         IF (NFBTHS .GE. BFBTHS)                     NMATCH=NMATCH+1
         IF (NHBTHS .GE. BHBTHS)                     NMATCH=NMATCH+1
         IF (AREA .GE. AREAMN)                       NMATCH=NMATCH+1
         IF (LOTSZ .GE. LOTMIN)                      NMATCH=NMATCH+1
         IF (PRICE .GE. PR1 .AND. PRICE .LE. PR2)    NMATCH=NMATCH+1
         IF (NMATCH .GE. 4) THEN
            PRINT *, HLIST, STRNUM, STREET, TWNSHP, NBDRMS, NFBTHS,
     1           NHBTHS, AREA, LOTSZ, PRICE, NMATCH
            NUMACC=NUMACC+1
         ELSE
            NUMREJ=NUMREJ+1
         END IF
         PRINT *, 'ENTER DATA FOR THE NEXT HOUSE'
         READ *, HLIST,STRNUM,STREET,TWNSHP,NBDRMS,NFBTHS,NHBTHS,
     1        AREA,LOTSZ,PRICE
      END DO

      PRINT *, '   '
      PRINT *, 'NO. OF HOUSES SELECTED: ',NUMACC
      PRINT *, 'NO. OF HOUSES REJECTED: ',NUMREJ
      STOP
      END
```

**FIGURE 9.5** Statements for Example 9.3.

**9.1.1.4 Tests with Arbitrary Combinations** Another type of useful test is one in which there are several conditions, but it is not necessary to meet all of them in order for the outcome to be .TRUE.. However, unlike the situation in Section 9.1.1.2, where only one of the comparisons had to be .TRUE., this type of test requires several (but not all) of the comparisons to be .TRUE.. Furthermore, any number of different combinations could satisfy the test's requirements. For instance, each real estate line in the previous example had seven pieces of descriptive information about a particular house (not counting the i.d. number and address). Now, suppose that a buyer could specify a requirement for each of those seven items, and a house would be selected for the printout if at least four of the items matched; *any four*. If we were to construct this as a combined test (as we did in Example 9.2), it would be necessary to list all the possible combinations of four items. For this situation, with seven items from which to choose, this turns out to be 35 combinations. Speaking strictly in technical terms, that would be one whale of an IF statement.

A more effective way to do it is to abandon the idea of trying to look at each possible combination. Instead, we examine each item separately and keep track of how many matches were found. Thus, for seven items, we need only seven comparisons (each in its own IF statement). Every time a match is found, we add one to the match counter, and examine it at the end of the test cycle. If it has at least a 4 in it, the house is selected. It is as simple as that.

**Example 9.3** We shall design a more general version of the previous example: This time, the processing will be made much more flexible, allowing the buyer to specify requirements for each of the seven descriptive items listed for each house: In addition to minimum number of bedrooms, two township choices, and a price range as permitted before, the buyer is able to define minimum values for full baths, half baths, living area, and lot size. Information about a house is to be printed if any four of these seven criteria are met.

The program (Figure 9.5) is not much more complex than the previous version. In a way, it is somewhat simpler, because the long IF statement has been replaced with a series of simpler tests. Note that the THEN. . . ELSE construction is not needed until the last test, the one where we actually decide what to print.

### 9.1.2 Construction of Actions for THEN and ELSE

Thus far we have used FORTRAN's decision facilities to implement actions whose construction is relatively simple: One such construction, the most basic type in the language, specifies an action limited to a single statement:

```
IF (condition) statement
```

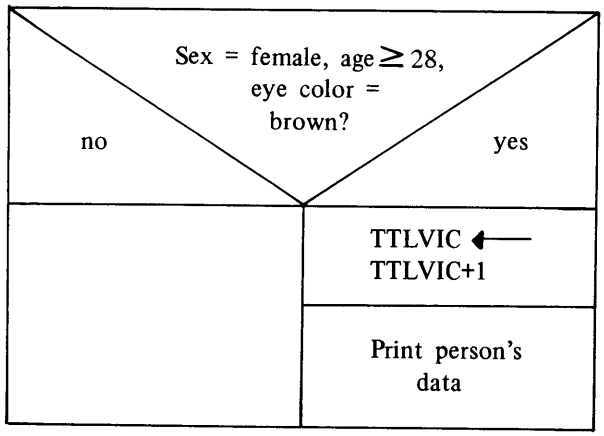
This is called a *logical IF statement*. Note that the THEN is missing. In fact, it cannot be used there. Furthermore, we cannot specify an alternative action because ELSE cannot be used with this form either. The reason this statement appears at all is that it was FORTRAN's only standard logical decision mechanism until a more versatile form was added to FORTRAN 77. Accordingly, we note the existence of the logical IF statement but deemphasize its use.

Specification of an IF-THEN-ELSE construct in FORTRAN 77 begins with *block IF statement*:

```
IF (condition) THEN
```

The word THEN is part of the block IF statement. Any action specified in conjunction with this statement (including the ones we have used in previous examples) is described in the form of one or more *IF-blocks* and concluded with an END IF statement:

```
IF (condition) THEN
    statement                          action, expressed as an IF-block
      .
      .
    statement
END IF
```




```
If
  (SEX .EQ. FEMALE        .AND.
  1979-BRTHYR .GE. 28 .AND.
  EYECLR. EQ. BROWN)

Then
TTLVIC=TTLVIC+1
PRINT *, NAME, STRNUM, STREET,
          TWNSHP, ZIP, BRTHYR,
          SEX, EYECLR, YR1
Endif
```

**FIGURE 9.6 (a)** Description of Block-IF Construction.

```
if
    sex is female
        and
    age is at least 28
        and
    eye color is brown
then
    "Add 1 to number of victims."
    "Print the person's data."
endif
```

**FIGURE 9.6 (b)** Decision Statement with IF-Block.

The action described by the IF-block may be as simple or complicated as it needs to be, and the programmer may use any number of statements to describe it. Regardless of the number of statements used, the form shown above remains a relatively simple type of IF block (from a structural point of view) since only one action is specified.

The decision mechanism used in Example 9.1 has this type of structure. Our test condition (i.e., selection of a resident as a potential victim) leads to a single action (adding 1 to the number of potential victims and displaying information about that person); the alternative is to perform no action at all. This is seen in Figure 9.6, where the statements are reproduced from the example program for convenience, along with the corresponding structure in diagrammatic and pseudocode form.

A second IF-block can be added by introducing an ELSE statement. This produces a direct representation of the IF-THEN-ELSE construct. Thus, we can restate this important building block in terms of FORTRAN's programming components:

```
IF (condition) THEN
      statement
            .                          action 1, expressed as an IF-block
            .
      statement
ELSE
      statement
            .                          action 2, expressed as an IF-block
            .
      statement
END IF
```

**9.1.2.1 Empty IF-Blocks** When the IF-THEN-ELSE construct was defined earlier as one of the fundamental components of a well-structured program, the discussion stressed the fact that either or both of the alternative actions could be as simple or as complicated as the programmer wished. This flexibility includes the extreme case in which one of the

alternatives is to do nothing at all. An instance of such a situation occurred in the decision structure of Example 9.1 (This part of the program was extracted and shown in Figure 9.6.) Note that the THEN appeared without an ELSE to go with it. Now that the block IF statement has been defined and we saw that the word THEN concludes this type of statement, it is clear that a single IF-block is being used to express the resulting action. Since the alternative is to do nothing at all, it does not appear.

When this type of situation occurs, many people prefer to preserve the full IF-THEN-ELSE structure by explicitly indicating an alternative, even if no action is specified. This is quite legal in FORTRAN 77, since the language allows the use of the ELSE statement without any accompanying action. Such usage often is called the *empty ELSE*. Thus, we could have written the decision rule from Example 9.1 as shown below without affecting the results:

```
      IF (SEX .EQ. FEMALE .AND. 1979-BRTHYR .GE. 28
     1                    .AND. EYECLR .EQ. BROWN)      THEN
          TTLVIC=TTLVIC+1
          PRINT *, NAME, STRNUM, STREET, TWNSHP, STATE, ZIP
      ELSE
      END IF
```

Structurally, this is the same as any other IF-THEN-ELSE construct: The block IF statement is followed by two IF-blocks, the second of which happens to contain nothing but the ELSE statement. As before, an END IF statement concludes the structure.

We can do the same kind of thing with the first IF-block. That is, we can write a structure such as

```
IF (condition) THEN
ELSE
     statement
          .
          .
     statement
END IF
```

thereby indicating that the action resulting from a successful test (i.e., a value of .TRUE.) is to do nothing. While this construction is possible, it should be pointed out that it is somewhat awkward in terms of overall program clarity. In most cases, it is a better idea to set up a decision structure so that some type of action is performed in response to a successful test result.

**9.1.2.2 Actions with Extended Decision Rules** A particularly powerful way to use the block IF facility is to build decision networks in which the final outcome, and actions triggered by it, are determined after a *series* of tests, performed one at a time. This situation is quite different from the type discussed in the previous sections, where we built increasingly complicated tests, but still produced a single outcome. Now, we are interested in developing structures that enable us to set up actions which themselves involve further testing. This is handled conveniently within the regular IF-block mechanism.

One way to look at a test performed by an IF statement is to note that it assigns the object of this test to one of two categories. For instance, suppose we are in the midst of a program that looks at a set of data items for various individuals. Right now, we are examining a particular person's gender (which we shall call GENDER) with the following programming:

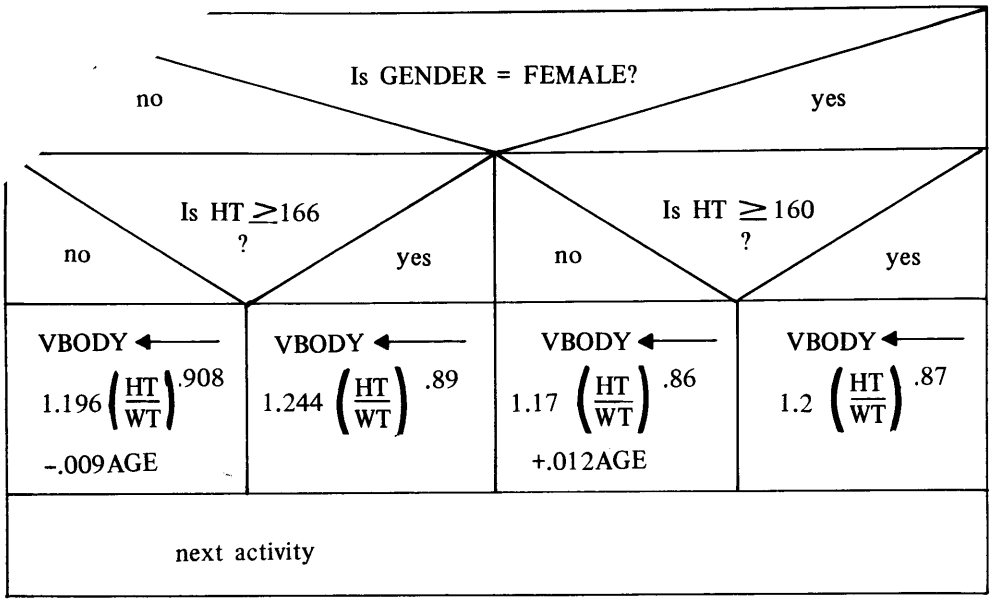



**FIGURE 9.7 (a)** Nested Decision Structure.

```
gender is female
then
    if
        height is at least 160
    then
        "Compute
    else
        "Compute
    endif
else
    if
        height is at least 166
    then
        "Compute
    else
        "Compute
    endif
endif
"Next activity."
```

**FIGURE 9.7 (b)** Pseudocode for a Nested Decision Structure.

```
IF (GENDER .EQ. FEMALE) THEN
   action 1
ELSE
   action 2
END IF
```

As a result of this test the person has been identified as being either female or male (i.e., not female). Now, we may want to make a further distinction, say, between taller and shorter females, and between taller and shorter males. Once that subdivision has been performed, we shall have four categories and we are interested in specifying a separate action for each one.This type of situation is shown in the diagram and pseudocode of Figure 9.7. For purposes of illustration we have defined some variable VBODY which is computed in four different ways depending on the outcome of the tests. Note from Figure 9.7 that there are three tests: the first one assigns an individual to the male or female category; then, *each* of the two categories has another test to assign the individual to one of four subcategories (taller males, shorter males, taller females, shorter females).

This subdivision process is expressed very naturally in FORTRAN 77 as an extension of the block IF construction:

```
      IF (GENDER .EQ. 'F')  THEN
         IF (HT .GE. 160.0)  THEN
            VBODY=1.2*(HT/WT)**0.87
         ELSE
            VBODY=1.17*(HT/WT)**0.86          action for females
     1       +0.012*AGE
         END IF
       ELSE
         IF (HT .GE. 166.0)  THEN
            VBODY=1.244*(HT/WT)**0.89
         ELSE
            VBODY=1.196*(HT/WT)**0.908        action for males
     1       -0.009*AGE
         END IF
```

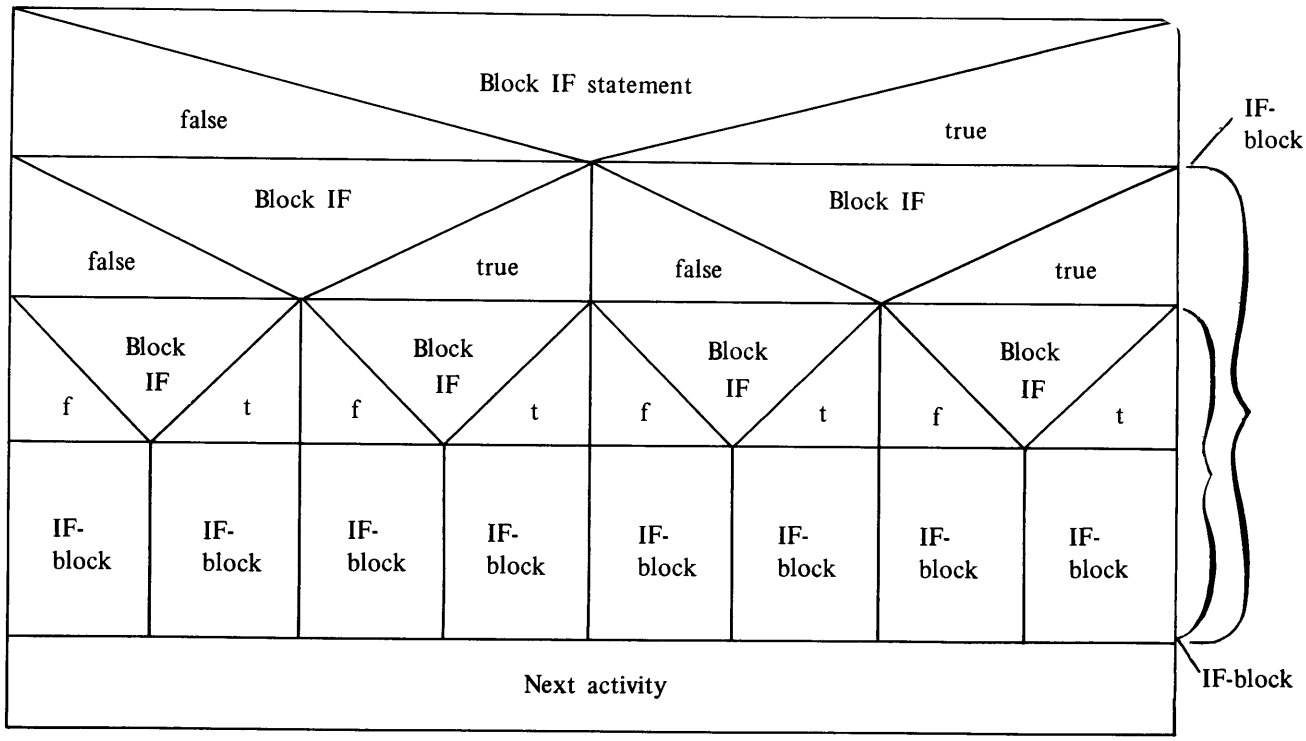



**FIGURE 9.8 (a)** Basic Construction of Nested Decisions.

```
if
    test1
then
    if
        test11
    then
        if
            test111
        then
            "Action."
        else
            "Action."
        endif
    else
        if
            test112
        then
            "Action."
        else
            "Action."
        endif
    endif
else
    if
        test21
    then
        if
            test211
        then
            "Action."
        else
            "Action."
        endif
    else
        if
            test212
        then
            "Action."
        else
            "Action."
        endif
    endif
endif
```

**FIGURE 9.8 (b)** Pseudocode for Deeply Nested Decisions.

When we look at this construction in general terms (Figure 9.8) we see it as a block IF statement in which each of the two alternative actions is itself a block IF statement with two associated actions. This technique of specifying decision structures inside other decision structures is called *nesting*. It can be carried to any level necessary. The amount of such nesting is not governed by any limitation within the language. Instead, it should be determined by the needs of the problem and the clarity of the resulting program.

**9.1.2.3 Construction of Nested Decisions** Use of nested IF-blocks to specify more extensive decision mechanisms is easily controlled when handled systematically. There are two factors that provide help in this regard. We have been taking advantage of one such aid without calling specific attention to it: Indentation has been used to emphasize a program's structural components. This is a matter of voluntary programming style rather than a rule imposed by the language. For the representation of these decisions, we have chosen to align the block IF and END IF statements. The two associated actions are aligned with each other. This makes it very easy to spot (and keep track of) which ELSE goes with which IF, and provides a direct connection between the structured diagram or pseudocode and its representation as a sequence of FORTRAN statements. THEN belongs to the block IF statement.

The second source of help is an organizational one, established by FORTRAN's rules and built into the compiler. When an IF-block is nested in another one, it must be completely contained within that block. This relationship is monitored by FORTRAN in terms of the *level of nesting*.

To understand how this works, we shall imagine that the compiler, in going through a program, keeps count of the number of block IF statements and END IF statements found in that program. At the start of the program, both of these counters are set to zero. Then, whenever it finds a block IF statement, the compiler adds 1 to the block IF counter (let us call that counter NBL). Similarly, whenever it comes across an END IF, the compiler adds 1 to the END IF counter (let us call it NEI). The numbers in these counters are available to the compiler at any time, so that they can be checked constantly. Of course, all of this is automatic and is out of reach of the FORTRAN programmer. With this in mind, we can describe the rules governing nested decisions, and how these values reflect the rules:

1. The *level* of any statement in a program is defined, simply, as NBL–NEI. The particular statement being examined does *not* count in obtaining the value of NBL or NEI.
2. An IF-block cannot have a level below zero. If it does, that means that the block IF and END IF statements are not balanced. Specifically, it means that there are more END IF statements than necessary. Detection of this situation by the compiler will produce an error message.
3. At the end of a program, the level should be exactly zero, indicating that the block IF and END IF statements have been balanced. If this is not the case, there will be an error message and the program will not run.

**Example 9.4** We shall set up a hypothetical procedure with some nested decision structures in it so that we can imitate the compiler's monitoring operations. The desired decision network is a rather intricate one, as indicated in the diagram and pseudocode in Figure 9.9. Instead of transforming this diagram or pseudocode into specific FORTRAN statements, we shall make it easier to concentrate on the structure of the example by not using specific computations or variables. Instead, we shall use the word *compute* to indicate some kind of statement. Each statement (Figure 9.10) is earmarked so that we can refer to it conveniently when we discuss the program. Since we are not doing anything specific, there will be no named variables. A statement saying *"declarations"* will indicate the fact that such definitions would be present in an actual program:

We shall analyze the program's progress through these decision structures to see how the nesting level changes:

1. Just before statement 3 (the first block IF) NBL and NEI are at their initial values, so that the nesting level is at zero. Therefore, *compute1* and *compute2* are performed at that level.
2. Right after statement 3, NBL is increased to 1. Since NEI still is zero, the nesting level (NBL–NEI) now becomes 1, and all statements till the next END IF or

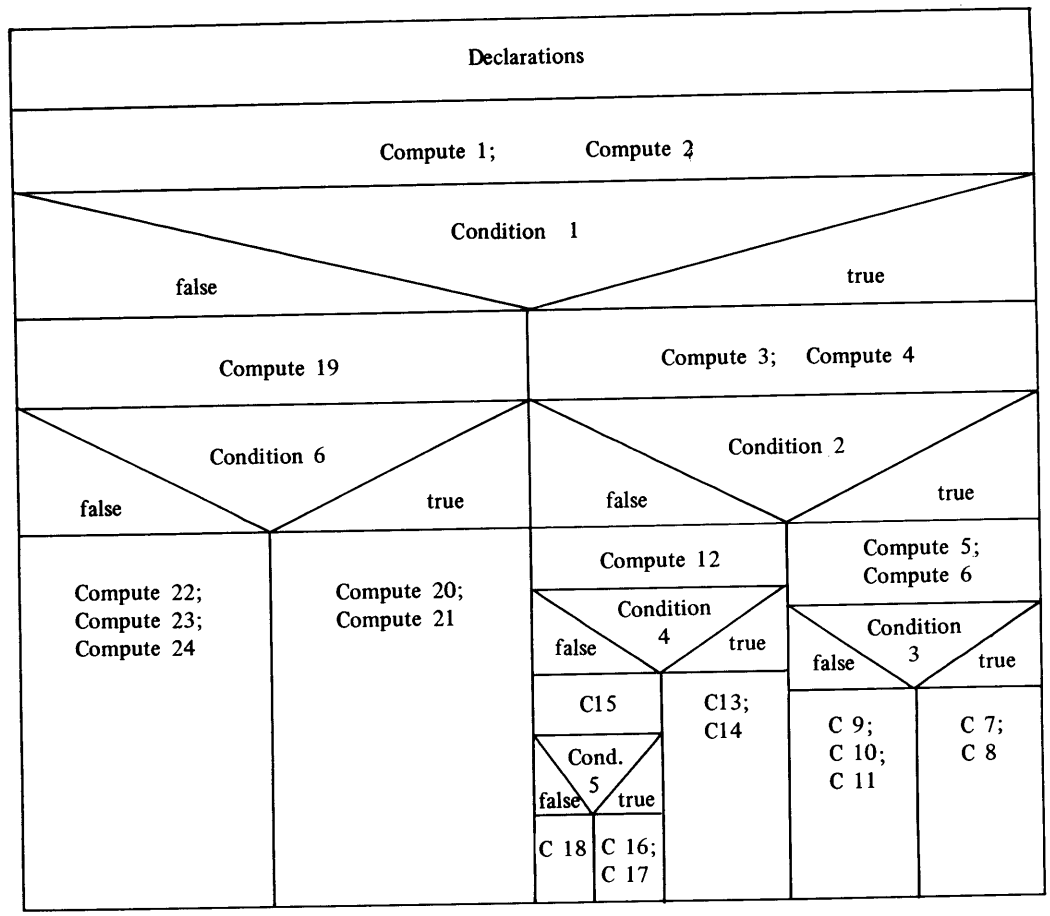



**FIGURE 9.9 (a)** Diagram for Example 9.4.

```
"Declarations."
"Compute1."
"Compute2."
if
    condition 1 is true
then
    "Compute3."
    "Compute4."
    if
        condition 2 is true
    then
        "Compute5."
        "Compute6."
        if
            condtion 3 is true
        then
            "Compute7."
            "Compute8."
        else
            "Compute9."
            "Compute10."
            "Compute11."
        endif
    else
        "Compute12."
        if
            condition 4 is true
        then
            "Compute13."
            "Compute14."
        else
            "Compute15."
            if
                condition 5 is true
            then
                "Compute16."
                "Compute17."
            else
                "Compute18."
            endif
    endif
else
    "Compute19."
    if
        condition 6 is true
    then
        "Compute20."
        "Compute21."
    else
        "Compute22."
        "Compute23."
        "Compute24."
    endif
endif
```

**FIGURE 9.9 (b)** Pseudocode for Example 9.4.

```
      PROGRAM          EX904
      declarations
C
      compute1
      compute2
C
    3 IF (condition 1) THEN
          compute3
          compute4
          IF (condition 2) THEN
              compute5
              compute6
   23         IF (condition 3) THEN
                  compute7
                  compute8
              ELSE
                  compute9
                  compute10
                  compute11
   24         END IF
          ELSE
              compute12
   33         IF (condition 4) THEN
                  compute13
                  compute14
              ELSE
                  compute15
   43             IF (condition 5) THEN
                      compute16
                      compute17
                  ELSE
                      compute18
   44             END IF
   34         END IF
   14     END IF
      ELSE
          compute19
          IF (condition 6) THEN
              compute20
              compute21
          ELSE
              compute22
              compute23
              compute24
          END IF
    4 END IF
```

**FIGURE 9.10** Structure of Example 9.4.

block IF statement are at level 1. Thus, *compute3* and *compute4* are at level 1.

3. Statement 13 brings the level to 2 (NBL=2, NEI=0), and *compute5* and *compute6* are at that level.
4. Statement 23 increases the level, so that *compute 7* through *compute11* operate at level 3.
5. The END IF at statement 24 reduces the level to 2 (NBL=3, NEI=1), thereby placing *compute12* at level 2.
6. Statement 33 brings NBL–NEI to 3 again, and that level applies to *compute13* through *compute15*.
7. Statement 43 changes the level to 4 (NBL=5, NEI=1), thereby including *compute16*, *compute17* and *compute18*.
8. The END IF in statement 44 reduces the level to 3.
9. The END IF in statement 34 reduces the level to 2.
10. The END IF statement 14 reduces the level to 1, so that *compute19* operates at that level.
11. Statement 53 raises the level to 2 again, and *compute20* through *compute24* are at that level.
12. Statement 54 brings the level back to 1.
13. Statement 4 closes out the entire decision structure, with the final level properly at zero.

Now that the levels have been defined, we can see what they mean in terms of how the actions relate to the various tests:

**Table 9.1** IF—Levels for the Statements in Example 9.4

| *Statement* | *Level* | *Statement* | *Level* |
|---|---|---|---|
| 3 | 0 | 43 | 3 |
| compute3 | 1 | compute16 | 4 |
| compute4 | 1 | compute17 | 4 |
| 13 | 1 | compute18 | 4 |
| compute5 | 2 | 44 | 4 |
| compute6 | 2 | 34 | 3 |
| 23 | 2 | 14 | 2 |
| compute7 | 3 | compute19 | 1 |
| compute8 | 3 | 53 | 1 |
| compute9 | 3 | compute20 | 2 |
| compute10 | 3 | compute21 | 2 |
| compute11 | 3 | compute22 | 2 |
| 24 | 3 | compute23 | 2 |
| compute12 | 2 | compute24 | 2 |
| 33 | 2 | 54 | 2 |
| compute13 | 3 | 4 | 1 |
| compute14 | 3 | next statement | 0 |
| compute15 | 3 | | |

1. Since the introduction of test condition 1 places the decision structure at level 1, *the associated action or actions take place at level 1 or above.* Another way of saying this is that the action includes everything after the block IF statement up to the next ELSE or END IF statement that is at the same level as that block IF statement. Applying that guideline to our example, we see that there are two such alternative actions, expressed by two IF-blocks:
   1. The block starting with *compute3* and ending just before statement 14;
   2. The block starting with *compute19* and ending just before statement 54.
2. The test for condition 2 occurs as part of the THEN action resulting from condition 1. Its outcome triggers one or two alternative actions, each of which occurs at level 2 or above:
   1. The block starting with *compute5* and ending at the ELSE just ahead of *compute12*.
   2. The block starting with *compute12* and ending just before statement 34.
3. The test for condition 3 is performed as part of the action taken if condition 2 is . TRUE. , and the outcome of *that* test causes one of two actions to take place:
   1. The IF-block consisting of *compute7* and *compute8*, or
   2. The IF-block consisting of *compute9* up to statement 24.
4. Condition 4, tested as part of the ELSE activity stemming from condition 2, also has two alternative activities. Each of these is performed at level 3 and above:
   1. If condition 4 is . TRUE. , the resulting activity includes *compute13* and *compute14*.
   2. Otherwise, the activity that is performed includes everything from *compute15* up to statement 34.
5. Part of the action resulting from a . FALSE. outcome of condition 4 is to test condition 5. Two activities are possible as a result:
   1. A successful outcome causes *compute16* and *compute17* to be performed.
   2. An outcome of . FALSE. results in *compute18* being processed.

```
      PROGRAM
      declarations
C
      compute1
      compute2
C
   3  IF (condition 1) THEN
      compute3
      compute4
  13  IF (condition 2) THEN
      compute5
      compute6
  23  IF (condition) 3) THEN
      compute7
      compute8
      ELSE
      compute9
      compute10
      compute11
  24  END IF
      ELSE
      compute12
  33  IF (condition 4) THEN
      compute13
      compute14
      ELSE
      compute15
  43  IF (condition 5) THEN
      compute16
      compute17
      ELSE
      compute18
  44  END IF
  34  END IF
  14  END IF
      ELSE
      compute19
  53  IF (condition 6) THEN
      compute20
      compute21
      ELSE
      compute22
      compute23
      compute24
  54  END IF
   4  END IF
```

**FIGURE 9.11** Structure of Example 9.4 Shown Without Indentation.

6. This leaves us with condition 6, a test that is performed as part of the ELSE action resulting from a . FALSE. outcome for condition 1. The two alternative actions are:
   1. Perform *compute20* and *compute21* if condition 6 is . TRUE. .
   2. Perform *compute22* through *compute24* if condition 6 is . FALSE. .

The entire decision structure is summarized in Table 9.1, where the level is shown for each statement. Thus, the levels give us a clear indication as to exactly which actions will be performed in response to which tests, and what the extent of each action will be.

Now, the value of indenting the various levels of IF-blocks can be seen more dramatically by contrasting this form with exactly the same statements written in straight vertical order. This is shown in Figure 9.11.

## 9.2 THE CASE CONSTRUCTION

By combining the facilities discussed in the previous sections, we can build enormously powerful structures for using tests to select directions for further processing. A basic type of situation still to be considered is one in which a single test could have more than two outcomes. That is, instead of asking, "Is this condition true or false?" we may want to ask, "Which of these six conditions is true?" Once the appropriate one has been selected, the program is to follow an action (out of six available choices) appropriate for that condition.

Strictly speaking, it is possible to construct this type of decision structure by going through a series of true-false tests using block IF statements. In fact, we shall be doing something like that. However, by treating this situation as a separate type instead of looking at it as an extension of the IF-THEN-ELSE construction, it will be easier to take advantage of some organizational conveniences provided by the language. Moreover, it will be possible to set up a simple structure that gets bigger with more choices, but does not get more complicated.

A multiple-choice type of decision is called a CASE *construction*. It can be represented as shown in Figure 9.12. Regardless of whether we use pseudocode or N-S

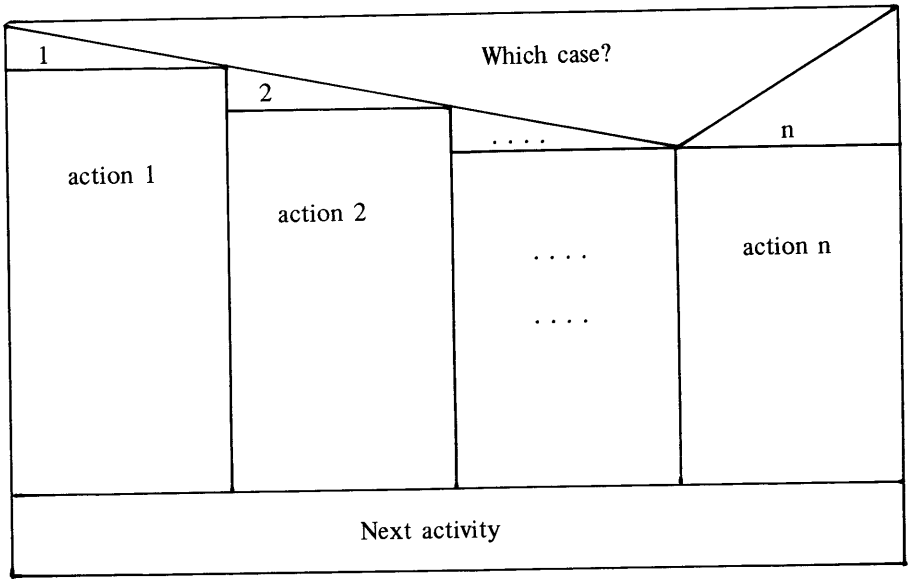



**FIGURE 9.12 (a)** The CASE Construct.

**FIGURE 9.12 (b)** Pseudocode Representation of the CASE Construct.

diagrams, note that each action is shown clearly as a single unit. This possibility is emphasized by the fact that the choices are numbered; only one action is selected while the others are ignored. Once the selected action is completed, there is a recombination of pathways to a single sequence of events. FORTRAN's capabilities make it convenient to deal with such constructs, as we shall see in the rest of the chapter.

### 9.2.1. The ELSE IF Statement

FORTRAN 77 makes a specific point of introducing an ELSE IF statement (there is no separate THEN IF statement) as a handy way of specifying CASE constructs. This statement is used in conjunction with a block IF statement as follows:

```
IF (condition) THEN
   statement
   statement          IF-block
        .
        .
   statement
ELSE IF (condition) THEN
   statement
        .
   statement          IF-block
   statement
        .
   ELSE
        .
END IF
```

This construction can continue to any level, with as many ELSE IF statements being added as necessary to take care of the entire range of choices. An example will demonstrate that this is as simple as it sounds.

**Example 9.5** Windowledge and Co. is a brokerage house selling stocks and bonds to the investing public. The fee charged for each sale is based on the amount of the sale in accordance with the following

schedule:

| *Amount of Sale* | *Windowledge's Commission* |
|---|---|
| <$100.00 | $15.00 |
| $100.00–$499.99 | $20.00 |
| $500.00–$999.99 | $20.00+3% of sale over $500 |
| $1000.00 and up | $35.00+2.5% of sale over $1000 |

Each sale is recorded on an input line that contains the customer number, customer name, number of shares, and price per share. We shall design a program that prints the customer name, along with the

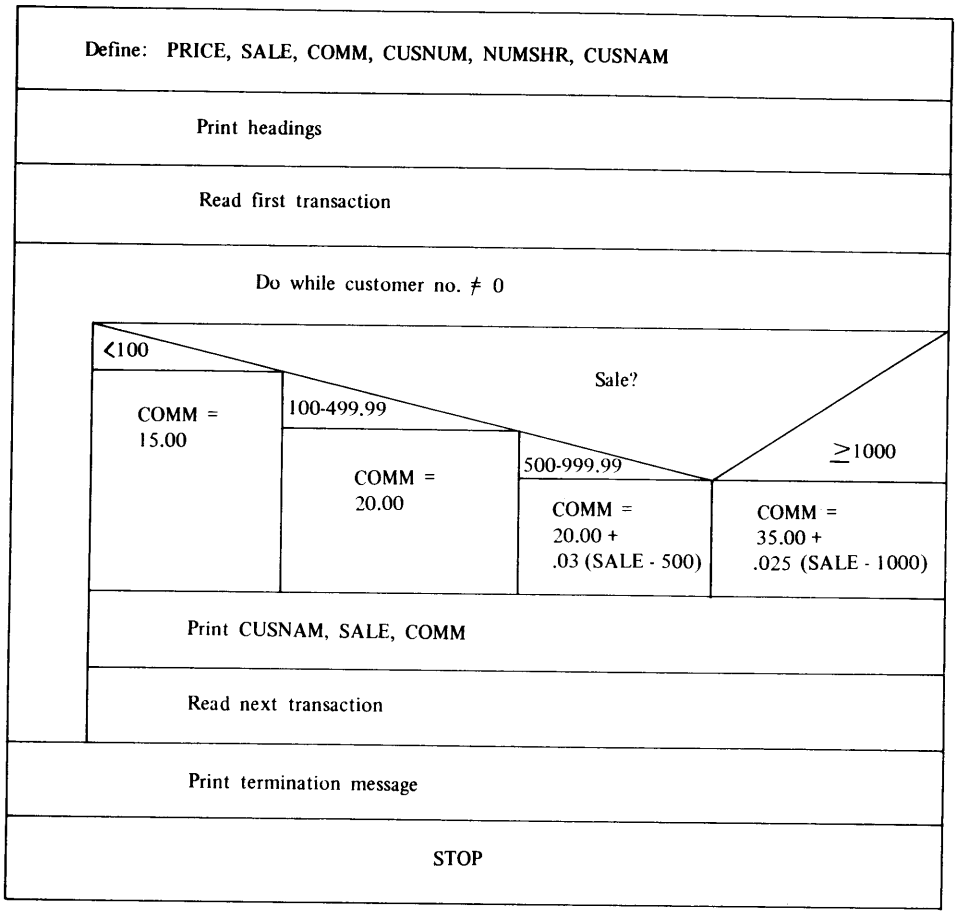



**FIGURE 9.13 (a)** Diagram for Example 9.5.

```
"Define PRICE,SALES,COMM,CUNSUM,NUMSHR,CUSNAM."
"Print the headings."
"Read the first customer's transaction."
while customer number is not equal to zero:
    case SALE of
    case1:
        "Commission = 15.00."
    case2:
        "Commission = 20.00."
    case3:
        "Commission = 20.00 + 3% of sales over 500.00."
    case4:
        "Commission = 35.00 + 2.5% of sales over 1000.00."
    endcase
    "Print CUSNAM, SALE, COMM."
    "Read the next customer's transaction."
endwhile
"Print termination message."
"Stop."
```

**FIGURE 9.13 (b)** Pseudocode for Example 9.5.

```
************************************************************
*                        EXAMPLE 9.5                       *
************************************************************
*   THIS PROGRAM ILLUSTRATES THE USE OF ELSE IF TO         *
*   BUILD A SIMPLE CASE CONSTRUCTION. ONE OF FOUR          *
*   COMMISSION FORMULAS IS SELECTED BASED ON THE VALUE     *
*   OF SALE (I.E., PRICE*NUMSHR).                          *
*   PRICE:      PRICE PER SHARE                            *
*   SALE:       AMOUNT OF SALE                             *
*   COMM:       COST OF SALE (BROKER'S COMMISSION)         *
*   CUSNUM:     CUSTOMER'S I.D. NUMBER                     *
*   NUMSHR:     NUMBER OF SHARES IN THIS SALE              *
*   CUSNAM:     CUSTOMER'S NAME                            *
************************************************************
      PROGRAM          EX905
      IMPLICIT         NONE
      REAL*4           PRICE,SALE,COMM
      INTEGER*2        CUSNUM,NUMSHR
      CHARACTER        CUSNAM*25

      PRINT *, 'NAME','AMT OF SALE','COMMISSION'
      PRINT *, '     '
      PRINT *, 'ENTER THE DATA FOR THE FIRST CUSTOMER'
      READ *, CUSNAM,CUSNUM,NUMSHR,PRICE

      DO WHILE (CUSNUM .NE. 0)
         SALE = PRICE * NUMSHR

*-----HERE IS THE CASE CONSTRUCTION. WE ARE STARTING WITH COMM
*-----INITIALIZED TO ZERO TO GUARANTEE THAT THERE IS A KNOWN
*-----VALUE IN IT.

         COMM = 0.0
         IF (SALE .LT. 100.0) THEN
           COMM = 15.00
         ELSE IF (SALE .GE. 100.0 .AND. SALE .LT. 500.0) THEN
                COMM = 20.00
              ELSE IF (SALE .GE. 500.0 .AND. SALE .LT. 1000.0) THEN
                     COMM = 20.00+0.03*(SALE-500.0)
                   ELSE
                     COMM = 35.00+0.025*(SALE-1000.0)
         END IF

         PRINT *, CUSNAM,SALE,COMM
         PRINT *, 'ENTER DATA FOR THE NEXT CUSTOMER'
         READ *, CUSNAM,CUSNUM,NUMSHR,PRICE
      END DO

      PRINT *, '     '
      PRINT *, 'END OF RUN. NORMAL TERMINATION.'
      STOP
      END
```

**FIGURE 9.14** Statements for Example 9.5.

amount of the sale and the commission. A customer number of zero will stop the run. The diagram and pseudocode for the program are shown in Figure 9.13 and the statements are in Figure 9.14. The heart of the program is the CASE construction that selects the proper commission formula for each set of input values. Note that each test is spelled out completely, even though the ELSE IF construction makes it unnecessary. For instance, a value of . TRUE. for the first test (i.e., a sale less than $100.00) sets the commission at $15.00 and immediately the program goes to statement 100, bypassing the other tests. (Incidentally, statement 100 is labeled only because of the reference to it in this discussion.) The ELSE IF statement guarantees that the only way the second test can be performed is for the first test to come out . FALSE. , so that the sale is already "known" to be $100.00 or more. The (unnecessary) test is included for clarity, to show exactly how the commission formula is selected.

There is one important precaution to keep in mind when designing a CASE construct for a particular situation: the choices specified in the construct must be complete. That is, the programmer must make sure that there can be no situation which does not match one of the choices. Often, this means that there needs to be a "catch-all" activity that the program will perform in the event that all of the other tests produce a . FALSE. outcome. In Example 9.5, such an escape hatch has been included for illustration, even though it was not needed, by setting COMM to zero before the CASE construct is entered for each input set. In this way, COMM is always assured of having a value. For more complex situations, it may be more appropriate to include an additional activity as the last part of the construct so that it says, in effect, "Here are *n* alternative activities. Choose *one* of them by going through these tests. If all of the tests fail, perform this activity instead." This process is illustrated below. The pseudocode representation also may show this type of construction, as illustrated in Figure 9.15.

```
IF (condition 1) THEN
   action 1
ELSE IF (condition 2) THEN
        action 2                    n regular choices
     ELSE IF (condition 3) THEN
          action 3
                .
                .
                .
        ELSE IF (condition n) THEN
             action n
          ELSE
             catch-all action              escape hatch
END IF
```

```
case description of selector of
case1:
    "Action1."
case2:
    "Action2."
case3:
    "Action3."
case4:
    "Action4."
        ..
        ..
casen:
    "Actionn
else
    "Escape action."
endcase
```

**FIGURE 9.15** Pseudocode for CASE Construction with an Escape Hatch.

### 9.2.2 More Extensive CASE Constructs

The basic construction shown in the previous section serves equally well in situations requiring more extensive actions.

**Example 9.6** To demonstrate the idea that the CASE statement retains its simplicity when its processing expands, we shall take another look at Windowledge and Company. This time, the company wants to compute something it calls an "investment profile index" (IPX). As with commisssions, an IPX value is determined in one of four ways, depending on the amount of the sale:

| *Sale* | *IPX* |
|---|---|
| <$100.00 | (NUMSHR/PRICE)**.8+3.12*(PRICE-2*NUMSHR) |
| 100.00–499.99 | ((NUMSHR-1)/PRICE)**.67+4.14*(PRICE-1.8/NUMSHR) |
| 500.00–999.99 | (PRICE**.214)*(NUMSHR-1) |
| 1000.00 AND UP | (NUMSHR/PRICE+2.1*LOG(SALE))**1.16 |

Output for each sale is to report the IPX in addition to its previous contents. The revised program (Figure 9.16) is unchanged in structure.

### 9.2.3 Another Way to Implement the CASE Construct

FORTRAN has an additional facility that offers another way to represent a CASE construct. It is not necessarily "better" or "worse" than the use of IF-THEN-ELSE IF sequences. For some programmers, it represents a more awkward approach because it requires additional statements; others prefer it because, for them, it is a more direct reflection of the basic construct.

The basis for this technique is FORTRAN's *computed* GO TO statement. It is called "computed" because, unlike the unconditional GO TO, its destination is not fixed. Rather, it is selected from a choice of predefined alternatives, and the choice can be computed by the program at any time before the GO TO is executed. The general form is

```
GO TO (label1,label2,label3,.....,labeln),i
```

*label1* is the label attached to the first statement of the first alternative activity, *label2* indicates where the activity for the second choice starts, and so on. The i after the comma outside the parentheses (the comma after the closing parenthesis is optional in FORTRAN 77, required in earlier FORTRANs) is the name of an integer variable whose value at that instant indicates which of the destinations the program is to select. For example, the computed GO TO statement

```
GO TO (32,26,41,17,19),SWVAR
```

specifies five destinations from which to choose. The choice will depend on SWVAR (which we assume was defined earlier in the program by an INTEGER statement, and was given a value in some way). If SWVAR's value is 1, then the next statement executed by the program would be statement number 32; a value of 2 in SWVAR sends the program to statement 26, and so on. Thus, in this particular example, we would expect SWVAR to have a value of either 1, 2, 3, 4, or 5. A well-designed program, then, would include statements to test SWVAR before allowing the program to reach the computed GO TO. If the program happens to execute the computed GO TO with SWVAR's value being out of range (i.e., less than 1 or greater than 5 in this case), the program simply will execute the statement immediately following the GO TO, as if the GO TO were not there.

There is no particular limit to the number of destinations that can be specified in a computed GO TO. Nor is it necessary to have a different destination for each alternative. For instance, the statement

```
GO TO (16,88,88,24,35,16),CHOICE
```

```
C*********************************************************
C                  EXAMPLE 9.6                            *
C*********************************************************
C   ALL THE DEFINITIONS ARE THE SAME AS IN EXAMPLE IX.5 *
C   EXCEPT FOR THE ADDITION OF IPX, THE INVESTOR PROFILE*
C   INDEX, COMPUTED ONE OF FOUR WAYS, DEPENDING ON THE  *
C   SIZE OF THE SALE.                                   *
C*********************************************************
      PROGRAM             EX906
      IMPLICIT            NONE
      REAL*4              PRICE,SALE,COMM,IPX
      INTEGER*2           CUSNUM,NUMSHR
      CHARACTER           CUSNAM*25

      PRINT *, 'NAME','AMT OF SALE','COMMISSION','INV PROF INDEX'
      PRINT *, '     '
      PRINT *, 'ENTER DATA FOR THE FIRST CUSTOMER'
      READ *, CUSNAM,CUSNUM,NUMSHR,PRICE

      DO WHILE (CUSNUM .NE. 0)
         SALE = PRICE * NUMSHR
         COMM = 0.0
         IPX = 0.0

         IF (SALE .LT. 100.0) THEN
           COMM=15.0
           IPX=(NUMSHR/PRICE)**0. + 3.12*(PRICE-2*NUMSHR)
         ELSE IF (SALE .GE. 100.0 .AND. SALE .LT. 500.0) THEN
                COMM=20.00
                IPX=((NUMSHR-1)/PRICE)**0.67+4.14*(PRICE-1.8/NUMSHR)
              ELSE IF (SALE .GE. 500.0 .AND. SALE .LT. 1000.0) THEN
                     COMM=20.0+0.03*(SALE-500.0)
                     IPX=(PRICE**0.214)*(NUMSHR-1)
                   ELSE
                     COMM=35.0+0.025*(SALE-1000.0)
                     IPX=(NUMSHR/PRICE+2.1*LOG(SALE))**1.16
         END IF

         PRINT *, CUSNAM,SALE,COMM,IPX
         PRINT *, 'ENTER DATA FOR THE NEXT CUSTOMER'
         READ *, CUSNAM,CUSNUM,NUMSHR,PRICE
      END DO

      PRINT *, '     '
      PRINT *, 'END OF RUN. NORMAL TERMINATION.'
      STOP
      END
```

**FIGURE 9.16** Statements for Example 9.6.

is perfectly legitimate, as long as CHOICE is an integer variable and there are statements in the program with the labels 16, 88, 24, and 35. With this setup, the program will continue at statement 16 when CHOICE is either 1 or 6, statement 88 when CHOICE is 2 or 3, 24 when CHOICE is 4, and 35 when CHOICE is 5.

The way the computed GO TO is used to develop a CASE construct is to provide a sequence of tests (simple IF statements) in which each of the alternative actions assigns a particular value to an integer variable. That variable then is used as the switching control in a single computed GO TO statement that directs the program to the proper action. Each action begins with a statement whose label matches one of the choices given in the GO TO, and each action concludes with an unconditional GO TO to a single CONTINUE statement at the end of all the actions. As we have seen earlier, the CONTINUE statement does not actually compute anything. Here, it serves as a definite punctuation symbol, indicating the conclusion of the CASE construct.

```
C*******************************************************
C                    EXAMPLE 9.7                        *
C*******************************************************
C   THIS PROGRAM IMPLEMENTS THE SAME PROCEDURE AS IN   *
C   EXAMPLE IX.6, WITH THE CASE CONSTRUCT BUILT AROUND*
C   THE COMPUTED GO TO STATEMENT.                      *
C*******************************************************

      PROGRAM          EX907
      IMPLICIT         NONE
      REAL*4           PRICE,SALE,COMM,IPX
      INTEGER*2        CUSNUM,NUMSHR,SWITCH
      CHARACTER        CUSNAM*25

      PRINT *,'NAME','AMT OF SALE','COMMISSION','INV PROF INDEX'
      PRINT *,'      '
      PRINT *, 'ENTER INPUT DATA FOR THE FIRST CUSTOMER'
      READ *, CUSNUM,CUSNAM,PRICE,NUMSHR

      DO WHILE (CUSNUM .NE. 0)
         SALE = PRICE * NUMSHR
         SWITCH = 1

         IF (SALE .LT. 100.0)                                SWITCH=1
         IF (SALE .GE. 100.0 .AND. SALE .LT. 500.0)          SWITCH=2
         IF (SALE .GE. 500.0 .AND. SALE .LT. 1000.0)         SWITCH=3
         IF (SALE .GE. 1000.0)                               SWITCH=4
         GO TO (21,22,23,24),SWITCH

   21    COMM = 15.00
         IPX = (NUMSHR/PRICE)**0.8 + 3.12*(PRICE-2*NUMSHR)
         GO TO 29
   22    COMM = 20.00
         IPX = ((NUMSHR-1)/PRICE)**0.67 + 4.14*(PRICE-1.8/NUMSHR)
         GO TO 29
   23    COMM = 20.00 + 0.03*(SALE-500.00)
         IPX = (PRICE**0.214)*(NUMSHR-1)
         GO TO 29
   24    COMM = 35.00 + 0.025*(SALE-1000.00)
         IPX = (NUMSHR/PRICE + 2.1*LOG(SALE))**1.16
   29    CONTINUE

         PRINT *, CUSNAM,SALE,COMM,IPX
         PRINT *, 'ENTER INPUT DATA FOR THE NEXT CUSTOMER'
         READ *, CUSNUM,CUSNAM,PRICE,NUMSHR
      END DO

      PRINT *, '     '
      PRINT *, 'END OF RUN. NORMAL TERMINATION.'
      STOP
      END
```

**FIGURE 9.17** Statements for Example 9.7.

**Example 9.7** We shall rewrite the program in Example 9.6, replacing the ELSE IF sequences with an equivalent CASE construct based on a computed GO TO. Everything else remains the same, except for the definition of an additional integer variable (SWITCH) to control the GO TO. The revised version is shown in Figure 9.17. Neither the N-S diagram or pseudocode needs to be changed because the *structure* still is the same as it was for Example 9.6. Only the *implementation* of that structure changes.

## 9.3 SUMMARY

FORTRAN makes it possible to set up an endless variety of decision-making mechanisms. These are based on the performance of a test designed to have an outcome of "true" (designated in FORTRAN as . TRUE. ) or "false" (indicated as . FALSE. ). Such tests are constructed and specified by means of the IF statement, which may be used in

several forms:

**Basic Logical IF Statement:**

```
IF (condition) statement 1
statement 2
  etc.
```

**operation:** An outcome of . TRUE. causes statement 1 to be executed. This may or may not be followed by statement 2, depending on the type of statement 1. An outcome of . FALSE. causes statement 1 to be ignored, and statement 2 is executed.

**IF-THEN-ELSE Construction:**

```
IF (condition) THEN
   . .
   . .
   statements
   . .
ELSE
   . .
   statements
   . .
   . .
END IF
next statement
```

**operation:** An outcome of . TRUE. causes the group of statements between THEN and ELSE to be processed. The statements between ELSE and END IF are ignored and processing continues with "next statement." An outcome of . FALSE. causes the statements between ELSE and END IF to be processed (with those between THEN and ELSE being ignored), and processing continues with "next statement."

An extension of these forms makes possible the convenient construction of decision networks in which there are more than two choices. This type of decision mechanism is called a CASE *construct.* Two implementation techniques are summarized below:

**CASE Constructions with ELSE IF Statements:**

```
IF (condition 1) THEN
  action 1
ELSE IF (condition 2) THEN
    action 2
  ELSE IF (condition 3) THEN
      action 3
        . .
        etc.
        . .
    ELSE
END IF
next statement
```

Another way to format this construction is:

```
IF (condition 1) THEN
   action 1
ELSE IF (condition 2) THEN
     action 2
  ELSE IF (condition 3) THEN
     action 3
  . . . . . .
     etc.
  . . . . . .
ELSE
END IF
next statement
```

**operation:** Each of the tests is performed in turn, starting with condition 1. As long as the outcome of a particular test is .FALSE., the program goes on to perform the next test. When the first .TRUE. outcome is achieved, the program performs the action associated with that condition and then executes "next statement." If all the tests fail, the program acts as if the entire CASE construction were not there.

**CASE Constructions with the Computed GO TO:**

```
   IF (condition 1) M = 1
   IF (condition 2) M = 2
         ....
         ....
   IF (condition n) M = n
   GO TO (L1, L2, . . . , Ln) M
L1 action 1
    . . .
    . . .
   GO TO NN
L2 action 2
    . . .
    . . .
   GO TO NN
      ....
      ....
LN action n
   . . .
   . . .
NN CONTINUE
   next statement
```

**operation:** The series of tests selects the appropriate choice and signals that choice by setting variable M to 1, 2, 3, . . . , n. Then, the computed GO TO statement acts as a multiple switch, sending the program to destination L1 (for M=1), L2 (for M=2), etc. Each action ends by directing the program to the conclusion of the construction, and processing continues with "next statement."

Test conditions are specified as comparisons. For example,

```
( (X+Y) **1.8 .GE. Z-3.8*Y) )
```

compares the two indicated expressions. If the first expression's value is greater than or equal to that of the second, a .TRUE. outcome results; otherwise, an outcome of .FALSE. results. Five other *relational operators* (.EQ., .NE., .LT., .LE., .GT.) are available. More extensive tests, still with a final outcome of .TRUE. or .FALSE., can be constructed by combining comparisons with logical operators. The test given below,

```
(  ( (X+Y) **1.8 .GE. Z-3.8*Y) .AND. (W .LT. 1.7/Y) )
```

will produce an outcome of .TRUE. only if both comparisons are true. The .OR. operator also is available for constructing such test combinations.

**PROBLEMS**

1. Indicate the output value(s) printed by each of the independent sequences given below. Assume the following declarations and assignments:

```
REAL*4          R1, R2, R3, R4
INTEGER*2       N1, N2, N3
CHARACTER*8     C1, C2, C3*4, C5*12, BLANK*1
PARAMETER       (BLANK = ' ')
DATA            R1, R2, R3/18.0, 6.0, 2.0/N2, N3/2, 3/
1               C1, C2, C3/'FARMLAND', 'FARMHAND', 'HARM'/
```

(a)
```
R4 = 0.0
IF (R1+R2+R3 .LE. R3** (N2*N3) ) R4=R4+R2**N2
PRINT *, R4
```

(b)
```
IF (R2**N2 .GT. R1*R3) THEN
  R4 = R3*(R1+R2)
ELSE
  R4 = R3*(R1-R2)
END IF
PRINT *, R4
```

(c)
```
IF (MAX(R1,R2*R3, (R1/R2)**N3) .LT. R3**(N2+N3)) THEN
  R4 = 27.9
ELSE
  R4 = R1/(R2+R3)
END IF
PRINT *, R4
```

(d)
```
IF (C1(:4) .EQ. C2(1:4)) THEN
  C5 = C1(:3)//BLANK//C1(5:)
ELSE
  C5 = C2(:5)//'OUSE'
END IF
PRINT *, C5
```

(e)
```
R4 = R1*R2*R3
C5 = BLANK
IF (C3 .EQ. C1(2:4) THEN
  R4 = R4/N3
  C5(5:8) = C3
ELSE
  R4 = R4/(N2+N3)
  C5(2:) = C3//'LESS'
END IF
PRINT *, C5, R4
```

(f)
```
R4 = R1*R2*R3
C5 = BLANK
N1 = N2-N3
IF (R4**N1 .LE. 1.0/(R1*R2*(R3-1))) THEN
  C5(9:) = C3
  PRINT *, R4, C5
ELSE
  C5(6:10) = C3
  R4 = 0.6*R4
  PRINT *, N1, C5, R4
END IF
```

(g)
```
  R4 = 0.0
  IF (C1(:4)//C2(6:) .LT. C2(1:4)//C1(6:8)) THEN
    R4 = 2.0*R4
    DO 8 N1 = 1,5
      R4 = R4 + N1*(N2+N3)
      C5(2*N1-1:2*N1) = C1(N1:N1)
8   CONTINUE
  ELSE
    R4 = 2.0 + R4
    DO 9 N1 = 1,5
      R4 = R4+N1(N2-N3)
      C5(N1:N1) = C2(N1:N1)
      C5(N1+5:N1+5) = C2(N1+1:N1+1)
9   CONTINUE
  END IF
  C5(11:) = BLANK
  PRINT *, R4, C5
```

2. Write a sequence of FORTRAN statements to represent each of the specifications given below. Assume the following declarations:

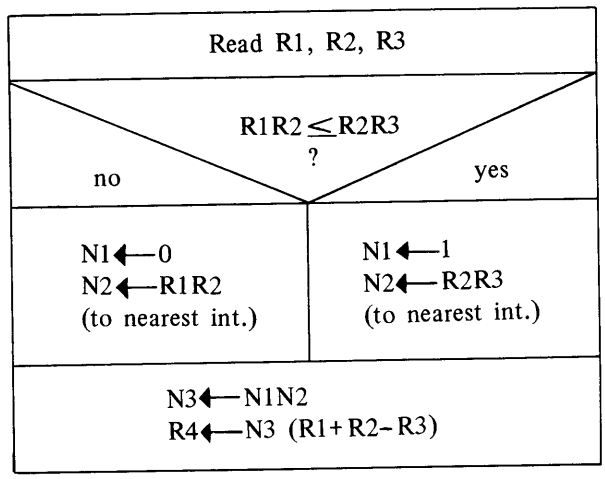
```
REAL*4          R1, R2. R3. R4
INTEGER*2       N1, N2, N3,
CHARACTER       C1*8, C2*8, C3*12
```

(a) After reading in values for R1 and R2, set N1 to 3 if R1 is greater than R2; otherwise, set N1 to 4.

(b) "Read a value for C1."
*if*
C1 has any vowels in it
*then*
"Set N1 to the number of vowels in C1."
*else*
"Set N1 to zero."
*endif*

(c) "Read values for R1, R2, R3, and R4."
*if*
the sum of R1 and R2 is greater than the sum of R3 and R4
*then*
"Place in N1 the sum of R1 and R2, rounded to the nearest integer."
"Place in N2 the sum of R3 and R4, rounded to the nearest integer."
*else*
"Place in N1 the sum of R3 and R4, rounded to the nearest integer."
"Place in N2 the sum of R1 and R2, rounded to the nearest integer."
*endif*

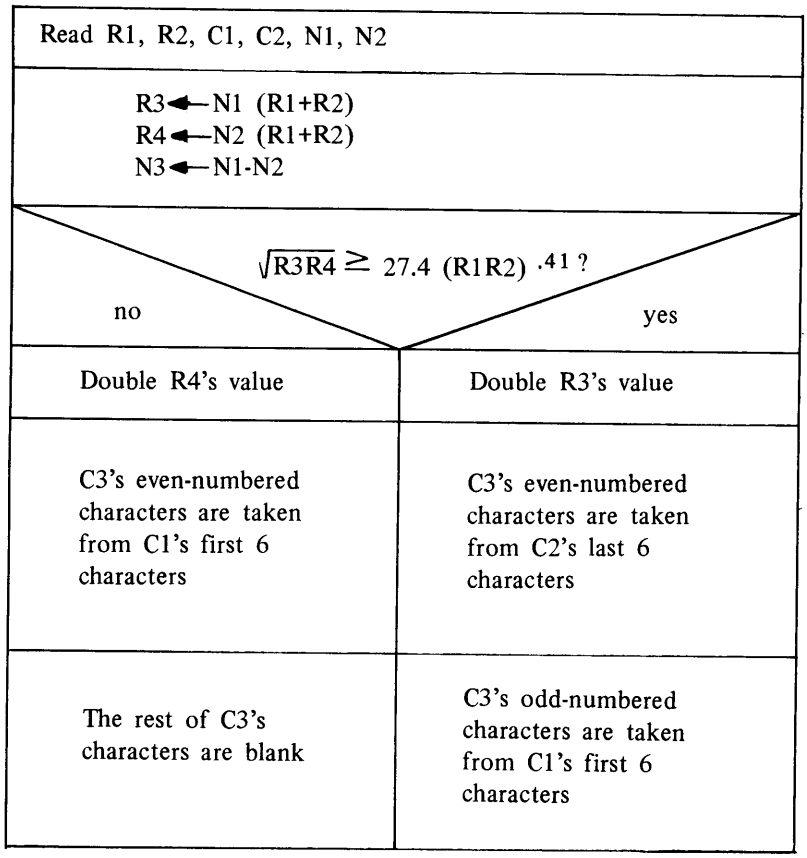
(d)




(e) Read values for R1, R2 and R3. Assign to R4 a value equal to the sum of the two largest input values.

(f) Read values for C1 and C2. assign to N1 the number of letters in C1 and C2 that are in the second half of the alphabet. Assign to N2 the number of letters in C1 and C2 that are in the first half of the alphabet.

(g) "Read R1, R2, N1 and N2."
"Read R3."
*if*
N1 is odd
*then*
"Set R4 to R3+ (R1+R2) ** (N1+N2) ."
*else*
"Set R4 to R3+ (R1+R2) ** (N2) ."
*endif*




3. Assume the following declarations and assignments:

```
REAL*4          R1, R2, R3 (6) , R4
INTEGER*2       N1, N2, N3 (6) , N4
CHARACTER       C1*8, C2*8, C3 (6) *4
R1 = 2.0
R2 = 3.0
N1 = 1
N2 = 4
C1 = 'MAKEWELL'
C2 = 'BAKERIES'
```

Indicate the values printed as a result of each of the following sequences:

(a)
```
DO N4 = 1, 6
   IF (MOD (N4, 3) .EQ. 0)  THEN
     R3 (N4) = R1 + N4*R2** (N1*N2)
     N3 (N4) = N4* (N1-N2)
   ELSE
     N3 (N4) = (N4-1) * (N2-N1)
     R3 (N4) = R2 + N4*R1**N3 (N4)
   END IF
END DO
PRINT *, R3, N3
```

(b)
```
R4 = SQRT ( (R1/R2) **N1)
IF (R4 .GT. R1 .AND. R4 .LT. R2)  THEN
  N4 = R4* (N1+2*N2)
ELSE
  N4 = R4* (2*N1+N2)
END IF
PRINT *, R4, N4
```

```
(c) R4 = 0.0
    DO N4 = 1,3
      N3(N4) = MOD(R1+R2),N4)
      IF (MOD(N3(N4),2) .EQ. 0)  THEN
        R3(N4) = (R1+R2)**N3(N4)
        R3(7-N4) = (R1-R2)**N3(N4)
        N3(7-N4) = MOD(N3(N4),N4))
      ELSE
        R3(N4) = N3(N4)*(R1+R2)
        R3(7-N4) = 0.5*R3(N4)
        N3(7-N4) = 2*N3(N4)
      END IF
    R4 = R4 + 0.5*(R3(N4)+R3(7-N4))
    END DO
    PRINT *, R4
(d) DO N4=1,6
      N3(N4) = N4*(N1+N2)
      IF (MOD(N4,2) .NE. 0)  THEN
        R4 = (R1+R2)**N3(N4)
      ELSE
        R4 = (R1-R2)**N3(N4)
      END IF
      IF (ABS(R4) .LT. R2)
    1              .OR.
    2     (C1(N4:N4) .GE. C2(N4:N4))  THEN
        R3(N4) = R4
        C3(N4) = C2(N1:)
      ELSE
        R3(N4) = 2*R4
        C3(N4) = C1(N1:)
      END IF
    END DO
    PRINT *, N3
    PRINT *, R3
(e) IF (C1(:2) .LE. C2(:2) .AND. (R1+R2)**2 .LE. 2*R1*R2)  THEN
        R4 = 0.0
        DO 8 N4=1,6
          N3(N4) = N4
          IF (N4 .LE. 4)  THEN
            R3(N4) = (R1+R2)**N4
            C3(N4) = C1(N4:N4)//C2(N4:N4)
    1                //C1(N4+1:N4+1)//C2(N4+1:N4+1)
          ELSE
            R3(N4) = 2*(R1+R2)**N4
          END IF
          R4 = R4+R3(N4)
    8   CONTINUE
      ELSE
        R4 = 1.0
        DO 12 N4=1,6
          N3(N4) = N4/2
          IF (N4 .LE. 4)  THEN
            R3(N4) = (R1+R2)**N4
            C3(N4) = C1(N4:N4+1)//C2(N4:N4+1)
          ELSE
            R3(N4) = 3*(R1+R2)**N4
          END IF
          R4 = R4*R3(N4)
```

```
12  CONTINUE
    END IF
    PRINT *, N3
    PRINT *, R3
    PRINT *, C3
    PRINT *, R4
```

4. Write FORTRAN statements for each of the specifications given below. Assume the following declarations:

```
REAL*4          R1, R2, R3 (8) , R4, R5 (8)
INTEGER*2       N1, N2, N3 (8) , N4)
CHARACTER       C1*8, C2*8, C3 (8) *4
```

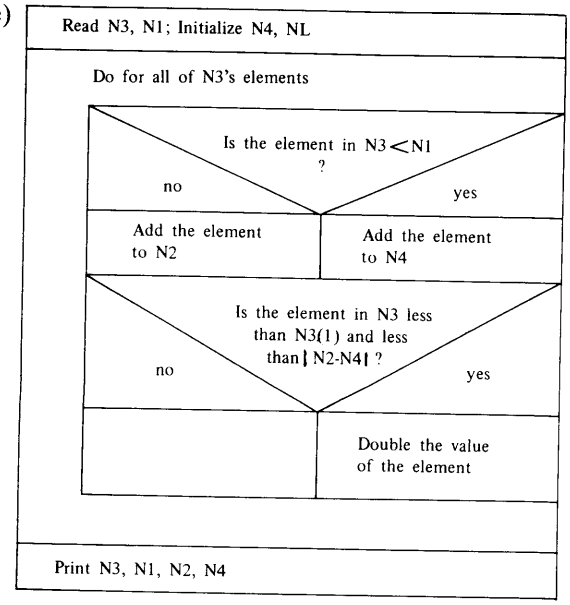
(a) Read R1, R2, N1, and N2. If N1 and N2 are both odd numbers, compute R4 as R1**N1*R2**N2; otherwise compute R4 as R1**N2*R2**N1.

(b) Read R1 and R2. Store in N1 the number of odd digits (1, 3, 5, 7, 9) in the integer portions of R1 and R2. Store in N2 the number of even digits (0, 2, 4, 6, 8) in the integer portions of R1 and R2. (High order zeros do not count. For example, the value 001070.027 has two zeros in its integer portion.) If either N1 or N2 is greater than 3/4 of their sum, R1 is set to 10.5; otherwise it is set to −10.5.

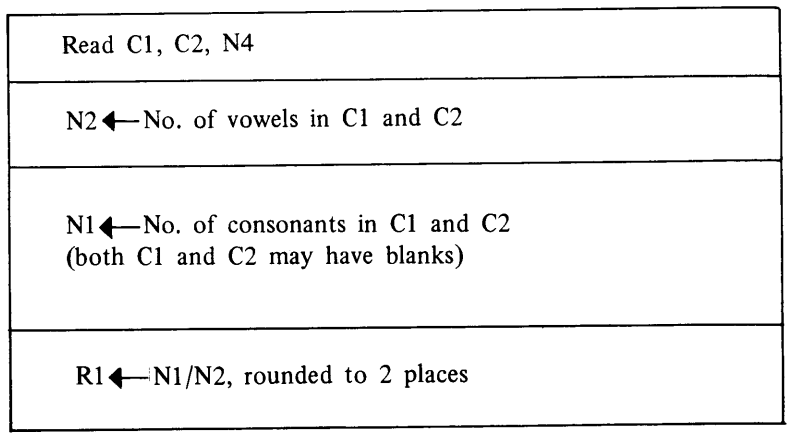
(c) "Read R1, R2, R3, and R4."
*if*
R1's value is between R2's and R4's value
*then*
"Multiply by R1 all of R3's elements whose values are larger than R1's."
*else*
"Multiply by (R2/R4) all of R3's elements whose values are less than R1's."
*endif*

(d) "Read R3, R1, and R2."
*do* for all elements in R3:
*if*
an element in R3 is less than or equal to R1 as well as R2
*then*
"Store in R5's corresponding element the sum of the corresponding element from R3, R1, and R2."
*else*
"Store in R5's corresponding element the value of the corresponding element from R3 less half of R1's value."
*endif*
*enddo*

(e)



(f)




(g) "Read R1 and R3."
"Initialize N1 and N2."
"N1 = the number of elements in R3 whose values are less than R1."
"N2 = the number of elements in R3 whose values are greater than R1."
"R2 = N1/N2, rounded to 2 places."
*if*
    there are no elements in R3 equal to R1
*then*
    "Set N4 to zero."
*else*
    "Set N4 to 1."
*endif*
*if*
    at least three of R3's elements are greater than R1
*then*
    ''R4 = (N4+2)/R2.''
*else*
    ''R4 = (N4−2)/R2.''
*endif*
"Print N1, N2, N4, R2, R4, each variable or array on a separate line."

5. Write a complete program to perform the following processing: Each input line contains a three-digit positive integer value (INTVAL). If the rightmost (third) digit is equal to the sum of the other two digits, that number is to be printed on a separate line along with the message THIS IS A SPECIAL NUMBER. If not, there is to be no output for that number. Thus, 246 and 729 are special numbers while 264 and 381 are not. A run may consist of any number of input values. After the last value is processed, the program is to print the number of values that met the requirement described before (SPECLS) and the number of values (ORDNRS) that did not. Here are some suggested data values:

303
627
718
339
336
347
112
end-of-run signal (up to you)

6. Here is a more challenging version of the previous problem: We are reading positive integer values as before (one at a time, using a variable named INPVAL). This time, however, the number of digits is not fixed. If a particular value reads the same way in either direction, the program is to print the value, along with the message THIS IS A SYMMETRICAL NUMBER. If not, there is to be no output for that input value. Thus, 6116, 747, 2002, 36463 and 88 are symmetrical numbers while 4141, 30, 32732, and 9 are not. As in the previous problem, there may be any number of input values in a given run. After the last value

has been processed, the program is to print the number of symmetrical values (NUMSYM), their sum (SUMSYM), the total number of values (TTLVAL) and *their* sum (SUMTTL). Finally, the program is to print the ratio of SUMSYM to SUMTTL, rounded to three decimal places. Here are some suggested input values:

```
9009
33
9090
30103
26326
442442
424424
6
end-of-run signal (up to you)
```

7. It had to happen: four hotshots have gotten together and formed Astute Dating, a nationally franchised (no less) computerized dating service. ("Let Astute compute your next cute beaut.") Now they need a program to produce dating matches. The program is to operate as follows: Each client's data are prepared on a line containing a 5-digit integer client number followed by yes-or-no answers to 20 Significant Questions. A "yes" is represented by the integer 1 and a "no" is represented by the integer 0. Thus, a typical set of client input data might say

   ```
   30244    1,1,1,0,1,0,0,1,0,1,1,1,0,0,1,0,0,0,1,0
   ```

   Now, suppose that one of Astute's female clients requests a report from the matching service in which potentially suitable males are identified and listed. To do this, the program would read and store the requestor's data. Then, those data would be compared, in turn, with each of the input sets from Astute's Male Collection. For each question answered "yes" by both parties, the program would add 1 to the CompuLove Score Index (CMPNDX). If such a comparison between two sets of data produces an Index of at least 15, the program prints the following output line:

   ```
   NUMBER nnnnn IN OUR COLLECTION MATCHES WITH nn
   ```

   nnnnn is the identifying number from the input and nn is the matching score. The output starts with a heading line that says

   ```
   COMPULOVE REPORT PREPARED ESPECIALLY FOR nnnnn
   ```

   After the last set in the collection has been processed, the program prints NUMYES, the number of matches found in the collection. You may assume that there is no client number 0.

8. Here is a more challenging version of the previous problem, brought about by Astute's runaway success. The fabulous franchiser is looking to refine the matchmaking by introducing a slight complication in the way people's input data are compared: There still are 20 yes-or-no questions, but studies have shown that they are not all equally important. Consequently, it is necessary to assign a different weight to each question depending on its significance in the overall matching process. But wait. (Weight? No; wait): It is not as easy as that. Astute's researchers have worked out the following weighting schedule:

   (a) Questions 1, 2, 6, 11, 14, 16, and 19 have weights of 1 each. That is, an answer of "yes" for any of these adds 1 to the Score Index.
   (b) Questions 3, 12, and 18 have weights of 2 each.
   (c) Questions 5 and 15 have weights of 3 each.
   (d) Question 7 has a weight of 5. (This is some question.)
   (e) Questions 4 and 9 have weights of 3 each if the answer to question 5 is "yes"; otherwise their weights are 2 each.
   (f) Questions 8, 10 and 17 have weights of 4 each if the answers to questions 2, 11, and 15 all are "yes."
   (g) Questions 13 and 20 have weights of 2 each if the answer is "yes" to either 1, 16, or 18.

   A total score of 36 or above is considered a successful match, for which the program produces the same output as described in the previous problem. All the other requirements are the same as those given in that problem.

9. The Golden Glut is a limited-menu restaurant specializing in mediocre food, a fact hidden only with partial success by a bewildering collection of prefab cutesy decorations. One of the Glut's popular specialties is a prepackaged Dinner for Eight (reservations two days in advance, please) whose basic cost is $62.00. For that amount, each Glutperson (as the diners are affectionately known) gets a salad, an entree, a beverage, and a dessert. This sounds simple enough, but it gets a little involved because of the conditions under which the items are selected:

   (a) Everybody gets the same salad.
   (b) There are three kinds of entrees (types 1, 2, and 3). The fixed price entitles the dining parties to eight entrees, sure enough, but at least three of them must be type 1, at least 2 of them must be type 2, and at least 1 of them must be type 3. As long as these requirements are met, anything else is flexible.
   (c) Everybody gets any of the three beverages on the menu. (Well, limited is limited.)
   (d) There are two types of desserts (types 1 and 2). Four of the desserts must be from each type.

   Of course, the diners have the right to deviate from these rules, but it will cost them:

   (e) If less than three type 1 entrees are ordered, there is a $2.00 penalty for each one less. If less than two type 2 entrees are ordered, there is a $1.75 penalty for each one less.
   If there are no orders for type 3 entrees, there is a $1.50 penalty.
   (f) If more than eight beverages are ordered, there is an additional charge of $1.00 each for the first five and $0.85 for each one beyond that.
   (g) Type 1 desserts are $1.50 each and type 2 desserts are $1.25 each. If diners order less than four desserts of either type, there is no credit for the unordered dessert. For instance, if a party of eight decides on 6 desserts of type 1 and only 2 desserts of type 2, they are charged $3.00 extra for the two extra type 1 desserts. (If you are going to say that is how the cookie crumbles, don't say it.)

   The Glut would like a program that computes and prints a total amount to be billed for each party of eight. Input (for each party) consists of eight lines. Each line contains the name (up to 15 characters) under which the reservation was made (RESERV), the type of entree (ENTREE), recorded as 1, 2, or 3, the number of beverages (NUMBEV), and the type of dessert, indicated as 1 or 2. For each party, the program is to print five output lines: The name, total surcharge for entrees, total surcharge for beverages, total surcharge for desserts, and the total amount of the bill. Each run may contain any number of input sets (i.e., dinner parties). Leave a blank line between output sets and select your own method of signalling the end of the run.

10. Write a program that computes and prints a date (month/day/year) given a starting date and an elapsed time period. That is, each input set consists of six integer values: Starting month (MSTART, starting day (DSTART), starting year (YSTART), number of elapsed months (MLAPSE), number of elapsed days (DLAPSE)and number of elapsed years (YLAPSE). These are used to compute a final month (FMONTH), final day (FDAY) and final year (FYEAR). Each input set produces five lines of output. For instance, starting values of 3, 17, and 1981 (month, day, and year) and elapsed values of 6, 10, and 4 (months, days and years) will give the following output:

```
STARTING DATE:          3/17/81
ELAPSED TIME:      6      MONTHS
                  10      DAYS
                   4      YEARS
FINAL DATE:             9/27/85
```

   Leave a blank line between output sets and end the run with a signal of your own choice. The following assumptions apply to this program:

   (a) All dates will be in the 20th century.
   (b) MLAPSE will never exceed 12, DLAPSE will never exceed 31, and YLAPSE always will be sized so as not to violate (a).
   (c) All starting dates will be legal (properly entered).
   (d) All time lapses will be forward. (Final date will be later than initial date.)

11. This is a slightly more involved version of Problem 10. Specifically, we shall not make assumption (c). The only thing we can say about the six input values is that they always will be numbers. Since assumptions (a) and (b) are still being made, this program needs to check for errors in all six input values. That is, it should recognize and reject an illegal starting date. In addition, it should recognize and reject values of MLAPSE,

DLAPSE or YLAPSE that violate assumption (b). In doing this, make sure that the program is able to recover from the error (after printing an appropriate error message) by going on to the next set of input values.

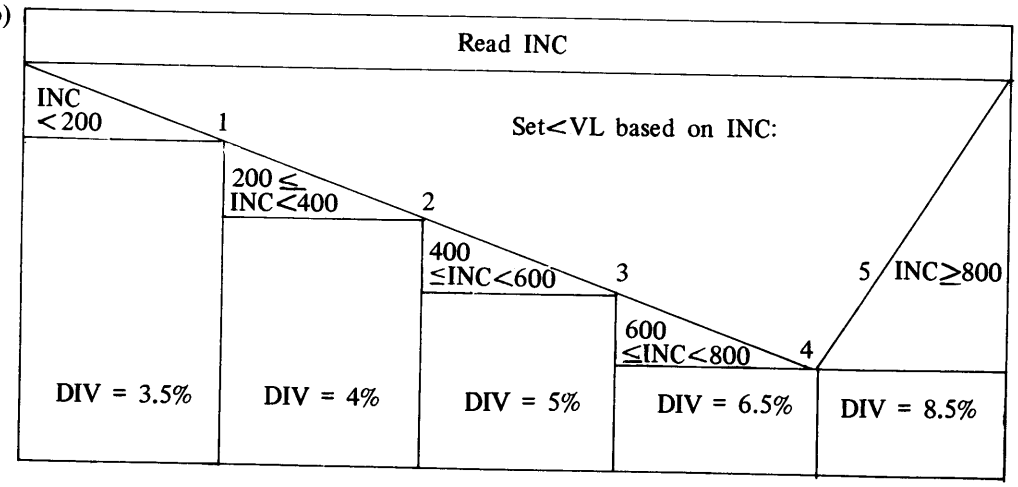
12. This is a modified version of Problem 11 in which we complicate things a little more by removing assumption (b). Thus, an elapsed time period of 0 years, 14 months and 39 days, for example, now is acceptable.

13. Here is one more modification, a little more involved than Problem 12. For this version, we remove assumption (a), leaving only assumption (d). (The more adventurous may remove assumption (d) as well.)

14. Write a sequence of FORTRAN statements that specifies each of the CASE constructs described below. You may use either the ELSE IF or computed GO TO approaches:

(a) "Define real variables X, Y, Z and integer variable MV."
"Read X and Y."
"Set MV in accordance with X and Y:
X less than or equal to Y: MV = 1;
X greater than Y but not greater than 2Y: MV = 2;
X greater than 2Y but not greater than 3Y: MV = 3;
X greater than 3Y: MV = 4."
*case* MV of
case1:
"Set Z = X."
case2:
"Set Z = X + SQRT(Y)."
case3
"Set Z = X + Y."
case4:
"Set Z = X + 2Y.
*endcase*

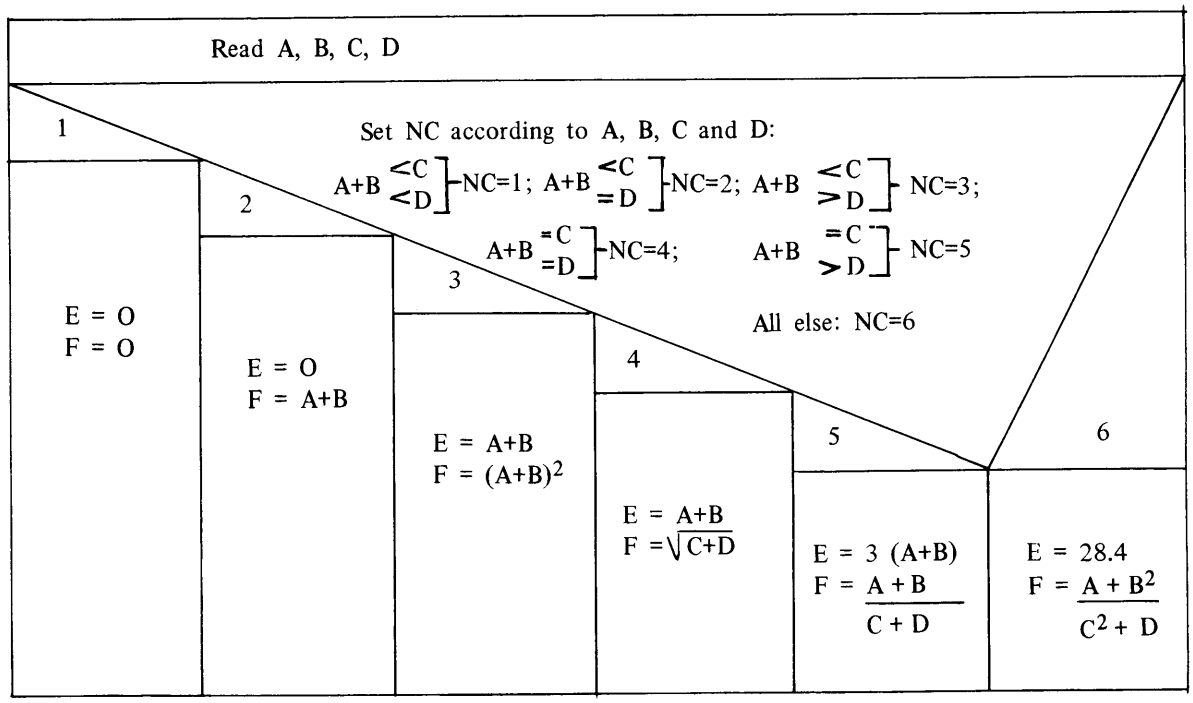
(b)




(c) "Define real variables X, Y and Z and integer variable CHC."
"Set CHC in accordance with X, Y, and Z:
Z less than X and:
Z less than Y: CHC = 1;
Z not less than Y but less than 2Y: CHC = 2;
Z not less than 2Y: CHC = 3;
Z equal to or greater than X and:
Z not greater than Y: CHC = 4;
Z greater than Y but not greater than 3Y: CHC = 5;
Z greater than 3Y but less than 4Y: CHC = 6;
Z not less than 4Y: CHC = 7."

*case* CHC *of*
case1:
"Double the value of Z."
"Reduce X's value by 3.2."
case2:
"Double the value of Z."
"Reduce X's value by 2.8."
case3:
"Reset Z to 1.8 times its current value."
case 4:
"Add 1.9 to X's value."
case5:
"Reset Z to 1.5 times its current value."
"Add 1.5 to X's value."
"Reset Y to 0.8 times its current value."
case6:
"Leave everything the way it is."
case7:
"Reset Z to 0.96 times its current value."
"Add 2.2 to Y's current value."
case8:
"Replace Z with X+2Y."
*endcase*

(d)




(e) Indicate the nesting level for each of the IF blocks when the construct in Part (d) is implemented using the ELSE IF form.

15. The Vital Signs Casualty Company insures motorists according to a schedule of basic rates which then are modified by the type of car (1, 2, or 3), the customer's occupational category (A, B, C, D, or E), and the customer's Scientific Client Profile. The latter is based on a special screening process that assigns a value of 10, 31, or 77. (I don't know what these numbers mean, but Vital does.) Basic rates are as

follows:

| | |
|---|---|
| Liability: | \$20 for each \$10,000 of coverage |
| Comprehensive: | \$ 5 for each \$1,000 of coverage |
| Medical: | \$15 for each \$10,000 of coverage |
| Collision: | \$150 flat rate for \$100.00 deductible coverage |

These rates are adjusted as follows:

| | |
|---|---|
| Car Type 2: | 5% discount on liability |
| | 10% discount on comprehensive |
| Car Type 3: | 8% discount on liability |
| | 11% discount on comprehensive |
| Occupational category A: | 5% discount on liability |
| | 2% discount on comprehensive |
| Occupational category B: | 6% discount on liability |
| Occupational category D: | 2% surcharge on liability |
| | 2% surcharge on comprehensive |
| Occupational category E: | 4% surcharge on liability |
| | 3% discount on comprehensive |
| Client profile 10: | 7% discount on liability |
| | 4% discount on comprehensive |
| | 10% discount on medical |
| Client profile 77: | 15% surcharge on liability |
| | 20% surcharge on medical |

Each of the discounts or surcharges is applied independently. Thus, a client with a Profile of 10 and occupational category of A driving a type 3 car enjoys a 20% liability discount, 17% discount on comprehensive coverage, and a 10% medical discount as well. Oboy!

Input for each client is a line containing the following items: Customer number (CNUM), a six digit integer; type of car (CARTYP), occupational category (OCCUP), profile (SCP), desired amount of liability coverage (LIABIL) in tens of thousands of dollars, desired medical coverage (MED) in tens of thousands of dollars, and the collison option (COLSN), recorded as 'YES' or 'NO'. For example, the following line

```
103472    2  'B'  31  5  6  8  'YES'
```

represents input for customer 103472 having a type 2 car, occupational category B and SCP of 31. This individual wants \$50000 of liability coverage, \$6000 of comprehensive coverage, \$80000 of medical coverage, and collison insurance.

Write a program that prints each input item on a separate line followed by a final output line showing the policy's total cost, rounded to the nearest dime. Separate each output set by a blank line and end the run with a customer number of zero.

16. Write a program that reads sets of three real values A, B, and C. Each set is processed as follows:

| | |
|---|---|
| When A, B, and C are equal, | `D = A(B+C)` and E is zero; |
| When A and B are equal and both are less than C, | `D = A*B**2` and `E = C**2` |
| When A and C are equal and both are greater than C, | `D = A**2*B**2` and `E = B**2*C` |
| When A and C are equal and both differ from B, | `D = B**2(A-C)` and `E = A/B` |
| When B and C are equal and both are less than A, | `D = (A*C)/(A+B)` and `E = 1` |
| When B and C are equal and both are greater than A, | `D = SQRT(2A)` and E is `1/D` |
| When A is less than B and B is less than C, | D and E both are zero |
| When A is less than B and B is greater than C, | `D = SQRT(A+B+C)` and `E = SQRT(D)` |
| When A is less than C and C is less than B, | `D = (A+B+2C)**1.8` and `E = LOG(D)` |
| When A is less than C and C is greater than B, | `D = SQRT(A+B**2-C**2)` and `E = D/2` |
| When anything else happens, | D is 1 and E is 2. |

Print A, B, and C on one line, followed by D and E on a second line. Leave a blank line between output sets and end the run with zero values for all three input variables. Here is some suggested input:

| *A* | *B* | *C* |
|---|---|---|
| 17.4 | −2.2 | 4.1 |
| −6.8 | 0.0 | 28.7 |
| 31.0 | 31.0 | 31.0 |
| 449.5 | −449.5 | 449.5 |
| 217.8 | 46.4 | −81.0 |
| 67.8 | 386.9 | 67.8 |
| −7.1 | −90.0 | −108.4 |
| 0.0 | 0.0 | 0.0 |

17. Here is a research question: Pick your favorite (or least favorite) airline and find out how many different ways there are to determine what it costs to fly between two particular cities. Then, having defined the information you need to make the computations, write a program that reads this information and computes (and prints) the appropriate cost. Some suggested journeys are listed below. A few are easier than others, but remember that all of them are subject to such considerations as class of flight, family plans, special excursions, children's rates, night flight or day flight,and so. Have a good trip:

    (a) Saint Louis to Kansas City
    (b) Philadelphia to Pittsburgh
    (c) New York to Chicago
    (d) New York to Miami
    (e) New York to Los Angeles
    (f) New York to London
    (g) Washington, D.C. to Chicago
    (l) Chicago to Honolulu
    (m) Chicago to London
    (n) Los Angeles to Acapulco
    (o) San Francisco to Las Vegas

18. Here is another research question: Find out the rates charged by your telephone company. To keep it relatively simple, limit your scope to residential service. Then, define the necessary date items you will need as input. Based on those definitions, write a program that reads in a customer's identification, together with a list of desired options, and produces a rate for that customer.

# 10
# *Cyclic Operations*

The fundamental importance of the program loop is reflected in FORTRAN 77's extensive support of techniques and features that make it convenient to set up an endless variety of such loops. We shall explore these capabilities by reviewing the structured programming concepts within which such loops operate, after which we can extend our ability to build them through the use of additional language capabilities.

## 10.1 LOOPS AS STRUCTURAL COMPONENTS

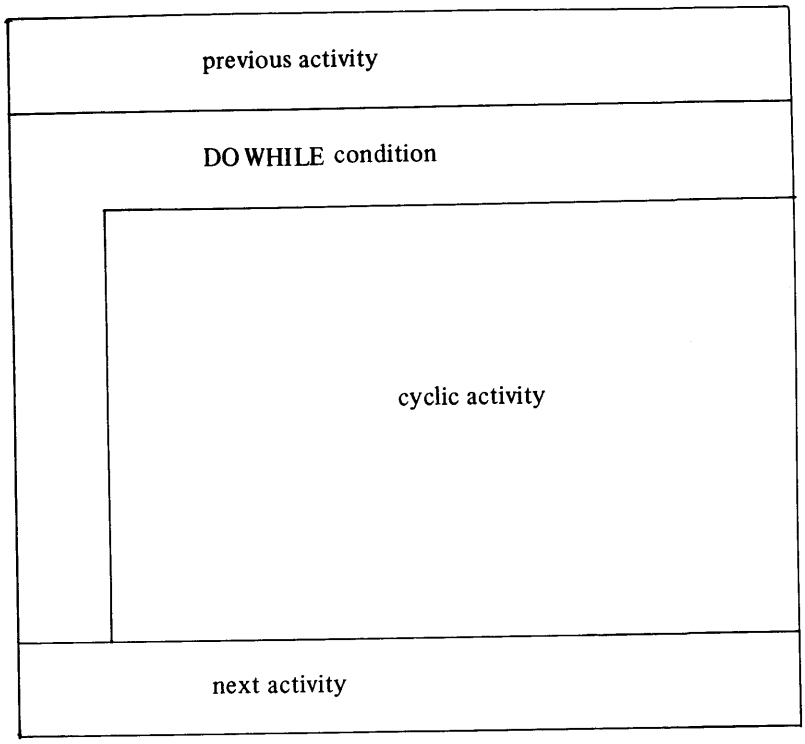
Regardless of its length or complexity, a well-designed program loop in FORTRAN represents an implementation of one of two basic procedural components: the DO-WHILE construction, or the DO-UNTIL construction. (The latter also is known as the REPEAT-UNTIL construction.)

### 10.1.1 The DO-WHILE Construction

This is the basic construction that we have been using all along to design sequences of operations that can repeat again and again under program control. The composition of the DO-WHILE construction (repeated for convenience in Figure 10.1) starts with a test to determine whether or not to perform the loop's operations. As long as the test's conditions are met, the program enters the loop and performs whatever processing is specified inside. Then, before that processing can be repeated, the condition is tested again to see if the outcome still is the same as it was for the previous test, thereby allowing the loop to be entered again. This continues until the condition changes, at which time the entire loop is bypassed and processing resumes with the statement just after the conclusion of the loop. Thus, it is quite possible for a program to reach a DO-WHILE construction, test the condition, find that it is not met, and not perform the loop even once.

**10.1.1.1 Direct Implementation of the Do-WHILE Component** Perhaps the most puzzling aspect of standard FORTRAN 77's scope is the absence of a direct representation for the DO-WHILE structure. The fundamental nature of the component and its importance in the programming process makes it an obvious candidate for inclusion. Consequently, it is not surprising that HP FORTRAN 77 includes this important extension to the standard language. We have been using it all along, so it will not be necessary to carry our discussion here much beyond a concise restatement of the general form:

```
DO WHILE (condition)
  statement
  . . . . .
  statement
END DO
```




(b)

"Previous activity."
**while** this condition is true:
    "Perform this activity."
**endwhile**

(a)

**FIGURE 19.1 (a)** Pseudocode Representation for the DO-WHILE Construct. **(b)** N-S diagram for the DO-WHILE Construct.

The processing inside the loop is bracketed specifically by the DO WHILE and END DO statements. As long as the condition specified in the DO WHILE statement is met (that is, as long as its outcome is . TRUE. ), the activities inside the loop will be repeated.

**10.1.1.2 Implementation of DO-WHILE with the Logical IF Statement** Since HP FORTRAN 77 provides the DO WHILE. . . . END DO form as a direct implementation of the Do-WHILE structure, any alternative form is redundant (and likely to be more awkward as well). However, since standard FORTRAN 77 lacks a direct mode of expression for this fundamental component, we shall outline (briefly) two ways around this deficiency. This information will come in handy for those instances in which you may be obligated to prepare a FORTRAN program for use (i.e., for recompilation) on a standard FORTRAN 77 system.

The general approach here is to "duplicate" the test at the entrance to the loop by using a labeled logical IF statement. Because of the properties of this statement, it is necessary to set up the test so that an outcome of . TRUE. sends the program around the loop, and an outcome of . FALSE. allows entry. The activity inside the loop is described as before. Then, instead of relying on an automatic mechanism (arranged by the DO WHILE statement) to send the program back to the entry test, the return must be specified explicitly by means of a GO TO statement whose destination is the labeled IF statement just described:

```
24  IF (condition) GO TO 99
        loop activity
        . . . . . .
        loop activity
    GO TO 24
        . . . . . .
99  next activity
```

For instance, in many of our examples, we have been specifying the processing within a loop that goes around for each set of input. When some particular signal is detected, the loop is bypassed and we continue with the next part of the program. The following fragment

```
. . . . . .
DO WHILE (CUSTNO . NE. 0)
   loop activity
   . . . . . .
   read data for the next customer
END DO
. . . . . .
next activity
```

The same type of process can be expressed as

```
1  IF (CUSTNO . EQ. 0) GO TO 99
      loop activity
   . . . . . .
   read data for the next customer
   GO TO 1
   . . . . . .
99 next activity
```

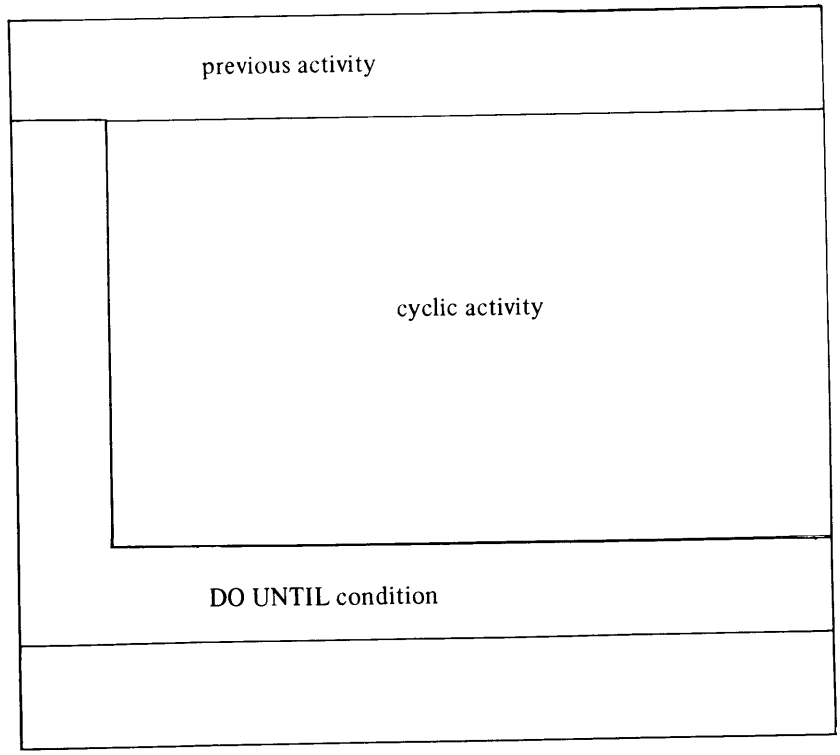
**10.1.1.3 DO-WHILE Implementation with the Block IF Statement** A more direct carry-over of the DO-WHILE's concept is possible with the block IF statement, since it allows us to specify any number of statements as a single conceptual activity. By using the IF. . . . . THEN form without an ELSE activity to go with it, i.e.,

```
n  IF (condition)  THEN
   . .
   action 1
   . .
   GO TO n
   END IF
   next statement
```

we can build a DO-WHILE construct in which the loop is expressed as "action 1." Note that action 1 concludes with a GO TO statement that brings the program right back to the test. Thus, as long as the test's outcome is true, the action repeats. As soon as the outcome is false, the *entire* IF-block is bypassed, and the next thing the program does is whatever is specified in "next statement." The little example in the previous section, then, becomes

```
1  IF (CUSTNO . NE. 0)   THEN
      loop activity
      . . . . . .
      read data for the next customer
   GO TO 1
   END IF
   next statement
```

Some people follow a practice that calls for each DO-WHILE loop to start with a labeled CONTINUE statement and end with a GO TO to that statement. Since the CONTINUE statement "does" nothing (it merely indicates the forward progress of the program), its use standardizes the loop's boundaries and makes them more con-




(b)

"Previous activity."
**do**
    "Perform this activity over and over."
**until** this condition is true.

(a)

**FIGURE 10.2 (a)** Pseudocode Representation of the REPEAT-UNTIL Construct. **(b)** N-S Diagram for the DO-UNTIL Construct.

spicuous. As an example, we can use this convention to rewrite the previous fragment:

```
1 CONTINUE
  IF (CUSTNO . NE. 0)   THEN
    loop activity
    . . . . . .
    read the data for the next customer
  GO TO 1
  END IF
  next statement
```

### 10.1.2 The DO-UNTIL Construction

This form specifies a loop in which we wish to guarantee that the processing inside that loop will be performed *at least once*. Accordingly, the controlling test is placed *after* the statements instead of in front of them, as it is in the DO-WHILE construct. Thus, each time the test's outcome is true, the program will go back and repeat the processing in the loop. This cycle continues until the outcome is false, at which time the program moves on. For this reason, some people refer to this structure as the REPEAT-UNTIL component. A diagram and pseudocode representing this structure are shown in Figure 10.2.

Various forms may be used to implement this construction in FORTRAN. The approach shown below illustrates one simple way to do it:

```
      previous statement
MM    first statement in the loop
        second statement
            .....
            .....
      statement                          activity to be repeated
      statement
            .....
        final loop statement
      IF (condition) GO TO MM            controlling test
                                         next activity
```

**Example 10.1** Decibel and Associates, a public relations firm, is in the midst of a campaign to improve the image of Ratt and Sons Sewer Cleaning Company ("If your sludge won't budge, have a chat with Ratt.") As part of the effort, a local sweepstake has been launched in which people are being urged to send in their name, address, and their favorite one-word name (8 letters or less) for sewage. On a certain date, Decibel will run a program that examines all the available data and selects the first fifty people whose last names have certain combinations of letters. (The actual combinations will not be made public until the day the program is run.) Each lucky winner will receive a handsome glossy photograph of B. W. Ratt (suitable for framing) and a certificate entitling the holder to free removal of the first fifty pounds of sludge when he or she arranges for such work with Ratt and Sons.

Of course, the program to select the fifty fortunate folks must contain information that defines how the choices are to be made. The program will select anyone with a double letter in his or her last name (like Feffer, Frumpp, or Deedle). Data for each person entering the great drawing will be submitted to the program in no particular order and will consist of:

1. Last name, first initial (up to 15 letters for the last name, with the final letter followed by a comma, a blank, the first letter of the first name, and a period)
2. Street number (up to 5 digits)
3. Street name (up to 10 characters)
4. Street type (e.g., ST, AVE, LN, RD) (up to 4 letters)
5. City (up to 10 letters)
6. State (two letters)
7. Zipcode (5 numerical characters)
8. Sludge word (up to eight letters)

A typical input line is shown in Figure 10.3. For each winner, the program is to print the name and address, as well as the winner's number (for example, a number of 27 would indicate that the person is the 27th one selected.)

```
'PIPPICK, M. ' '17 DEBRIS LN ANTIQUE IL 606060 'GRUNGE'
```

**FIGURE 10.3** Input Card Data for Ratt Image Campaign.

Note from Figure 10.3 that the address is represented as a single character string that includes street number, street name, city, state, and zipcode. That is, the entire address is submitted inside a single set of apostrophes. Using the limits given in the previous list of the address components, and allowing for a blank between each part and the next one, we can determine that the address may be as long as 5+1+10+1+4+1+10+1+2+1+5 or 41 characters. (Lengths for the name and sludge word already are defined.)

The basic component of the processing is a DO-UNTIL construction that examines each last name and continues to select and print winners until there are fifty. This is done as follows: starting at the

```
"Define NUMENT, NUMWIN, LTRIND, NAME, ADDRSS, WORD."
"Initialize NUMENT, NUMWIN."
"Print the headings."
do
    "Read an entry."
    "Add 1 to NUMENT."
    "Initialize LTRIND."
    while there still are letters to examine in NAME and this entry has not been declared a winner:
        if
            a double letter has been found
        then
            "Add 1 to NUMWIN."
            "Print NUMWIN, data for the winner."
        else
            "Continue the search for a double letter."
        endif
    endwhile
until there are 50 winners.
"Stop."
```

**FIGURE 10.4 (a)** Pseudocode Representation for Example 10.1.

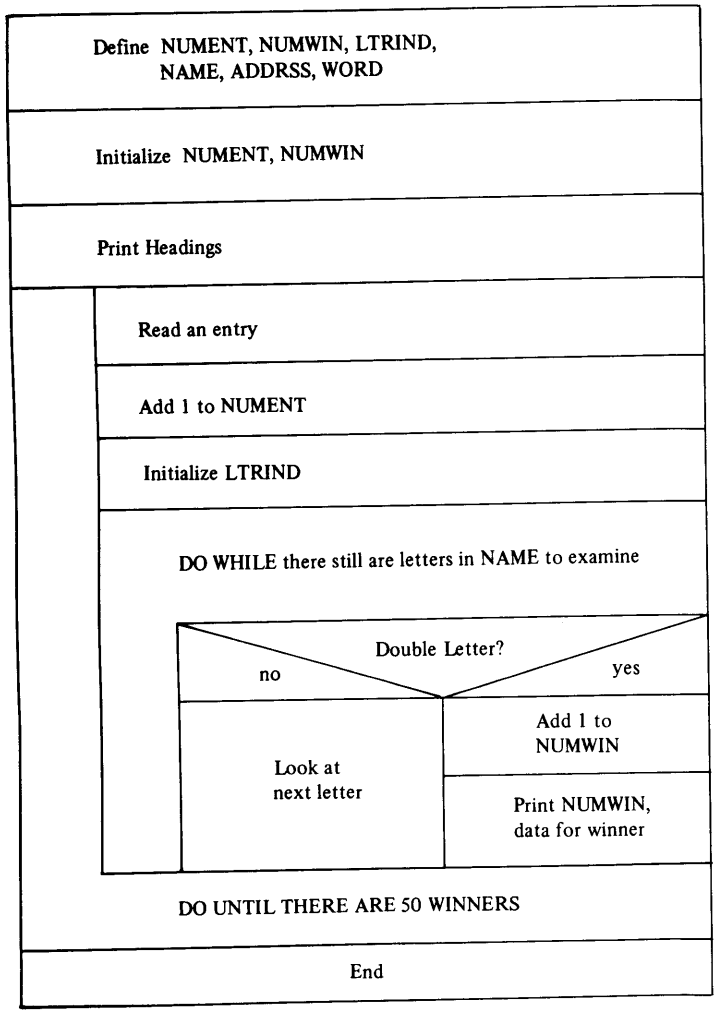



**FIGURE 10.4 (B)** N-S Diagram for Example 10.1.

```
************************************************************
*                    EXAMPLE 10.1                          *
************************************************************
*   NUMENT:     A COUNTER TO KEEP TRACK OF THE TOTAL NO.   *
*               OF ENTRIES                                 *
*   NUMWIN:     A COUNTER KEEPING TRACK OF THE NO.         *
*               OF WINNERS                                 *
*   LTRIND:     COUNTER THAT KEEPS TRACK OF THE LETTER IN  *
*               IN THE LAST NAME THAT IS CURRENTLY BEING   *
*               COMPARED WITH ITS NEIGHBOR.                *
************************************************************
      PROGRAM        EX1001
      IMPLICIT       NONE
      INTEGER*2      NUMENT,NUMWIN,LTRIND
      CHARACTER      NAME*19,ADDRSS*41,WORD*8,COMMA*1
      PARAMETER      (COMMA=',')

      NUMENT = 0
      NUMWIN = 0
      PRINT *, 'RATT AND SONS SUPER SWEEPSTAKES'
      PRINT *, '     '
      PRINT *, 'WINNER NO.','NAME','ADDRESS'
      PRINT *, '     '
      PRINT *, 'ENTER THE DATA FOR THE FIRST INDIVIDUAL'

      DO WHILE (NUMWIN .LT. 50)
         PRINT *, 'SUBMIT INPUT DATA FOR AN ENTRANT'
         READ *, NAME, ADDRESS, WORD
         NUMENT = NUMENT + 1
         LTRIND = 1
         DO WHILE (NAME(LTRIND:LTRIND) .NE. COMMA)
            IF (NAME(LTRIND:LTRIND) .EQ. NAME(LTRIND+1:LTRIND+1)) THEN
               NUMWIN = NUMWIN+1
               PRINT *, NUMWIN, NAME, ADDRESS
               NAME(LTRIND:LTRIND) = COMMA
            ELSE
               LTRIND = LTRIND+1
         END DO
      END DO

      PRINT *, '     '
      PRINT *, 'END OF RUN. NORMAL TERMINATION.'
      STOP
      END
```

**FIGURE 10.5** FORTRAN Statements for Example 10.1.

beginning of the name, each letter is compared with the one following it. As soon as a match is found, that person is identified and processed as a winner, and the program goes on to the next entry, or it finishes if it has found fifty. If no match is found, the program continues to move through the name until it reaches the comma after the last letter, at which point it starts over with the next person's data. For this version of the program, we shall assume that Decibel and Associates has done a good job, so that there are enough entries to guarantee that at least 50 winners will be found long before we run out of data. The flowchart and pseudocode for the program are given in Figure 10.4 and the program itself is in Figure 10.5.

## 10.2 AUTOMATIC PROGRAM LOOPS: THE DO STATEMENT

Many times the control of a loop needs to be based on a certain number of repetitions instead of some other type of event. This can be set up easily enough by initializing a counter and using it to control the cycles. For instance, the loop shown below,

```
      INTEGER*2 COUNT
          .......
          .......
      COUNT = 1
10    statement
       ....
       ....                    action 1
      statement
      COUNT = COUNT+1
      IF (COUNT .LE. 36) GO TO 10
      next activity
```

is constructed so that "action1" is processed 36 times. When "next activity" is reached, the control variable (COUNT) will have a value of 37. Note that this is basically a DO-UNTIL construct with a counter's value as the controlling event.

HP FORTRAN 77 provides a convenient and versatile way to specify loops in which the cycles are controlled automatically. This mechanism is called a *DO loop*, and its general form can be illustrated by automating the little loop we set up earlier:

```
INTEGER*2       COUNT
     .......
     .......
DO COUNT=1, 36
   action 1              DO loop (action1 performed 36 times)
END DO
next activity
```

The corresponding flow diagram and pseudocode are shown in Figure 10.6.

As the example implies, control of the loop is concentrated in the *DO statement* with which the loop begins. Each part of the statement provides specific information that helps define exactly how such control is to be exercised. This is seen in the general statement given below:

DO *index* = *sv*, *lv*, *incr*

The statement's parts have the following meanings:

1. *index* (COUNT in our example) is the variable used to keep track of the cycles through the loop. In addition, it is available for a variety of uses inside the loop, as long as such usage does not affect its value. The place where this value changes (and the only place we want it to change) is within the DO statement, as part of the cycle's control mechanism. This is called the *index* or DO-*variable*.

2. *sv* is the starting value to which the index is set just before the loop is processed for the first time. Although *sv* is 1 in our example, it can be any value that suits the program's purpose.

3. *lv* is the limiting value that determines when the repetitions should stop. If the index has not gotten to be larger than *lv*, then action 1 is performed. Otherwise, the loop is bypassed and processing continues with "next activity." In our example, *lv* is 36, so that the loop continues to cycle as long as the index (COUNT) is no greater than 36.

"Previous activity."
**do** from starting value to limiting value by increment:
"Perform this activity over and over."
**enddo**

**FIGURE 10.6 A)** Pseudocode Representation of a DO Loop.

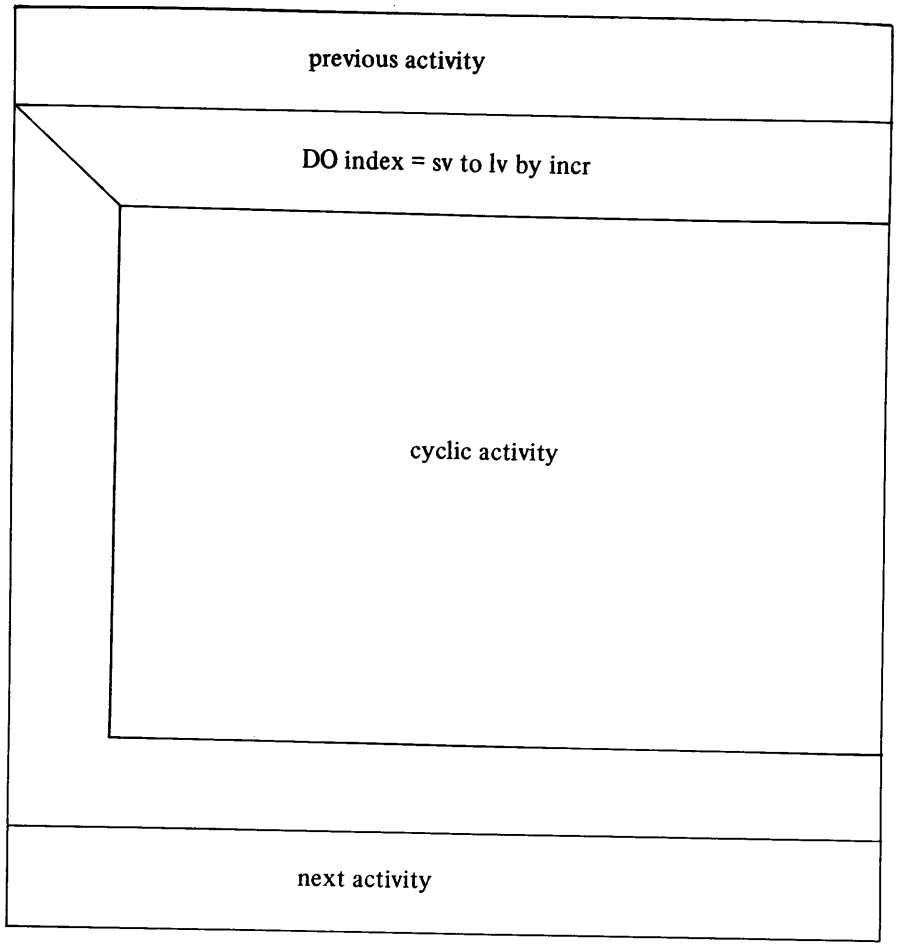



**FIGURE 10.6 (b)** N-S Diagram for a DO Loop.

4. *incr* specifies the amount that is added automatically to *index* after each cycle. Note that there is no *incr* specified in our example. When this information is absent, FORTRAN uses a *default* value of 1, so that the DO statement in our example behaves just as if we had written

```
DO 10 COUNT=1, 36, 1
```

*incr* is the only item that can be omitted from a DO statement. All the other specifications (*label*, *index*, *sv*, and *lv*) always must be there.

The extent of the loop (i.e., the number of statements that comprise the loop's activities) is set by the END DO statement.

This form is not available in standard FORTRAN 77. Instead, a loop's initial DO statement includes an additional specifier:

DO *label index* = *sv*, *lv*, *incr*

*label* refers to a label that must be attached to the loop's concluding statement. This is the way FORTRAN "knows" how far the loop extends, i.e., the *range* of the loop.

Thus, our structural example, rewritten in standard form, would look like this:

```
INTEGER*2          COUNT
    .......
DO 10 COUNT=1, 36
   action 1
    .......
10 Loop's concluding statement
   next statement
```

Although FORTRAN allows the programmer to conclude this type of DO loop with various types of statements, good practitioners end all such loops in a consistent way, so that their structure is standardized and, therefore, easily seen. The most common convention in this regard is to conclude every DO loop with a CONTINUE statement. Accordingly, that statement will have the same label as specified in the label part of the DO statement with which the loop starts.

**Example 10.2** One of the most common uses for DO loops is the processing of arrays such that some activity is performed on each array element in turn. In this example we have a one-dimensional array AR1 consisting of 30 integers, and we are required to compute and print a 30-element array AR2 in which each element is the sum of the corresponding element in AR1 and the two elements following it. For instance, AR2(1)'s value would be computed as AR1(1) + AR1(2) + AR1(3); AR2(7)'s value would be computed as AR1(7) + AR1(8) + AR1(9), and so on. Elements at the end of the array would be computed by using AR1's elements in a "wraparound" manner. That is, AR2(29) is computed as AR1(29) + AR1(30) + AR1(1) and AR2(30) is computed as AR1(30) + AR1(1) + AR1(2). Both AR1 and AR2 are to be printed in order, five elements per line. AR1's 30 elements are available as input, in proper sequence.

Processing for the program can be divided into four major parts:

1. A simple READ statement to bring in AR1;
2. A DO loop to print it;
3. A second DO loop to produce AR2;
4. A final DO loop to print AR2, this being similar to the loop for printing AR1.

We shall turn our initial attention to the second DO loop, noting that the loop's index offers us a convenient way to keep track of the element currently being processed. Thus, if some variable called INDX is designated for this purpose, we can use INDX as a subscript, thereby identifying AR2(INDX) as the particular element of interest during a given cycle through the loop. The corresponding element in AR1 would be AR1(INDX), and the next two, then, would be AR1(INDX+1) and AR1(INDX+2). As the value in INDX changes with each cycle, the reference to an array element automatically changes with it. We can see this clearly by setting up the loop for computing AR2:

```
DO INDX=1, 30, 1
   AR2(INDX) = AR1(INDX) + AR1(INDX+1) + AR1(INDX+2)
END DO
```

As the DO statement indicates, INDX will be started at a value of 1 so that, during the first cycle, AR2(1) is computed as AR1(1)+AR1(2)+AR1(3). Then, INDX is increased to 2 (since incr is 1) and the process is repeated, this time with AR2(2) being computed as AR1(2)+AR1(3)+AR1(4). The cycle continues this way, with INDX increasing by 1 each time. But wait: all is not well! When INDX is 28, the loop repeats, with AR2(28) being computed as AR1(28)+AR1(29)+AR1(30). Fine. Now, INDX is increased to 29, which still does not exceed 30, so another cycle begins. This time, the program attempts to use AR1(29), AR1(30), and AR1(31). This cannot happen, since there is no AR1(31). Consequently, this loop, as written, will not work for all 30 sets of computations.

One way to get around this difficulty is to use a DO loop for everything that works, and then take care of the loose ends separately. In this case, we could compute AR2's first 28 elements, handling AR2(29) and AR2(30) as individual cases:

```
DO INDX=1, 28, 1
  AR2 (INDX)  = AR1 (INDX) +AR1 (INDX+1) +AR1 (INDX+2)
END DO
AR2 (29)  = AR1 (29) +AR1 (30) +AR1 (1)
AR2 (30)  = AR1 (30) +AR1 (1) +AR1 (2)
```

This will work, but it is a poor way to handle the situation, even one as simple as this. Once such processes start getting more complex, the loose ends multiply and become tangled in each other, and the simple program structure that we are working to preserve goes out the window. This approach was shown simply as a starting point, so that its awkwardness can be noted and its use discouraged.

A somewhat less cumbersome approach is to include the computation of all 30 elements in a single loop. The two exceptions are singled out for special treatment, but they still are within that loop:

```
DO INDX = 1, 30, 1
  IF (INDX . LT. 29) THEN
    AR2 (INDX)  = AR1 (INDX) +AR1 (INDX+1) +AR1 (INDX+2)
  ELSE IF (INDX . EQ. 30) THEN        AR2 (INDX)  = AR1 (30) +AR1 (1) +AR1 (2)
       ELSE
         AR2 (INDX)  = AR1 (29) +AR1 (30) +AR1 (1)
  END IF
END DO
```

As indicated earlier, this construction is somewhat more preferable to the previous version. However, it is still better to find a way of handling all the elements consistently in a single loop, as long as the resulting program segment does not become so complicated that the clarity suffers. In this case, we can identify such an approach. By taking advantage of the MOD built-in function, it is possible to develop a fairly simple expression which will automatically adjust a number when that number goes over 30. In other words, the expression must produce the following type of result:

| *if number is* | *expression produces* |
|---|---|
| 1 | 1 |
| 2 | 2 |
| . | . |
| . | . |
| 29 | 29 |
| 30 | 30 |
| 31 | 1 |
| 32 | 2 |
| . | . |
| . | . |

The required expression can be constructed simply by applying the MOD function with a divisor of 30 (the size of the array) to 1 less than the number. Then, a final adjustment is provided by adding 1 to the returned value of the function. Thus, if k is the number needing adjustment, the mechanism that will perform that adjustment is MOD (k−1, 30) +1. For instance, if k happens to be 26, MOD (26−1, 30) is 25, and the final addition of 1 brings it to 26. Similarly, if k is 31, MOD (31−1, 30) is 0, and the addition of 1 produces the desired result, 1.

Now, we can apply this mechanism directly to the problem at hand: if the number is INDX, the adjustment is MOD (INDX−1, 30) +1. Since INDX never goes above 30, any adjustment to INDX produces the same value as INDX. (For instance, MOD (27−1, 30) +1 is still 27.) Consequently, INDX needs no adjustment. However, when the number is INDX+1, the result can be as high as 31. Consequently, INDX+1 needs adjustment, and when we apply the expression to it, we get MOD (INDEX+1−1, 30) +1, which simplifies to MOD (INDX, 30) +1. Similarly, when we apply the adjustment to INDX+2, the result is MOD (INDX+2−1, 30) +1, which simplifies to MOD (INDX+1, 30) +1. Using these developments, we can rewrite the loop more concisely:

```
 DO INDX=1, 30, 1
   AR2 (INDX)  = AR1 (INDX)  + AR1 (MOD (INDX, 30) +1)
1                            + AR1 (MOD (INDX+1, 30) +1)
 END DO
```

The loop that will be used to print AR1 illustrates another aspect of the DO statement's versatility. Since each PRINT statement starts a new line of output, we shall build our loop around a statement that lists five elements, and the loop will repeat that statement six times. To make sure we get the right elements in proper sequence, they will be listed as AR1 (INDX), AR1 (INDX+1), AR1 (INDX+2), AR1 (INDX+3), and AR1 (INDX+4). To keep the control mechanism consistent with the output, the increment in our loop will be 5 rather than 1. Thus, the first time through the loop, INDX will be 1 and the five elements to be printed will be AR1 (1) through AR1 (5). Then, if we increase INDX by 5, its value will be 6 when the loop is processed for the second time. Accordingly, the PRINT statement will use AR1 (6),

"Define AR1(30), AR2(30), and INDX."
"Read AR1."
"Print AR1."
**do** for each element in array AR1:
    "Assign to AR2(i) the sum of the element in AR1(i) and the next two elements in sequence. AR1 wraps around, so that AR1(1) follows AR1(30)."
**enddo**
"Print AR2."
"Stop."
"End."

**FIGURE 10.7 (a)** Pseudocode Representation of Example 10.2.

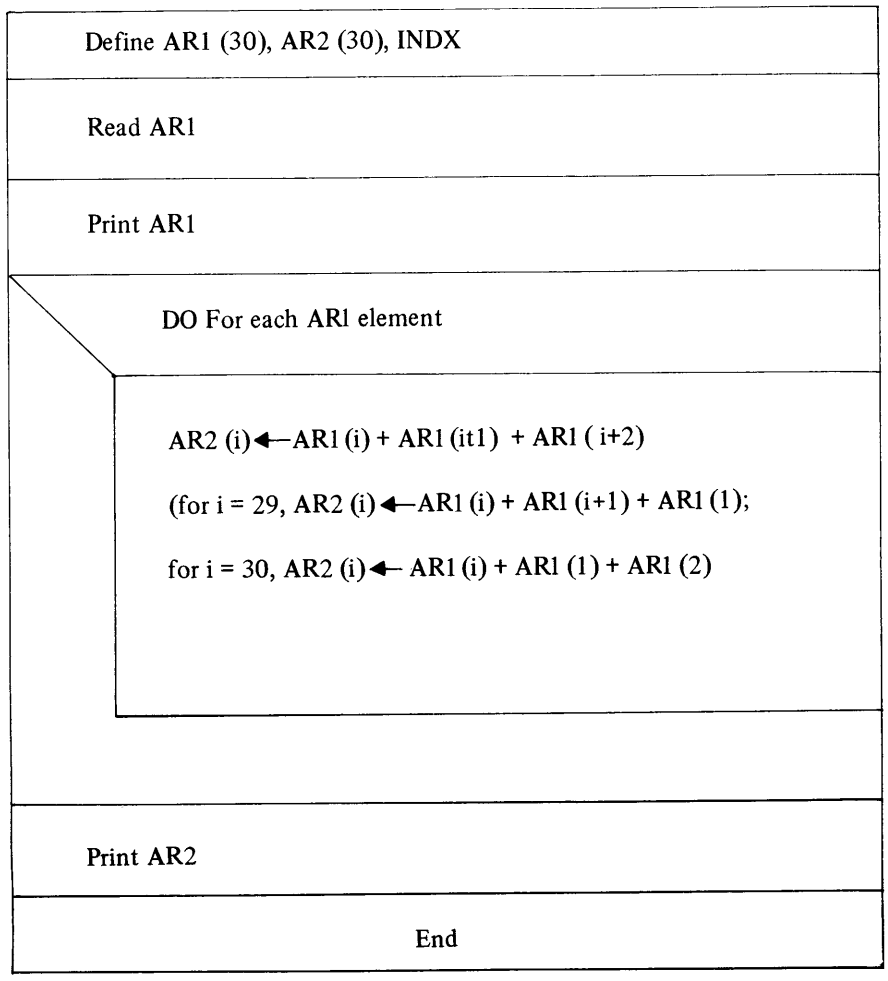



**FIGURE 10.7 (b)** N-S Diagram for Example 10.2.

AR1(6+1), etc. through AR1(6+4), or AR1(10):

```
      DO INDX=1,30,5
        PRINT *, AR1(INDX), AR1(INDX+1), AR1(INDX+2),
     1                      AR1(INDX+3), AR1(INDX+4)
      END DO
```

Note that when the loop is ready for a sixth cycle, INDX is at 26. Since this does not exceed the limiting value (30), the PRINT statement will be executed, producing the last five elements of AR1. Then, INDX will be increased by 5, making it 31. Since this exceeds 30, the loop will be bypassed. Incidentally, it is all right to use the same variable as an index for more than one DO loop as long as the loops are not nested. (We shall be dealing with nested loops a little later in the chapter.)

Now, having developed a little experience with the DO loop and its properties, we can put the pieces together to construct the entire program. Its diagram and pseudocode are given in Figure 10.7 and the statements are listed in Figure 10.8.

```
C*************************************************************
C                        EXAMPLE 10.2                         *
C*************************************************************
      PROGRAM          EX1002
      IMPLICIT         NONE
      REAL*4           AR1(30),AR2(30)
      INTEGER*2        INDX

      PRINT *, 'ENTER A SET OF VALUES FOR ARRAY AR1'
      READ *, AR1
      PRINT *, 'INPUT ARRAY AR1:'
      DO INDX=1,30,5
        PRINT *, AR1(INDX), AR1(INDX+1), AR1(INDX+2),
     1                    AR1(INDX+3), AR1(INDX+4)
      END DO
      PRINT *, '     '

      DO INDX=1,30
        AR2(INDX) = AR1(INDX) + AR1(MOD(INDX,30)+1)
     1                        + AR1(MOD(INDX+1,30)+1)
      END DO

      DO INDX=1,30,5
        PRINT *, AR2(INDX), AR2(INDX+1), AR2(INDX+2),
     1                    AR2(INDX+3), AR2(INDX+4)
      END DO

      PRINT *, 'END OF RUN. NORMAL TERMINATION.'
      STOP
      END
```

**FIGURE 10.8** FORTRAN Statements for Example 10.2.

### 10.2.1 Useful Techniques with DO Loops

This section presents a number of constructions designed to illustrate the versatility of the DO loop. By exploring ways in which the control elements can be used to regulate a loop's cycles, we shall be able to design more and more intricate controls without complicating the programming.

**10.2.1.1 Use of the Index in Loop Calculations** Although the index variable may not be changed within the loop, its value certainly is available for use. An illustration of such usage is shown below.

**Example 10.3** One of the original factors that motivated the development of computers was the need for tables of values computed for certain mathematical functions. These were required for astronomy, navigation, commerce, and a variety of other uses. This type of data production still is requested frequently, and the DO loop is a convenient way to do such work. For this example, we shall generate a value Y using the formula

$$Y = \frac{1 + K^2}{4K}$$

A series of such values is to be computed for K = 1, 2, 3, . . . , 50. The resulting output table is to contain 50 lines, with each one showing a value of K and the corresponding value of Y.

This plan can be implemented easily by setting up a loop in which the index is used for the value of K. If we initialize Y to zero before entering the loop for the first time, and then add `1+index**2/(4*index)` to Y each time through the loop, the value of Y during any cycle will be the value we want in the table. Thus, with K (the index) and Y being directly available, it is a simple matter to include a **PRINT** statement inside the loop, so that each cycle will produce the required values, and it will print an output

"Define Y,K."
"Print the headings."
"Initialize Y to zero."
**do** for k = 1 to 50 by 1:
    "Increase the current value in Y by 1 + K**2/(4*K)."
    "Print K,Y."
**enddo**
"Stop."

**FIGURE 10.9 (a)** Pseudocode Representation for Example 10.3.

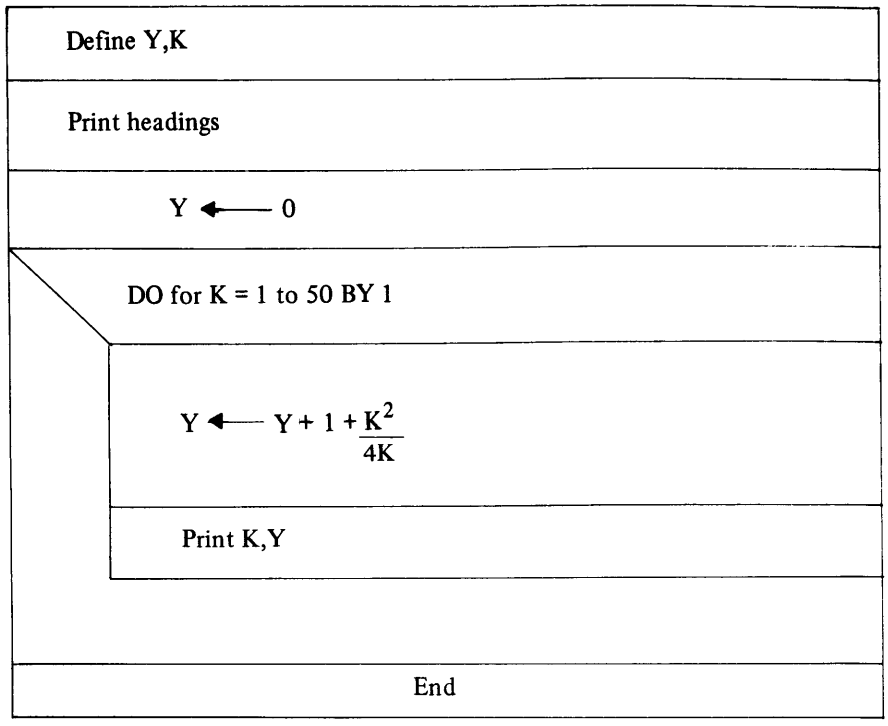



**FIGURE 10.9 (b)** N-S Diagram for Example 10.3.

```
C*********************************************************
C                      EXAMPLE 10.3                       *
C*********************************************************
C  THIS PROGRAM GENERATES A TABLE OF VALUES FOR Y AS      *
C  A FUNCTION OF K FOR K=1,2,3,...50, WHERE Y IS THE      *
C  SUM, FOR I=1 THROUGH K, OF THE QUANTITY                *
C                 1 + I**2/(4I)                           *
C  EACH LINE OF OUTPUT SHOWS K AND THE CORRESPONDING      *
C  VALUE OF Y.                                            *
C*********************************************************
      PROGRAM        EX1003
      IMPLICIT       NONE
      REAL*4         Y,REALK
      INTEGER*2      K

      Y = 0.0
      PRINT *, 'TABLE OF Y VALUES: Y=SUM OF I'S FROM ',
     1         '1 TO K OF (1+I**2/(4*I))'
      PRINT *, '       '
      PRINT *, '    K    ','    Y    '

C*********************************************************
C  NOW THAT Y IS INITIALIZED AND THE TABLE HEADINGS ARE   *
C  PRINTED, THE LOOP WILL BE USED TO PRODUCE THE ACTUAL   *
C  TABLE. REALK IS A TEMPORARY VARIABLE USED TO EXPRESS   *
C  K AS A REAL NUMBER. THIS IS A LITTLE MORE CONVENIENT   *
C  THAN HAVING TO CONVERT K TO REAL AS PART OF THE WORK   *
C  OF COMPUTING Y.                                        *
C*********************************************************

      DO K=1,50
        REALK = K
        Y=Y + (1.0+REALK*REALK/(4.0*REALK))
        PRINT *, K,Y
      END DO

      PRINT *, '  '
      PRINT *, 'END OF RUN. NORMAL TERMINATION.'
      STOP
      END
```

**FIGURE 10.10** FORTRAN Statements for Example 10.3.

line as part of the same process A flow diagram and pseudocode for this design are shown in Figure 10.9 and the program itself is seen in Figure 10.10. Figure 10.11 shows part of the resulting output, so that the action of the loop is clearly seen

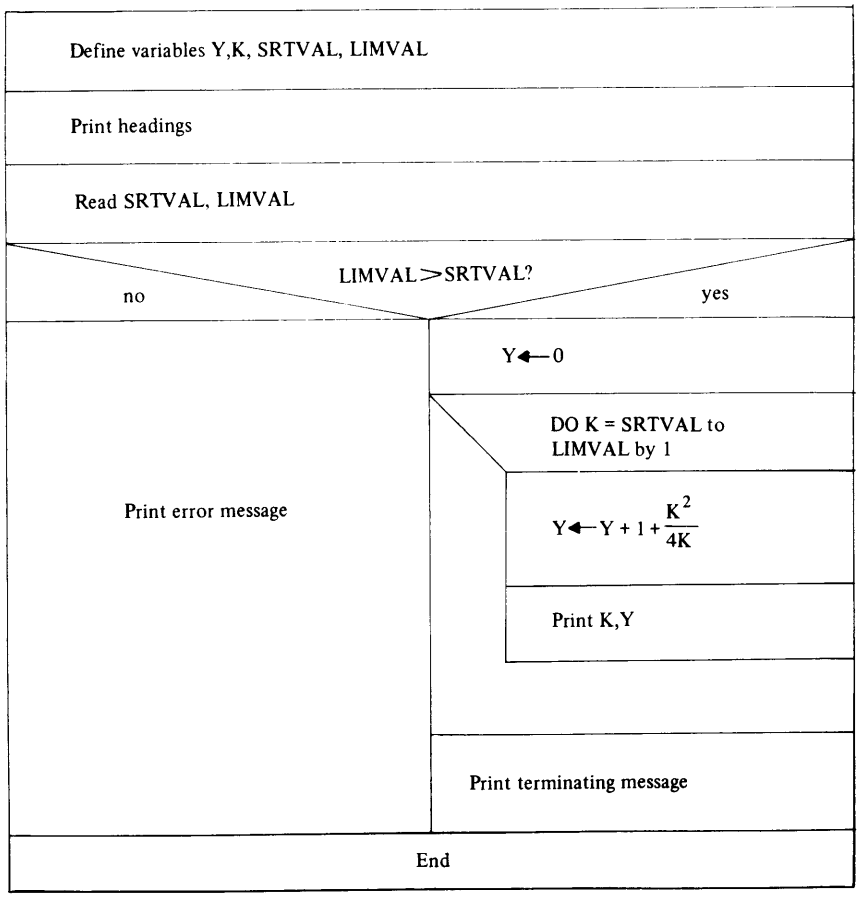
**10.2.1.2 Use of Variables as Loop Controllers** The number of times a loop is repeated in a given program need not be fixed as a permanent part of that program. Instead, *sv*, *lv*, and/or *incr* may vary from one run to another. In fact, any or all of these may vary from one usage to another in the same run. This can be done either by reading the values as input or by computing them earlier in the program. Regardless of where the specifica-

TABLE OF Y VALUES: Y = SUM OF I'S FROM 1 TO K OF (1+I**2/(4*I))

| K | Y |
|---|---|
| 1 | 0.1250000E 01 |
| 2 | 0.2750000E 01 |
| 3 | 0.4500000E 01 |
| 4 | 0.6500000E 01 |
| 5 | 0.8750000E 01 |
| 6 | 0.1125000E 02 |
| 7 | 0.1400000E 02 |
| 8 | 0.1700000E 02 |
| 9 | 0.2025000E 02 |
| 10 | 0.2375000E 02 |
| 11 | 0.2750000E 02 |
| . | ... |
| . | ... |
| 48 | 0.3420000E 03 |
| 49 | 0.3552500E 03 |
| 50 | 0.3687500E 03 |

END OF RUN. NORMAL TERMINATION.

**FIGURE 10.11** Portion of Output for Example 10.4.




**(a)**

```
"Define Y, K, SRTVAL, LIMVAL."
"Print the headings."
"Read SRTVAL, LIMVAL."
if
    LIMVAL is greater than SRTVAL
then
    "Initialize Y to zero."
    do from K = SRTVAL to LIMVAL by 1:

    Y's current value by 1 + K**2/(4*K)."

    K,Y."
    enddo
    "Print terminating message."
else
    "Print an error message."
endif
"Stop."
```

**(b)**

**FIGURE 10.12** **(a)** N-S diagram for Example 10.4. **(b)** Pseudocode Representation for Example 10.4.

tions come from or how they are acquired, their role as loop controllers is the same as that for constant values. Thus, it is perfectly legal to write a DO statement such as

```
DO COUNT = SVAL, LIMIT, AMT
```

as long as SVAL, LIMIT and AMT have been declared properly and contain appropriate values.

**Example 10.4** We shall generalize the previous example by redesigning it so that *sv* and *lv* are variables submitted as input. As seen in Figures 10.12 and 10.13, the revision itself is simple. However, the use of variable controllers does introduce an additional concern. The fact that the starting value, limiting value and increment can be variables means that their values will not be "known" to the program until the instant they actually are used during the run. Consequently, we can no longer leave it to FORTRAN to check whether the control values make any sense. For instance, if we write a statement like

```
DO INDX = 10,5,1
```

the control values are right there and FORTRAN "sees" that the limiting value (5) is less than the starting value (10). (More about this in Section 10.2.1.6.) However, if the loop's controls are variables, FORTRAN has no choice but to assume that the programmer will make sure the values in those variables will be consistent when the program runs.

This additional responsibility is reflected in this program (Figure 10.13) by the inclusion of comparisons among the control values. The test in this example is a simple one, its purpose being to emphasize the need for such checking. In other situations, it may be necessary to introduce more elaborate tests. Of course, the final decision as to what kind of testing is appropriate rests with the programmer.

**10.2.1.3 Mismatches Between the Limiting Value and Increment** Because of the flexibility provided by the controls in the DO statement, there is no reason to expect every situation to be one in which the index reaches an exact match with the limiting value. For instance, suppose we had an array of numerical values declared as

```
REAL*4 XVAL(20)
```

and we set up the following DO loop:

```
DO NXV = 1,20,3
   XVAL(NXV) = 3.7 * XVAL(NXV)
END DO
```

there is no question that the loop will work; the question is, what will it do? This is no particular mystery as long as we retain the idea that the decision to repeat or bypass the loop is related to whether the index *exceeds* the limiting value. Applying that rule to this situation makes the loop's progress clear: starting with an NXV of 1, the loop calculates a new value for XVAL(1). Then, NXV is increased by the specified increment (i.e., 3) resulting in a value of 4. Since the limiting value is not exceeded, the loop repeats, changing the value in XVAL(4). NXV is increased to 7, and the loop repeats, as it continues to do with NXV at 10, 13, 16, and 19. After changing the value in XVAL(19), NXV is increased (by 3) to 22. This time, the limiting value is exceeded and the loop is bypassed after having been processed seven times. Note that XVAL(20) was not affected even though there is an XVAL(20) and even though the loop has a limiting value of 20. It just turned out that the combination of the starting value and increment made it impossible for the program to produce an index value that was exactly equal to the limiting value. No crime in that.

It is easy to predict exactly what will happen during such mismatches if we understand how FORTRAN handles a DO loop. This can be described in terms of a rule that governs FORTRAN's use of the loop's controls, namely, *sv*, *lv*, and *incr*. When a program

```
C*************************************************************
C                      EXAMPLE 10.4                           *
C*************************************************************
C  IN THIS REVISION OF EXAMLE X.3, THE LOOP'S STARTING  *
C  VALUE (SRTVAL), LIMITING VALUE (LIMVAL) AND INCREMENT*
C  (INCR) ARE READ AS INPUT.                             *
C*************************************************************

      PROGRAM        EX1004
      IMPLICIT       NONE
      REAL*4         Y,REALK
      INTEGER*2      K,SRTVAL,LIMVAL

      PRINT *, 'TABLE OF Y VALUES: Y=SUM OF I'S FROM
     1          1 TO K OF (1+I**2/(4*I))'
      PRINT *, 'STARTING VALUE FOR K:',SRTVAL
      PRINT *, 'LIMITING VALUE FOR K:',LIMVAL
      PRINT *, '      '
      PRINT *,'    K    ','    Y    '
      PRINT *,'         '
      PRINT *, 'SUBMIT VALUES FOR SRTVAL AND LIMVAL'
      READ *, SRTVAL,LIMVAL

C---BEFORE WE START ACTUAL PROCESSING, WE CHECK TO MAKE SURE
C---THAT THE LOOP'S CONTROL VALUES ARE CONSISTENT.

      IF(LIMVAL .GT. SRTVAL) THEN
        Y = 0.0
        DO K = SRTVAL,LIMVAL,1
          REALK = K
          Y = Y+(1.0+REALK*REALK/(4.0*REALK))
          PRINT *, K,Y
        END DO
        PRINT *, '     '
        PRINT *, 'END OF RUN. NORMAL TERMINATION.'
      ELSE
        PRINT *, 'IMPROPER STARTING VALUE AND/OR INCREMENT'
      END IF

      STOP
      END
```

**FIGURE 10.13** FORTRAN Statements for Example 10.4.

reaches a DO statement, FORTRAN computes a number that indicates (to it) how many times the loop will repeat. This number, called the *iteration count*, is determined by the following formula:

$$\text{iteration count} = \text{MAX}(\text{INT}((lv - sv + incr)/incr), 0)$$

MAX and INT describe the actions performed by the built-in functions with these names (see Chapter 6). Thus, the DO statement in the previous illustration, i.e.,

```
DO NXV = 1, 20, 3
```

produces an iteration count of

```
   MAX(INT((20-1+3)/3),0)
= MAX(INT(22/3),0)
= MAX(7,0)
= 7
```

This count is set up and maintained outside of the programmer's direct control. Then, each time the loop is processed, the count is decreased by 1. The loop repeats until the count reaches zero, at which point the loop is bypassed. Some of the iteration limits can be summarized as follows:

1. If the iteration count is zero to begin with, the loop is skipped.
2. If the loop index is declared as INTEGER*2, the maximum possible number of trips through the loop is 65536 (i.e., $2^{16}$). For instance, if NTR is a loop index declared as a 16-bit integer, we could write

   ```
   DO 12 NTR = -32768, 32767
   ```

3. If the loop index is declared as INTEGER*4, the maximum possible number of trips through the loop is $2^{32}$. To illustrate, suppose CYCLES is declared as INTEGER*4. The following statement

   ```
   DO 14 CYCLES = -2**31, 2**31-1
   ```

   sets up a loop with that maximum number.

**10.2.1.4 Use of an Index Outside Its Loop** Once a program is outside of a DO loop, the index (i.e., the DO-variable) that was used by that particular loop retains the last value that it had. In the case of the previous example, that means that after the seventh time through the loop, when the index (NXV) was at 19, there was an attempt at another repetition, with NXV at 22. When that attempt failed and the loop was bypassed, NXV was left with a value of 22.

The same thing is true when there is an exact match between the DO-variable and limiting value. For instance, suppose the following loop

```
DO NTIMES = 1,30
   action
END DO
PRINT *, NTIMES
```

contained nothing inside it that would interrupt the 30 cycles. Consequently, the output statement would produce a value of 31 because after the 30th time through, there would be another (thwarted) attempt with NTIMES set at 31. The fact that FORTRAN turns the 31st attempt aside is seen when we introduce a small addition to the sequence of statements:

```
DO TIMES=1,30
   MYVAL = NTIMES
   action
END DO
PRINT *, MYVAL,NTIMES
```

The two values printed will be 30 and 31.

Of course, if something happens inside a loop that forces the program to break out of the loop before the predefined number of cycles has been completed, the DO-variable will show the value it had when entering the cycle that was interrupted.

**Example 10.5** Suppose we read in a value Y, followed by a succession of values for X. Let us say that there is an endless supply of X's. We wish to add the X's together, as they come in, accumulating the sum in a variable named TOTALX. This will continue until TOTALX reaches a value that is at least as large as Y. When that happens (and we guarantee it to happen) the program is to print Y, TOTALX and NUMX, the number of values it took to achieve the result.

We can set this up as a DO loop with a limiting value that we feel is so high that, for all practical purposes, we are saying, "keep on doing this, forever if necessary, until something happens in the loop to stop the repetitions." NUMX will be the index, so that we can take advantage of the DO loop's automatic counting mechanism. The program is shown in Figure 10.14.

**10.2.1.5 Negative Increments** FORTRAN 77 accepts DO statements such as

```
DO NUM = 20,1,-1
```

so that it is equally convenient to construct loops in which the index becomes smaller with each cycle. Here again, it is up to the programmer to make sure that the starting value is greater than the limiting value when the increment is negative. Note that the number of cycles formed by the program still is determined by the same rule described for a positive

```
************************************************************
*                     EXAMPLE 10.5                         *
************************************************************
*  THIS PROGRAM READS A VALUE Y, FOLLOWED BY AN ENDLESS    *
*  SUPPLY OF VALUES FOR X. A SUM (TOTALX) OF ALL THE       *
*  X'S IS ACCUMULATED UNTIL THAT SUM REACHES OR EXCEEDS    *
*  Y, AT WHICH POINT THE PROGRAM PRINTS Y, TOTALX, AND     *
*  NUMX, THE NUMBER OF X'S REQUIRED TO DO THIS.            *
************************************************************

      PROGRAM          EX1005
      IMPLICIT         NONE
      REAL*4           Y,X,TOTALX
      INTEGER*4        NUMX

      TOTALX = 0.0
      PRINT *, 'ENTER A VALUE FOR Y'
      READ *, Y

      DO NUMX = 1,100000
         PRINT *, 'ENTER A VALUE FOR X'
         READ *, X
         TOTALX = TOTALX + X
         IF (TOTALX .GE. Y) GO TO 99
      END DO

   99 PRINT *, 'Y = ',Y
      PRINT *, 'THE SUM OF THE X VALUES = ',TOTALX
      PRINT *, 'THE NO. OF X'S IN TOTALX = ',NUMX
      STOP

      END
```

**FIGURE 10.14** FORTRAN Statements for Example 10.5.

increment. For example, suppose we had the following loop:

```
DO K = 20, 3, -6
   action
END DO
```

In this loop, *sv* is 20, *lv* is 3, and *incr* is −6. Accordingly, the iteration count will be computed as

```
   MAX(INT((3-20-6)/-6), 0)
= MAX(INT(-23/-6), 0)
= MAX(3, 0)
= 3
```

If we follow the loop's progress, then it is clear that this iteration count is what we would expect it to be. The first time through the loop, K is 20. Adding the increment (−6) then reduces K to 14 and the iteration count drops from 3 to 2. After the second cycle, K becomes 8 (i.e., 14−6) and the iteration count is 1. Once the third cycle is completed, K is decreased to 2, and the iteration count is zero, so that further cycles are avoided. Thus, if we were to print K's value just beyond statement 12, it would appear as 2.

**Example 10.6** To illustrate the use of a negative increment, we shall design a program that reads character strings (words) and inserts them in their proper places in an alphabetized list (array) of words. For simplicity, we shall assume that there is an initial list available. That is, we shall bring in this list before the first "new word" is read and processed. Our input setup, then, will require two signals: one to indicate the end of the original list, and the other to indicate the end of the list of new words. The following additional assumptions will be made:

1. The longest word will contain ten letters.
2. The original list may contain up to 50 words, and it is already in alphabetical order. The last word is followed by the four letters 'XXXX'.
3. No more than 50 words will be added, but there will be at least one new word. The last word is followed by the four letters 'ZZZZ'.
4. There are no duplicate words either in the original list or the group of new words.

The program will print the original list (and the number of words in it), followed by the newly enlarged list, along with its number. Figure 10.15 shows the flowchart and pseudocode, and Figure 10.16 shows the program's statements.

This program also shows another example of the ability to break out of a DO loop before the prescribed number of cycles has been completed. Note that the loop which finds NEWWORD's proper place in WDLIST has an unpredictable exit based on the outcome of a test inside the loop. This technique is useful in many different applications.

**10.2.1.6 Extreme Conditions in Loop Controllers** Since sv, lv, and incr may vary independently, it is altogether possible for conditions to develop in which the starting and limiting values come out to be identical. Consequently, it is useful to know what FORTRAN does when this happens. Here again, the matter is relatively straightforward if we compare the situation against FORTRAN 77's rule for computing the iteration count. To focus on the behavior of the loop, we shall look at an unrealistically simple example in which we assume that N and K have been declared earlier as integer variables:

```
N = 1
DO K=1, N
   action
END DO
```

(In an actual program, the limiting value (N in our example) is likely to be determined in a

```
"Define   NUMOLD,NUMNEW,NUMLST,INHERE,NMOV,
          WDLIST(100), and NEWWRD."
"Initialize NUMOLD, NUMNEW and NUMLST to 0."
"Print the headings."
"Initialize NUMOLD to 1."
do
    "Read the next word into WDLIST(NUMOLD)."
    "Echo the word just read."
    "Add 1 to NUMOLD."
until the word just read is 'XXXX'.
"Decrease NUMOLD by 1 so as not to count the 'XXXX'."
"Set NUMLST to NUMOLD."
do
    "Read a new word into NEWWRD."
    "Add 1 to NUMNEW."
    "Initialize INHERE to 1."
    do
        "Compare NEWWRD to WDLIST(INHERE)."
        "Add 1 to INHERE."
    until NEWWRD is alphabetically before WDLIST(INHERE)
    do for each WDLIST entry from the end of the list
to
    WDLIST(INHERE):
        "Shift an element in WDLIST one position further
back in the WDLIST array."
    enddo
    "Place NEWWRD in the vacated element WDLIST(INHERE)."
until the last new word has been processed ('ZZZZ')."
"Print the final word list."
"Stop."
```

**FIGURE 10.15** **(a)** Pseudocode for Example 10.6.

more complicated way.) Our major concern, however, is with the result, namely, the fact that for this situation, the starting value and the limiting value are the same. Since *incr* is 1 by default, the number of times that the loop will be performed is

```
   MAX(INT((1-1+1)/1),0)
= 1
```

so that there will be one trip through the loop. The value of K after statement 30 will be 2.

Another, more extreme situation is shown below:

```
N = 1
DO L=5,N
   action
END DO
PRINT *, L
```

Since *sv* is greater than *lv*, the value of *incr* should be negative, and it is not. Following the same general rule that guided us earlier, we find that the resulting increment count is

```
   MAX(INT((1-5+1)/1),0)
= MAX(-3,0)
= 0
```

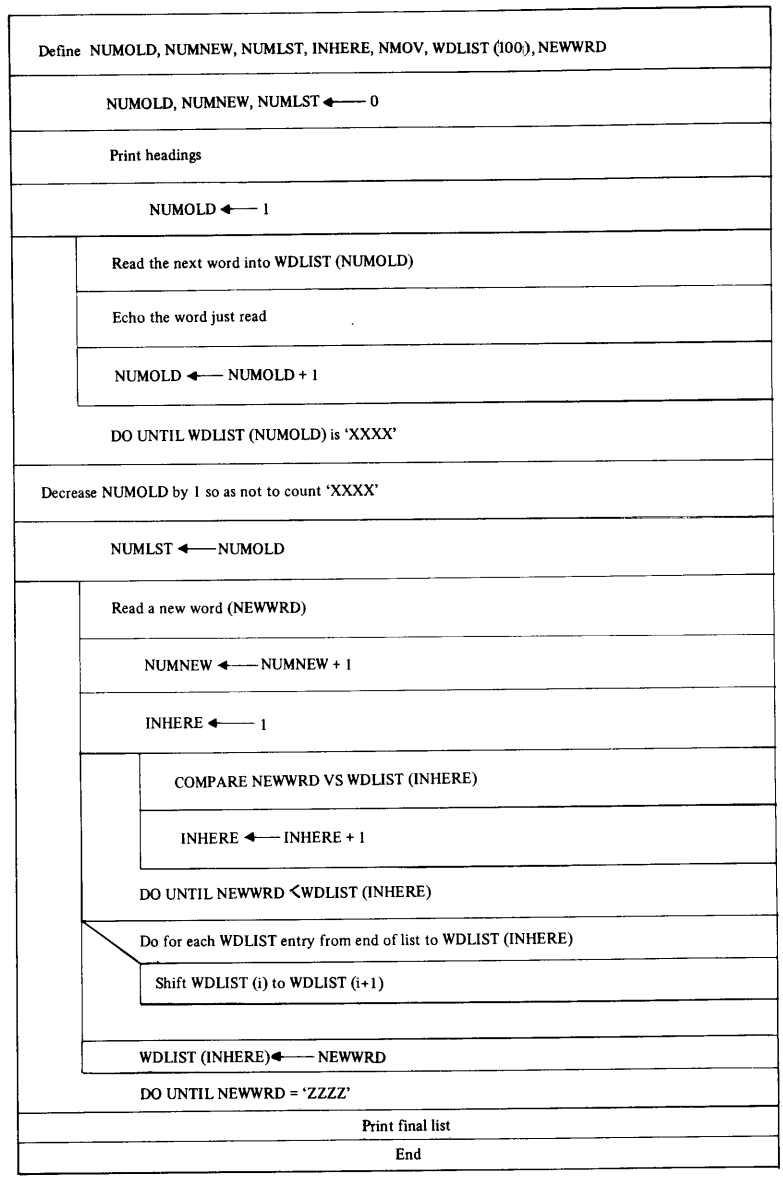



**FIGURE 10.15 (b)** N-S Diagram Flowchart for Example 10.6.

and this is exactly what FORTRAN 77 uses. In other words, it bypasses the loop completely, even before the first cycle.

**10.2.1.7 Non-Integer Controller Values** Now that we are acquainted with the general operation of a DO loop, we can turn our attention to the fact that there is even greater flexibility in the way we can specify the three control items. Besides being able to use constants or variables, any or all of the controls may be written as expressions. *Furthermore, it is not necessary to restrict them to integers.* Any legitimate integer, real or double precision expression (in accordance with the structural rules discussed in Chapters 5 and 6) may be used. Similarly, the DO-variable may be integer, real, or double precision.

The action of a loop under such controls is easily demonstrated by means of a little example. We return to the type of computations illustrated in Example 10.3. This time, however, the desired mathematical function involves fractional quantities, and the table

```
C*********************************************************
C                    EXAMPLE 10.6                         *
C*********************************************************
C  THIS PROGRAM READS A LIST OF NO MORE THAN 50 WORDS     *
C  (WDLIST) IN ALPHABETICAL ORDER. THEN, IT READS A       *
C  SUCCESSION OF NO MORE THAN 50 ADDITIONAL WORDS         *
C  AND PLACES EACH OF THEM IN ITS PROPER ALPHABETICAL     *
C  POSITION IN THE LIST. THE LAST WORD IN THE INITIAL     *
C  LIST IS FOLLOWED BY 'XXXX' AND THE LAST WORD IN THE    *
C  LIST OF NEW WORDS IS FOLLOWED BY 'ZZZZ'.               *
C     WDLIST: THE ARRAY OF WORDS                          *
C     NEWWRD: A WORD TO BE INSERTED IN WDLIST             *
C     NUMOLD: NO. OF WORDS IN ORIGINAL LIST               *
C     NUMNEW: NO. OF NEWLY ADDED WORDS                    *
C     NUMLST: NO. OF WORDS IN THE ENLARGED LIST           *
C     INHERE: POSITION IN WDLIST FOR A NEW WORD           *
C     NMOV:   A VARIABLE FOR USE AS AN INDEX              *
C*********************************************************

      PROGRAM          EX1006
      IMPLICIT         NONE
      INTEGER*2        NUMOLD,NUMNEW,NUMLST,INHERE,NMOV
      CHARACTER        WDLIST(100)*10, NEWWRD*10, ENDOLD*4, ENDNEW*4
      PARAMETER        (ENDOLD='XXXX',ENDNEW='ZZZZ')

      NUMNEW = 0
      NUMLST = 0
      PRINT *, 'ORIGINAL LIST:'
      PRINT *, '      '

C--FIRST, WE BUILD THE INITIAL LIST. SINCE WE KNOW THAT IT WILL
C--CONTAIN NO MORE THAN 50 WORDS, WE SHALL SET UP A LOOP WITH 60
C--CYCLES AND BREAK OUT OF IT WHEN WE FIND ENDOLD. AT THAT POINT,
C--THE INDEX (NUMOLD) WILL BE 1 GREATER THAN THE NUMBER OF WORDS
C--BECAUSE ITS COUNT WILL INCLUDE THE 'XXXX' SIGNAL.

      PRINT *, 'ENTER THE ORIGINAL LIST'
      DO NUMOLD = 1,60,1
        READ *, WDLIST(NUMOLD)
        IF (WDLIST(NUMOLD) .EQ. ENDOLD) GO TO 11
        PRINT *, WDLIST(NUMOLD)
      END DO

C--NOW, WE READ A NEW WORD, FIND OUT WHERE IT GOES, AND
C--INSERT IT THERE BY MOVING ALL THE WORDS IN WDLIST BEYOND
C--THAT POSITION, THEREBY OPENING A HOLE FOR THE NEWCOMER.
C--EVERY TIME THIS HAPPENS, OF COURSE, THE SIZE OF THE LIST
C--INCREASES BY 1.

   11 NUMOLD = NUMOLD-1
      NUMLST = NUMOLD
```

**FIGURE 10.16** FORTRAN Statements for Example 10.6.

```
      PRINT *, 'TYPE IN A NEW WORD'
      READ *, NEWWRD
      DO WHILE (NEWWRD .NE. ENDNEW)
         NUMNEW = NUMNEW+1
         DO INHERE=1,NUMOLD
            IF (NEWWRD .LT. WDLIST(INHERE)) GO TO 21
         END DO

C--THE ONLY WAY WE CAN REACH THIS POINT IN THE PROGRAM IS
C--FOR NEWWORD TO BE ALPHABETICALLY 'LATER' THAN THE LAST
C--WORD IN THE LIST SO FAR.

         INHERE = NUMLST+1
         GO TO 31

C--MOVE THE REST OF THE ARRAY TO OPEN A SPOT

   21    DO NMOV = NUMLST,INHERE,-1
           WDLIST(NMOV+1) = WDLIST(NMOV)
         END DO

   31    WDLIST(INHERE) = NEWWRD
         NUMLST = NUMLST+1
         PRINT *, 'SUBMIT THE NEXT NEW WORD'
         READ *, NEWWRD
      END DO

   99 PRINT *, 'ENLARGED LIST:'
      PRINT *, '      '
      DO NMOV = 1,NUMLST,1
        PRINT *, WDLIST(NMOV)
      END DO
      PRINT *, 'NO. OF WORDS IN ENLARGED LIST = ',NUMLST
      STOP
      END
```

**FIGURE 10.16** *(continued)*

we wish to generate will have entries separated by fractional increments. The desired function is

$$\text{LOG (VP)} = A + B/\text{TEMP} + C*\text{TEMP}**2$$

where VP is a pressure, TEMP is a temperature, and A, B, and C are constant values. Input to this program consists of A, B, C, and two temperature values TEMP1 and TEMP2. Using this information, the program is to produce a table of values for VP (not the logarithm of VP) for temperatures beginning with TEMP1 and going up by 0.1 degrees through TEMP2.

We shall dispense with the other parts of the program and concentrate on the loop itself; the declarations will be shown to provide context:

```
REAL*4          A, B, C, TEMP, TEMP1, TEMP2, VP, LOGVP
  . .                          . . . . . .
  . .                          . . . . . .
DO TEMP = TEMP1, TEMP2, 0.1
  LOGVP = A + B/TEMP + C*TEMP*TEMP
  VP = 10.0**LOGVP
  PRINT *. TEMP, VP
END DO
  . .                          . . . . . .
END
```

The number of cycles through this loop still is determined in the same way. Using values of 275.5 and 300.5 for TEMP1 and TEMP2 to illustrate, the program would be working with the iteration count formula as follows:

```
iteration count = MAX(INT((300.5–275.5+.1)/.1),0)
                = MAX(INT(25.1/.1),0)
                = MAX(251,0)
                = 251
```

Accordingly, the loop will generate and print 251 sets of TEMP and VP values, and the final value of TEMP to be carried outside the loop will be (that's right) 300.6.

### 10.2.2 Nested DO Loops

One simple rule governs the construction of a nested loop: *it must be completely surrounded by another loop.* (A pictorial representation is seen in Figure 10.17.) Organizationally, this is the same type of structure that applies to nested IF-blocks (Section 9.1.2.3). There is no particular limit as to how many loops may be nested within one another. As is true with many other features, the actual limit should be set by the programmer based on preserving a program's simplicity and clarity. If, for instance, a programmer finds it necessary to set up nineteen levels of nested loops, it is a good bet that the procedure is too complex and should be reexamined.

We shall take the same kind of approach with nested DO loops as was done with single loops: while FORTRAN accepts several different structural forms within the basic nesting rule, it turns out that most of them create more confusion than convenience. Consequently, we shall follow two standard practices:

1. Each loop will end with its own CONTINUE statement, whose placement is lined up with the corresponding DO statement.
2. The nesting structure will be emphasized by identing a nested loop within the loop that surrounds it. These forms are shown in Figure 10.17.

**10.2.2.1 Processing of Nested Loops** The execution of nested loops follows a pattern that is completely consistent with that established for a single loop. Since each loop has its own DO statement (and, therefore, its own controls), separate counters keep track of each set of cycles. Basically, an inner loop will go through a complete set of repetitions for each cycle of the surrounding loop, in much the same way that a minute hand on a clock (you remember clocks with hands on them?) will go through a complete set of movements (i.e., 60) before the hour hand moves once.

A look at a set of nested loops will demonstrate this nicely. Suppose we have the following situation:

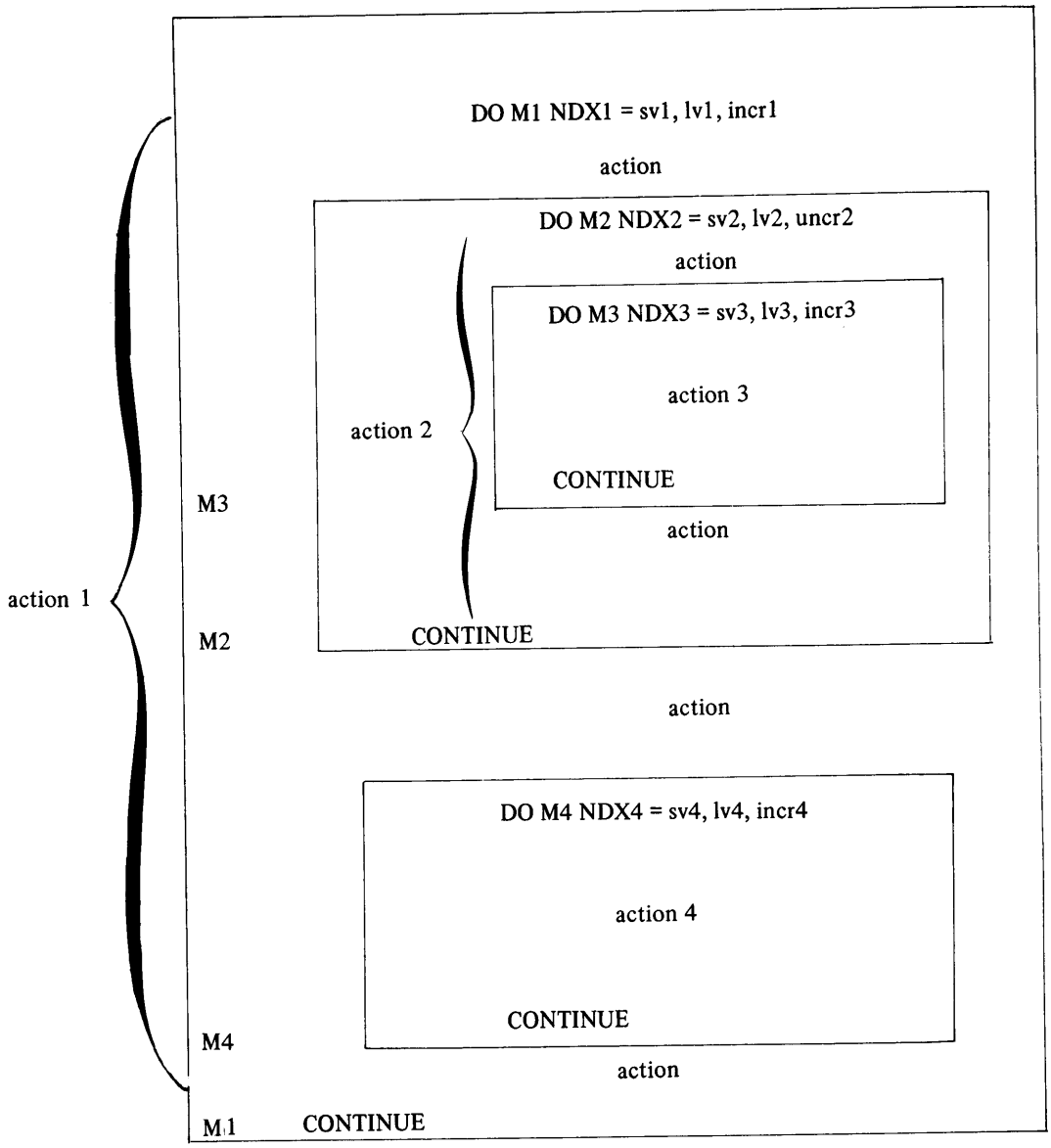



**FIGURE 10.17** Structure of Nested DO Loops.

```
    ....
DO OUTER = 1,10,1
   action 1
   DO INNER = 1,5,1
      action 2
   END DO
   action 3
END DO
action 4
   ....
```

Assuming there is nothing in action 1, action 2, or action 3 to cut short the specified number of cycles in either loop, execution will proceed as follows:

1. Processing of the first DO statement initializes OUTER to 1 (assuming OUTER was declared previously as INTEGER*2) and establishes an iteration count of 10 for the outer loop.
2. The processing in action 1 is performed.
3. The inner DO statement is executed, thereby initializing INNER to 1 and establishing an iteration count of 5 for that loop. This means that the inner loop will cycle five times for each cycle of the outer loop.
4. The processing in action 2 is performed.
5. The conclusion of the inner loop is reached, thereby triggering an increase in the loop's DO-variable (INNER) and a decrease in the iteration counter. When the iteration counter reaches zero, processing continues at Step 6; until that happens, processing repeats from Step 4.
6. This point is reached after each five cycles of the inner loop. Accordingly, action 3 is performed.
7. Since the next statement (statement 12) indicates the end of the outer loop, OUTER is increased by 1 and the loop's iteration counter is decreased by 1. If that counter has reached zero, the program avoids another cycle and moves on to Step 8; otherwise, it repeats the processing, starting at Step 2.
8. The processing of the nested loop structure is completed, and the program goes on to perform action 4.

Thus, we see that the outer loop is processed ten times. Each of these cycles includes one performance of action 1, *five* performances of action 2, and one performance of action 3.

**10.2.2.2 Processing of Multidimensional Arrays** If we have the following declarations:

```
REAL*4 XVAL(5, 4)
INTEGER*2 ROW, COL
```

and we specify the statement

```
READ *, XVAL
```

FORTRAN will read the next 20 input values and place them in XVAL in the following order:

```
XVAL(1, 1), XVAL(2, 1), XVAL(3, 1), . . . , XVAL(5, 1), XVAL(1, 2),
    XVAL(2, 2), XVAL(3, 2), . . . , XVAL(3, 4), XVAL(4, 4), XVAL(5, 4).
```

This is in exact accordance with FORTRAN's rules for the organization of multidimensional arrays (as defined in Chapter 8). Consequently, in order for this convenient statement to produce a desired result, it is necessary to arrange the input values so that their order matches that assumed by FORTRAN. When this is not the case, we cannot take advantage of the automatic input operation. Instead, we have to control the sequence ourselves, and an easy way to do this is with nested loops. For example, assuming the same declarations given above, if we were to write the following program segment:

```
DO ROW = 1, 5, 1
  DO COL = 1, 4, 1
    READ *, XVAL(ROW, COL)
  END DO
END DO
```

we still would read 20 input values (because the inner loop cycles four times for each of the outer loop's five cycles). However, the *locations* in which these 20 values will be stored will follow a different sequence:

```
XVAL(1,1), XVAL(1,2), XVAL(1,3), XVAL(1,4), XVAL(2,1), . . . ,
    XVAL(2,4), XVAL(3,1), . . . , XVAL(5,3), XVAL(5,4).
```

The same type of flexibility can be used when doing internal computations. For example, suppose we had a 5×5 array X of real numbers and we wanted to produce another array Y, also 5×5, such that each row in Y is taken from the corresponding column in X. (In mathematical terminology, we would say that Y is the *transpose* of X.) The appropriate loops would be written as follows:

```
REAL*4 X(5,5), Y(5,5)
INTEGER*2 ROW, COL
        . . . .
        . . . .
  input activity for X
        . . . .
        . . . .
DO ROW = 1,5,1
  DO COL = 1,5,1
    Y(COL,ROW) = X(ROW,COL)
  END DO
END DO
```

Another type of operation that often is used with two-dimensional arrays is one in which a particular row is selected as a divisor. Then, the elements in each of the rows are divided by the corresponding elements of the selected divisor row. As a result, the value in each element of the divisor row becomes 1.0 and the other rows' elements are changed accordingly. The following program segment shows this type of computation using a 5×6 array T, with the second row selected as the divisor row:

```
REAL*4 T(5,6)
INTEGER*2 ROW,COL
        . . . .
        . . . .
DO ROW = 1,5
  DO COL = 1,6
    T(ROW,COL) = T(ROW,COL)/T(2,COL)
  END DO
END DO
      . . . .
      . . . .
```

**10.2.2.3 Sorting** One of the most frequently used computing processes is *sorting*—the rearrangement of a collection of data according to a required sequence. There may be all kinds of different requirements involving the widest imaginable types of data values, but the objective remains the same: production of an organized collection in which the order of the individual items has been explicitly defined. The tremendous diversity of such sorted collections is emphasized by the few examples given below:

1. A list of inventory items in a warehouse, sorted by increasing part number.
2. A collection of words, sorted in alphabetical order.

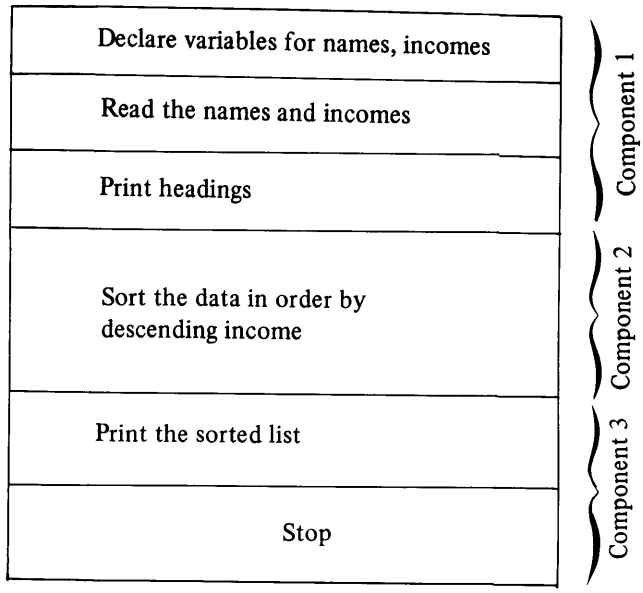
3. a collection of words taken from a particular essay or story, sorted in order of decreasing use (i.e., the most frequently appearing word first, etc.).
4. A series of temperature readings, sorted in chronological order (earliest reading first, etc.).
5. A list of American cities, sorted by population in descending sequence (i.e., highest population first).
6. A list of baseball players, sorted by batting average in descending sequence.
7. A list of Mozart's works, sorted by the dates in which they were completed (earliest work first).
8. A list of marchers in a parade, sorted by increasing height.

We can see that the possibilities are endless. In fact, successful books have been written with nothing but sorted lists in them.

When these sorted lists become long, or change often, it is particularly convenient to manipulate them on a computer. Accordingly, there are numerous algorithms for sorting such data; the selection of the most effective one for a particular situation is not always straightforward. For purposes of illustration, we shall use a rather simple procedure to sort a one-dimensional array of real numbers so that the resulting elements will be in descending order.

**Example 10.7** Nostril Surveys ("We Use Our Nose For News To Sniff Out The Truth") has just completed gathering data on the one hundred highest paid sword swallowers (for the Holy and Loyal Federation of Albanian Sword Swallowers) and wishes to publish the list arranged in descending order. Data for each swallower, consisting of the name (25 characters or less) and salary (in Bavvoozniks, to the nearest hundredth of a Bavvooznik), are recorded on a separate line. The cards are not in any particular order, so that the required program is to read them in, sort them, and print the sorted list. Figure 10.18 shows the overall outline of the program.

Component 1 (Figure 10.18) is simple enough. We know we shall need arrays to hold the 100 names (NAME) and corresponding incomes (INCOME). A simple DO loop will read the data in preparation for the sorting operation. This activity, together with the display of a set of headings, completes the definition of




(b)

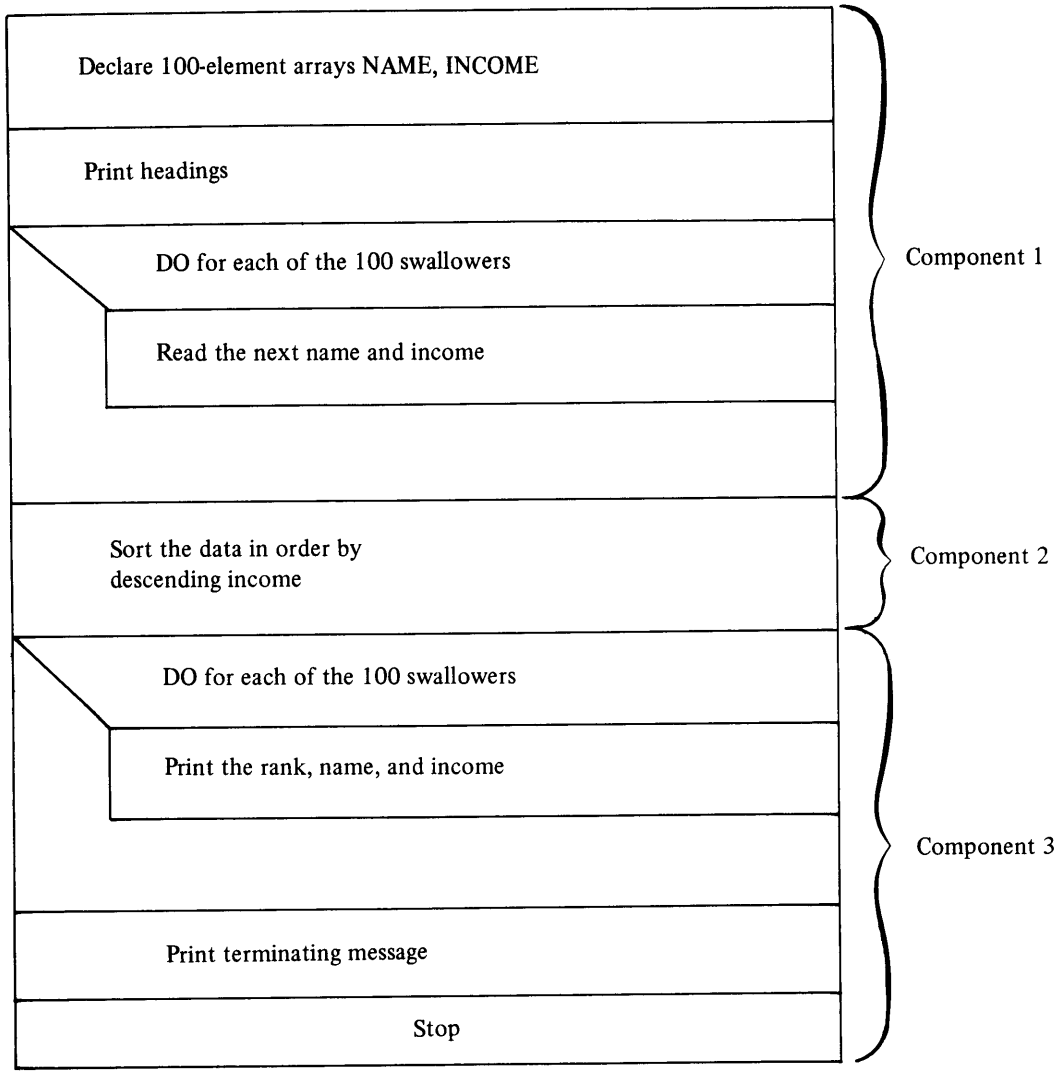
"Declare variables for names and incomes." component 1
"Read the 100 sets of names and incomes."
"Print the headings." component 2
"Sort the 100 sets of incomes in descending order (i.e., highest income first)." component 3
"Print the sorted list of incomes and corresponding names."
"Stop."

(a)

**FIGURE 10.18** **(a)** Overall Pseudocode Representation for Example 10.7. **(b)** Overall N-S diagram for Example 10.7.

"Declare 100-element arrays NAME and INCOME."
"Print headings." component 1
**do** for each of the 100 swallowers:
    "Read and store the name and income."
**enddo**
"Sort the data in order by descending income." component 2
**do** for each of the 100 swallowers:
    "Print the next name, rank, and income." component 3
**enddo**
"Stop."

**(a)**




**(b)**

**FIGURE 10.19** **(a)** Partially Developed Pseudocode for Example 10.7. **(b)** Partially Developed N-S Diagram for Example 10.7.

component 1 (at least for now), and a more detailed description can be given. (This is seen in Figure 10.19.)

Component 3 (Figure 10.18) is even more straightforward, so that its description can be developed merely by showing the display of the sorted list as another DO loop that repeats 100 times. This is shown in Figure 10.19, along with the undeveloped description for component 2.

Now we can turn to component 2, the actual sorting operation. The procedure to be used will find the largest income value and move it, along with the corresponding name, to the first position in the list. Then, the largest of the remaining income values will be moved to the second position. This pattern will be repeated until there are only two entries left to be sorted. Once the larger of the two has been found and positioned, the process is complete.

This can be implemented as follows:

```
DO for every position i from 1 through 99
   DO for every position j from 100 through i+1 by −1
      IF income (j) > income (j−1) THEN
         exchange
      END IF
   END DO
END DO
```

Thus, the first position ends up with the largest entry. Once that has been established, there is no reason to pay any further attention to that position. In a sense, the first position is "closed off" and the procedure begins another round of comparisons. The object this time is to place the largest *remaining* value in element 2. Thus, at the end of the second round, the two largest values will be in positions 1 and 2. Since we do not need to look further at these positioned values, each successive round of comparisons involves a smaller and smaller part of the array, until there is nothing left to compare. Figure 10.20 illustrates this process, step-by-step, for an array of five numbers.

```
START:                    2   7   5   6   8
AFTER COMPARISON 1:       2   7   5   8   6
AFTER COMPARISON 2:       2   7   8   5   6
AFTER COMPARISON 3:       2   8   7   5   6
AFTER COMPARISON 4:       8   2   7   5   6
                  SECOND ROUND
AFTER COMPARISON 5:       8   2   7   6   5
AFTER COMPARISON 6:       8   2   7   6   5
AFTER COMPARISON 7:       8   7   2   6   5
                   THIRD ROUND
AFTER COMPARISON 8:       8   7   2   6   5
AFTER COMPARISON 9:       8   7   6   2   5
                  FOURTH ROUND
AFTER COMPARISON 10:      8   7   6   5   2
                     FINISH
```

**FIGURE 10.20** Step-by-Step Illustration of a Bubble Sort.

An additional feature of this "bubble-sort" algorithm is the flag NOSWAP. This logical variable is used to determine whether any exchange(s) took place during the previous sweep through the array. If there were no exchanges, it means that the values are already in proper order and no more cycles are needed.

Since we are concerned with both a name and an income for each entry, it will be necessary to keep the name together with its matching income. An easy way to do this is to set up two arrays (100 elements each) so that corresponding elements will contain information for the same person. (We have already established this idea when we looked at component 1 of the program.) Thus, when we swap the contents of two positions, there will need to be two sets of exchanges: one for the name, and the other for the income. For instance, if NAME(J) and INCOME(J) are to be exchanged with NAME(J−1) and INCOME(J−1), it will take six statements to do so. Using TMPNAM and TMPINC for temporary storage, a swap could look as follows:

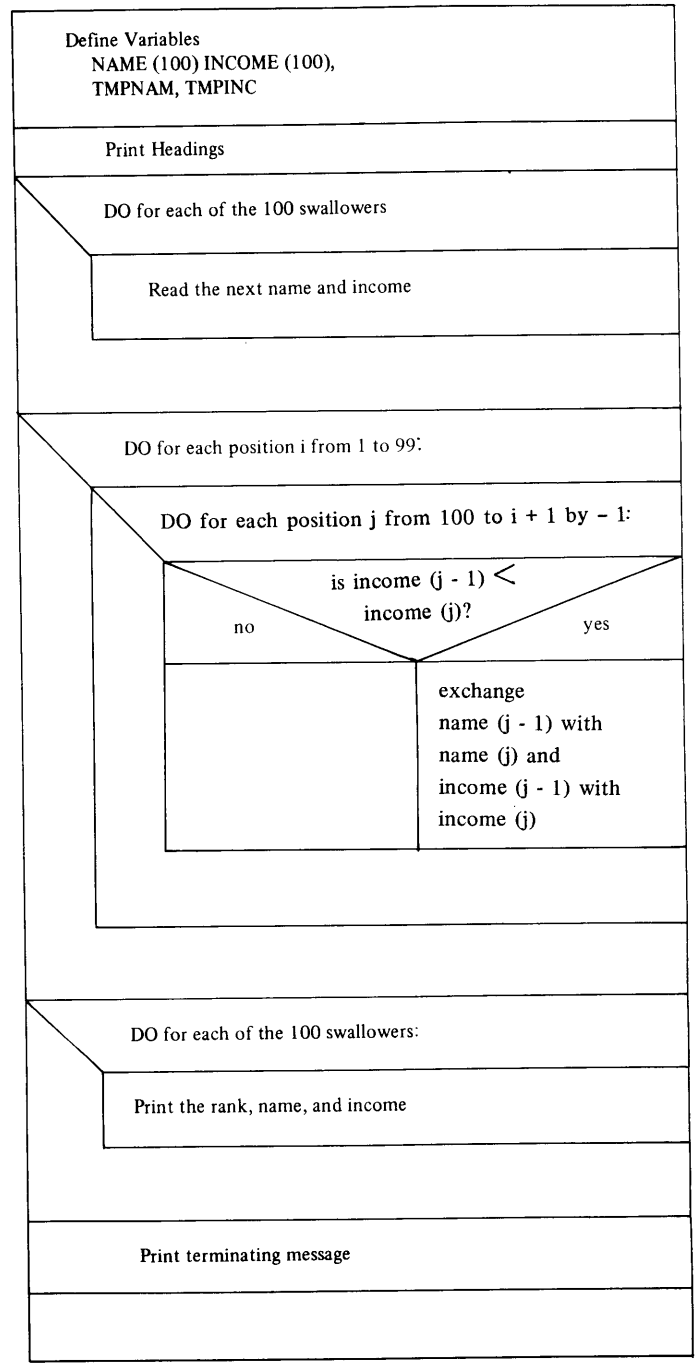
```
TMPNAM = NAME(J)
TMPINC = INCOME(J)
NAME(J) = NAME(J-1)
INCOME(J) = INCOME(J-1)
NAME(J-1) = TMPNAM
INCOME(J-1) = TMPINC
```

The list of declarations in component 1, then, is enlarged accordingly.

Remaining now are the details of implementation. The fact that there are two cycles suggests two nested loops:

```
"Define NAME(100), INCOME(100), TMPNAM, TMPINC."
"Print the headings."
do for each of the 100 swallowers:
    "Read and store the name and income."
enddo
do for every position i from 1 through 99:
    do for every position j from 100 through i + 1 by - 1:
        if
            income(i) is less than income(j)
        then
            "Exchange NAME(i) with NAME(j).
                INCOME(i) with INCOME(j)."
        else
        endif
    enddo
enddo
do for each of the 100 swallowers:
    "Print the rank, name, and income."
enddo
"Print terminating message."
"Stop."
```

**(a)**




**(b)**

**FIGURE 10.21** **(a)** Pseudocode Representation for Example 10.7. **(b)** N-S Diagram for Example 10.7.

1. An inner loop that finds the largest available value (from all those not yet positioned) and puts it in a particular position in the array.
2. An outer loop that activates the inner one 99 times—a round for each position but the last one.

The flowchart and pseudocode are shown in Figure 10.21. Note that component 1 shows the additional declarations, and component 3 is just a copy of the version in Figure 10.19. The program itself is in Figure 10.22.

**10.3 SUMMARY** Automatic control of cyclic processes can be set up by means of DO loops. The general structure of such a loop is as follows:

DO index = *sv*, *lv*, *incr*
    action
END DO

where index is the name of a DO-variable that automatically keeps track of the loop's progress, *sv* is the starting value to which index is set just before the first cycle, *lv* is the limiting value that helps define the loop's final cycle, and *incr* is the increment added to the DO-variable just before the next cycle is attempted. FORTRAN uses all of this control information to set up and monitor an *iteration counter*, computed as

MAX ( INT ( (*lv*–*sv*+*incr*) /incr ) , 0)

```
*                        EXAMPLE 10.7                                  *
*    THIS PROGRAM READS AND STORES 100 NAMES OF                        *
*    SWORD SWALLOWERS, ALONG WITH THEIR CORRESPONDING                  *
*    INCOMES. THE INCOMES ARE SORTED SO THAT THE FINAL                 *
*    OUTPUT CONSISTS OF THE LIST OF 100 NAMES, ALONG                   *
*    WITH THEIR RESPECTIVE INCOMES, SORTED IN DESCENDING               *
*    ORDER BY INCOME.                                                  *
*    NAME:      AN ARRAY FOR THE 100 NAMES                             *
*    INCOME:    AN ARRAY FOR 100 (CORRESPONDING) INCOMES               *
*    TMPNAM:    CHARACTER STRING TO HOLD A NAME DURING SWAP*
*    TMPINC:    REAL VARIABLE TO HOLD AN INCOME DURING SWAP*
*    INCNT:     DO-VARIABLE FOR AN INNER LOOP                          *
*    OUTCNT:    DO-VARIABLE FOR AN OUTER LOOP                          *
*    INDX:      ANOTHER DO-VARIABLE                                    *

      PROGRAM                EX1007
      IMPLICIT               NONE
      REAL*4                 INCOME(100),TMPINC
      INTEGER*2              INCNT,OUTCNT,INDX
      CHARACTER*25           NAME(100),TMPNAM
      LOGICAL                NOSWAP
      PRINT *, 'SWORD SWALLOWERS INCOME PROJECT'
      PRINT *, 'ANOTHER NOSTRIL SURVEY SERVICE'
      PRINT *, '     '
      PRINT *, 'RANK   NAME    INCOME IN BAVVOOZNIKS'
      PRINT *, '     '
```

**FIGURE 10.22** FORTRAN Statements for Example 10.7.

```
      DO INDX = 1,100
        READ *, NAME(INDX),INCOME(INDX)
      END DO

      DO OUTCNT = 1,99
        NOSWAP = .TRUE.
        DO INCNT = 100,OUTCOUNT+1,-1
          IF (INCOME(INCNT) .GT. INCOME(INCNT-1)) THEN
            TMPNAM = NAME(INCNT)          !SWAP!
            TMPINC = INCOME(INCNT)
            NAME(INCNT) = NAME(INCNT-1)
            INCOME(INCNT) = INCOME(INCNT-1)
            NAME(INCNT-1) = TMPNAM
            INCOME(INCNT-1) = TMPINC
            NOSWAP = .FALSE.
          END IF
        END DO
        IF (NOSWAP) GO TO 199
      END DO

  199 DO INDX = 1,100
        PRINT *, INDX,NAME(INDX),INCOME(INDX)
      END DO

      PRINT *, '    '
      PRINT *, 'NORMAL TERMINATION. ANOTHER NOSTRIL SUCCESS.'
      STOP
      END
```

**FIGURE 10.22** *(continued)*

This counter is decreased by one just after the completion of each repetition of the loop. As long as the counter is greater than zero, the next cycle is performed. Once it reaches zero, the loop is bypassed and the program continues beyond it.

The DO-variable is available for use during a loop. However, that usage must not change its *value*. Once the program leaves the loop, the DO-variable, still set at the last value it, had, becomes like any other variable.

**PROBLEMS**

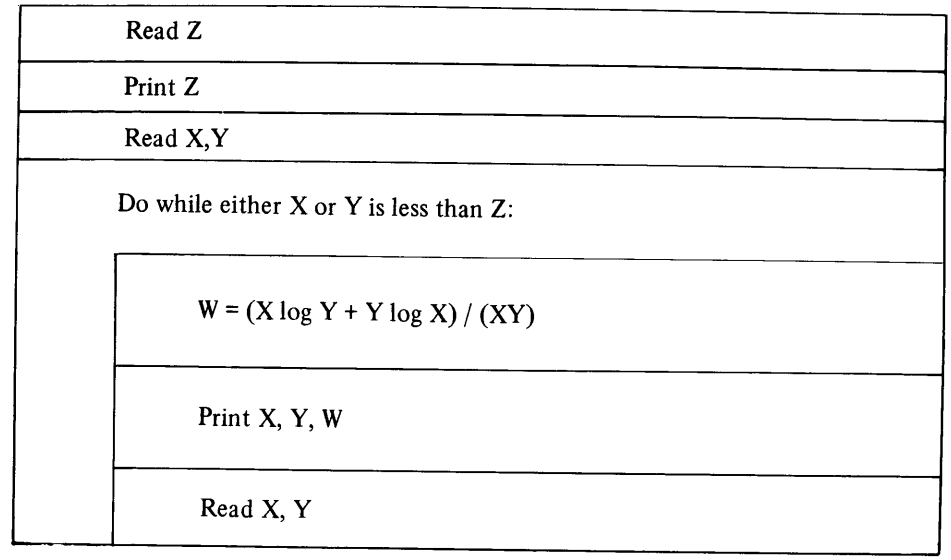
1. Write sequences of FORTRAN statements to handle each of the situations described below. Assume the following declarations:

```
REAL*4      X,Y,Z,R(10),S(10)
INTEGER*2   I,N
```

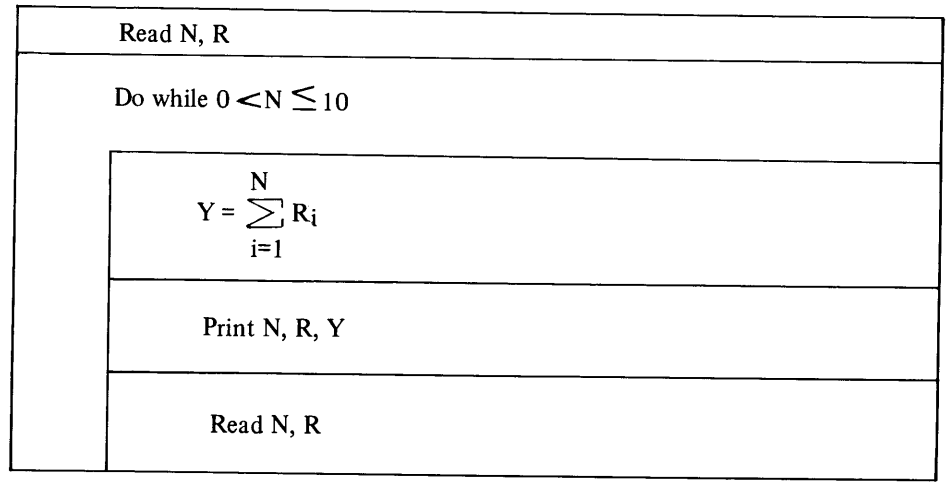
(a) "Read Y."
"Initialize Z and N to zero."
"Read the first value of X."
*while* Z is not greater than Y:
  "Add X to Z."
  "Add 1 to N."
  "Read the next X."
*endwhile*
"Print Y,Z,N."

(b) "Read X,Y."
*while* X is not equal to Y:
"Z = either X**Y or Y**X; the lower of the two values always is the exponent."
"Print X,Y,Z."
"Read the next X,Y."
*endwhile*
"Print 'end of run' message."

(c)




(d)




(e) "Read Y."
"Print Y."
"Initialize I to 1."
*do*
"X = Y or R(I), whichever is larger."
"Print R(I),X."
"Add 1 to I."
*until* I > 10 or R(I)=Y

2. Assume the following declarations:

```
REAL*4        S, X, A(8)
INTEGER*2     I, L, M, N
```

For each of the sequences of statements listed below, indicate

(1) The iteration count for each loop (if more than one loop is specified;
(2) The values printed by the output statement(s):

(a)
```
S = 0.0
X = 3.0
DO I=1,4
  S = S+I*X/(I+8)
END DO
PRINT *, S
```

(b)
```
    N = 0
    L = 2
    DO 10 I=3,8
      IF (MOD(I,2) .NE. 0) N=N+L*(2*I-1)-6/I
10  CONTINUE
    PRINT *, N
```

(c)
```
S = 1.0
X = 2.0
DO I=1,10,4
  S = S*X**I
END DO
PRINT *, S
```

(d)
```
DO I=1,8
  A(I) = (I+4)/I
END DO
L = 3
N = 6
S = 0.0
DO I=L,N
  IF (A(I) .GE. 1.8) S=S+A(I)
END DO
PRINT *, A, S
```

(e)
```
X = 4.0
DO I=1,4
  N = 9-I
  A(I) = X**I-2*I
  A(J) = 0.5*A(I)
END DO
S = 0.0
DO I = -6, -2
  L = ABS(I)+1
  S = S+A(L-I)
END DO
PRINT *, A, S, L
```

(f) NOTE: There is no assurance that this loop will be executed exactly 11 times; the effects of roundoff error make such assurance impossible.

```
X = 1000.0
DO Y=0.0, 1.0, 0.1
  IF(Y .LE. 5.00)  THEN
    X = X-1.0**Y
  ELSE
    X = X+10.0**Y
  END IF
END DO
PRINT *, X
```

3. Assume the following declarations:

```
REAL*4          X
INTEGER*2       I, J, NUM(3,4), R, C
CHARACTER       CH*12, T1*7, T2*7
DATA            NUM, T1, T2/3*4, 2, 0, 4*3, 1, 6, 0, 'AEIOUGH', '*****$$'/
```

Show the total number of cycles through each loop and the values printed by the output statements for each of the following sequences:

(a)
```
DO I=1,3
  DO J=1,4
    NUM(I,J) = I+J+NUM(I,J)
  END DO
END DO
PRINT *, NUM
```

(b)
```
    DO 10 I=1,4
      DO 8 J=1,3
        NUM(J,I) = NUM(J,I)/(I*J)
 8    CONTINUE
10  CONTINUE
    DO 6 I=1,3
      PRINT *, NUM(I,1), NUM(I,2),
      NUM(I,3), NUM(I,4)
 6  CONTINUE
```

(c)
```
      X = 0.0
      R = 3
      C = 3
      DO I=R, 1, -1
        DO 12 J=C, 1, -2
          IF (NUM(I,J) .LT. I+J)   THEN
            X=X+NUM(I,J)
          ELSE
            X=X-NUM(I,J)
          END IF
   12   CONTINUE
      END DO
      PRINT *, X
```

(d)
```
      CH = 'THOROUGHFARE'
      DO I=1, 12
        DO J=1, 7
          IF (CH(I:I) .EQ. T1(J:J))   T1(J:J)=T2(J:J)
        END DO
      END DO
      PRINT *, CH
```

4. One of the most frequently used techniques in data analysis is called *smoothing*. It is applied often to *time series*, that is, a set of readings or measurements or tallies recorded over a period of time at equally spaced time intervals. For example, in medical data processing, these might be blood pressure readings taken every six hours; in chemical analysis these might be pH readings taken every five minutes; in stock market analysis these might be daily stock averages. The point is that these readings might follow a trend which is partially hidden by the little fluctuations from one reading to the next. In order to suppress the effects of these local fluctuations and bring out the larger, more interesting trends with greater clarity, each is smoothed by averaging it with the next few readings and reporting that average in place of the individual reading. The number of readings used in the averaging process varies with the particular situation. In this problem, each set of readings will consist of twenty real values, and the smoothing process will be performed on groups of three readings. Thus, the first reading will be replaced by the average of the first, second, and third readings; the second reading will be replaced by the average of the second, third, and fourth readings, and so on. Readings 19 and 20 will be reported as is, since the smoothing process cannot be applied to them.

   Write a program that applies this smoothing process to any number of input sets. For each set, the program prints a line for each reading that shows the original value and the smoothed value. Output sets are to be separated by a blank line and the last set is to be followed by a terminating message.

5. Here is a more general version of the program described in the previous problem: Instead of expecting the same number of readings for each input set, the program is to accept and process input sets with any number of readings in each set. (It is up to you to determine how to separate the sets; however, the most obvious method cannot be used here: That is, you cannot include as part of each input set an explicit integer variable that specifies how many readings there are.)

6. Another generalization that we can add to the program in Problem 5 is one in which the user may specify the number of readings to be used for the smoothing process. Write a program in accordance with the specifications given in Problem 5 with the additional feature that it reads (as part of the input for each set of data to be smoothed) a value NMSMTH specifying the number of readings to be used for each group to be averaged. Your program should be designed to handle the possibility that NMSMTH is too large for the number of readings available for that run. An obvious case, for example, would be one in which NMSMTH is specified as 18, and there are only 16 readings. However, we shall be rather strict about the relation between NMSMTH and the number of readings: the program will allow the smoothing process to proceed to its conclusion only if at least three readings can be replaced by smoothed values. For practical purposes, we shall impose the additional restriction that NMSMTH can never be larger than 20.

7. Generalize the program in Problem 6 further by including a default value of 3 for NMSMTH. Then, if the user does not wish to specify his/her own value for NMSMTH, the program automatically will use 3. On the other hand, if the user wants a different value for NMSMTH, such a value may be specified and the program will use it. Now, that new value is to remain in force for one or more sets of readings as long as the user does not change it. When a change is wanted, the user should be able to specify either a new value for NMSMTH or a return to the program's default value.

8. The Tramplevanian Knowledge Authority needs to know the distances between each pair of principal cities in that far-off land. There are ten such cities. As part of an earlier project, somebody had prepared a set of X-Y coordinates on which each of the cities is represented as a point whose X and Y distances from

the origin is as Kloompldorf, the capital city. The coordinates for the cities, then, are defined as follows:

| *City* | *X* | *Y* |
|---|---|---|
| KLOOMPLDORF | 00.0 | 00.0 |
| MYUMBESBURG | 12.7 | −16.8 |
| GROSSENDORF | −6.5 | −22.8 |
| WEISENVILLE | 28.9 | 8.4 |
| SCHWARZDORF | −17.7 | 34.2 |
| MLEPPLSBURG | 61.0 | 0.9 |
| SCHLEPPVILLE | 81.3 | −6.6 |
| PINKENPLATZ | 2.3 | 89.4 |
| STROMENBACH | 77.0 | 91.1 |
| HALBESSTADT | −92.3 | 91.1 |

Recalling that Problem 10 in Chapter 6 gave a formula for computing the distance between two points expressed in terms of their respective X-Y coordinates, write a program that provides the Knowledge Authority with the information it wants. The distance between two given cities should appear on a separate output line that gives the names of the two cities and the distance between them, rounded to the nearest 0.1 kilometer.

9. This is a more convenient version of the program in Problem 8. (More convenient, that is, for the Knowledge Authority.) Using the same data and the same requirements, design and implement the program so that it prints the output arranged in ascending order by distance (i.e., the shortest distance first). If two distances should come out to be the same, it does not matter which is printed first, as long as their respective output lines are next to each other.

10. Trouble in Tramplevania: The head of the Knowledge Authority, just back from the Official Three-Hour Intensive Computer Workshop for Executives And Their Guests, has announced that Everything is Easy with Computers. Accordingly, she wants the program in Problem 8 written so that the output appears as follows: The cities are to be arranged in alphabetical order. That is, the first nine lines will show the first city (GROSSENDORF), along with the distances between it and each of the other nine cities. The other city on the first line should be the one closest to GROSSENDORF, the other city on the second line should be the one next closest to GROSSENDORF, and so on. Then, the next city (HALBESSTADT) will require only eight lines. (Because of the peculiar terrain in Tramplevania, the distance between GROSSENDORF and HALBESSTADT is the same as the distance between HALBESSTADT and GROSSENDORF, and so it goes between any two cities there.) This pattern continues, of course, so that the third city (KLOOMPLDORF) will appear in the first column of the next seven output lines, and so on.

11. Caleb Grutchkrinkler, working out of a small stone house in Perkins Warp, Vermont, has built quite a reputation as a master restorer of New England schmichiks. Almost any reasonable craftsman can restore a Mid-Atlantic, Southern, or Midwestern schmichik (these wonderful artifacts are unknown beyond Norton, Kansas), but it takes somebody special to do justice to the intricate New England schmichik, and Caleb G. is tops. Anyway, a New England schmichik consists of 15 parts. Thus, when Caleb G. gets hold of one, he inspects it immediately to establish its exact condition. If it is beyond repair and nothing can be salvaged, he just throws it away. However, that rarely happens. If anything at all can be saved, Caleb G. assigns a six-digit identification number (SCHMNO) and prepares a detailed description of it. His computing service records this information on a line that contains SCHMNO, along with SCHCST, the amount (to the nearest cent) that Caleb paid for the schmichik, and an integer value for each of the 15 parts. (Caleb never buys a schmichik for more than $100.00):

    O: The part is missing or no good; it must be replaced.
    1: The part can be rebuilt and is worth saving.
    2: The part is intact, needing only adjustment.
    3: The part is fine; no work required here.

    Up to now, Caleb G. has been content with a simple printout of these data. Up to now. But rapidly growing business has made this inadequate. Consequently, a program is needed that reads and processes any number of these lines as follows: For each line read, the program prints a line of output showing the schmichik identification and the number of parts in each of the four categories. The following form might be one way of showing this:

| *SCHMICHIK NO.* | *COND. 0* | *COND. 1* | *COND. 2* | *COND. 3* |
|---|---|---|---|---|
| 000323 | 4 | 3 | 2 | 6 |
| 006755 | 0 | 8 | 3 | 4 |

After all the data have read, the program is to leave a blank line and print the following summary information:

(a) The number of schmichiks processed;
(b) A line for each of the 15 parts (calling them PART 1, PART 2, etc.) showing how many of those parts are in condition 0, how many are in condition 1, and so on.

12. The success of the program in Problem 11 has sparked Caleb G.'s interest, and he wants to make it more useful to him and his New England schmichiks. Specifically, he wants to include some systematic way of computing how much it will cost him to restore a particular schmichik. Since he already knows what he paid for the schmichik in the first place, he can put these two pieces of information together and use them as a basis for establishing a sale price. After examining his records, Caleb G. has come up with a set of guidelines based on the estimated hours required for restoration. For this purpose, an overall condition score (OCS) is computed for each schmichik simply by adding together the conditions for the 15 individual parts. Thus, a schmichik in perfect shape would have an overall condition score of 45 (i.e., `15*3`), and a top-to-bottom dead loss of a schmichik (something Caleb would not even take for free) would have an overall condition score of 0. This overall condition score is related to the number of hours required for restoration according to the schedule shown below. The restoration time is affected also by certain details about the schmichik's condition, and this information is shown as well:
    (1) If a schmichik's OCS is less than 20, it is not worth restoring. The work involved in saving the usable parts and throwing away the rest amounts to 4.5 hours.
    (2) When the OCS is 20 or more, but less than 25, restoration time amounts to 11.5 hours.
    (3) Schmichiks with OCS values of 25 through 31 have restoration times of 9.0 hours.
    (4) When OCS is 32 through 38, restoration time drops to 4.5 hours.
    (5) When OCS is 39 through 42, restoration time is 3.25 hours.
    (6) If a schmichik is in good enough shape to earn an OCS above 42, there is little or no restoration (mostly cleanup). This time is 1.5 hours.
    (7) If a schmichik has at least four parts with condition 0 (regardless of OCS), add 0.75 hours to the restoration time.
    (8) If a schmichik has at least six parts with condition 1 (regardless of OCS), add 1.50 hours to the restoration time.
    (9) Regardless of anything else, if part number 6's condition is 0, add 0.75 hour to the restoration time.
    (10) Regardless of anything else, if part number 6's condition is 1, add 0.25 hour to the restoration time.
    (11) Regardless of anything else, if part number 6's condition is 3, deduct 0.5 hour from the restoration time.

    Caleb believes these figures take into account the fact that some of his schmichiks (for which he has paid) are not sold as complete items but serve instead as sources of parts for restoring other schmichiks. Consequently, Caleb's total cost is computed just by multiplying the restoration hours by Caleb's hourly labor rate and adding that result to the original cost. Since the hourly rate changes with the cost of living, it is read in at the beginning of each run instead of being set up as a parameter in the program.

    Based on these considerations, write Caleb's program for him. As in the previous problem, the program is to process any number of lines. The contents of each line is the same as described in Problem 11. A separate line, read prior to the first schmichik's line, defines the hourly rate of that run. Then, for each line processed, this program is to print a line showing the schmichik number, the original cost, the Overall Condition Score, and the total restoration time. If a particular schmichik is not worth restoring, an appropriate message is to appear on that schnmichik's line in place of the restoration time. After the last schmichik has been processed, the program is to leave a blank line and print each of the following summary results on a separate line:
    (a) The number of schmichiks processed
    (b) The number of schmichiks rejected for restoration
    (c) The total cost for all of Caleb's purchases
    (d) The total number of restoration hours
    (e) The total restoration cost
    (f) The total cost to Caleb, i.e., (c)+(e)

13. Write and run a program that computes and prints the number of ways to produce change for a dollar using pennies, nickels, dimes, quarters, half-dollars, and the metallic (Anthony) dollar. The program does

not need to read any input. Each line of output is to show a legitimate combination using the following general layout:

| *PENNIES* | *NICKELS* | *DIMES* | *QUARTERS* | *HALVES* | |
|---|---|---|---|---|---|
| 100 | 0 | 0 | 0 | 0 | |
| 95 | 1 | 0 | 0 | 0 | etcetera |

Since there is only one legitimate combination involving the metallic dollar (all by itself, with no other coins), that possibility is printed as a separate line after the table illustrated above. Then, there is to be a final line of output showing NWAYS, the number of combinations. (NOTE: this is an instance in which brute force will not pay off. Chances are that if you try looking at all possible combinations, you will run out of computer time. Consequently, it is a good idea to spend a little time on the design of a method that will avoid (as much as possible) looking at combinations that the program "knows" will not work out.)

14. The dollar is not the only currency for making change. In far-off Mammaligga, the unit of currency is the Riegloch. There are three Glepchiks to a Riegloch, seven Truppooks to the Glepchik, two Znehs to each Truppook, and three Pitinipehs to the Zneh. So. Not counting the Riegloch all by itself as one of the ways, repeat Problem 13, finding out how many ways there are to make change for a Riegloch. (In the Mammaliggan language, all units of currency are written and spoken with initial capital letters.)

15. A crack team of archaeologists from seven countries has made an astonishing discovery: Digging near the ancient Roman city of Tutti Ruino del Volcanico, the group came upon a large, sealed leaden box in reasonably good shape. Imagine their surprise when, upon opening the box with great care, they found a large collection of crudely punched cards. Oh yes they did. After some calm was restored, the team made photographic copies and began a painstaking examination. Sir Smedley Guavahead, second in command, finally figured it out. The cards definitely contained numerical data, but they were punched as Roman numerals, don't you see. Once this great intellectual leap had been made, it became clear that each card contained four separate items. A typical card looked like this, except a lot fuzzier:

```
'MCCCXXIII     'LXIV'     'DCCLXXIX'     'CCXCVIII'
```

Thus, before the scientists could settle down to the enormous task of trying to determine what these numbers meant, they have to know what these numbers are. For this purpose, a program is needed to read a sequence of these data (carefully rerecorded from the photographs) and convert and print each of the values as an integer. Examination of the entire batch, and manual conversion of a few sample values, has established that the highest value in the collection comes out to be 3888. Each card's data is to produce two lines of output, separated by a blank line. The first output line is to be an echo of the input data for one line, and the second shows the converted values. For example:

```
ROMAN:    'MCCCXXIII'   'LXIV' 'DCCLXXIX'  'CCXCVIII'
ARABIC:   1328          64     779          298
```

To help in this effort, Sir Smedley himself has prepared the following helpful table:

| *ROMAN* | *ARABIC* | *ROMAN* | *ARABIC* |
|---|---|---|---|
| I | 1 | II | 2 |
| III | 3 | IV | 4 |
| V | 5 | VI | 6 |
| IX | 9 | X | 10 |
| XI | 11 | XIV | 14 |
| XIX | 19 | XX | 20 |
| XL | 40 | XLV | 45 |
| L | 50 | XC | 90 |
| XCIX | 99 | C | 100 |
| CD | 400 | D | 500 |
| CM | 900 | M | 1000 |

Here are some sample cards:

```
'MCCCXXIII'     'LXIV'     'DCCLXXIX'     'CCXCVIII'
'XVIII'     'MDCCCXCIX'     'MCDXCII'     'XXXIII'
'MMMDCCCLXXXVI'     'XXXVIII'     'V'     'LXXXVIII'
```

16. Rewrite the program in Example 10.5 using the DO WHILE statement. The information produced is to be the same as in the example.

# 11
# *Introduction to Subprograms*

Chapter 6 introduced the idea of a *subprogram*—an independent group of statements that can be incorporated into a larger program and used there again and again. The subprograms in FORTRAN's library enabled us to pretend that relatively complicated tasks such as the computation of a square root or a logarithm were single operations and could be treated as such. Now, these concepts will be developed further by looking at the construction and use of subprograms that we design ourselves.

## 11.1 BASIC PROPERTIES OF SUBPROGRAMS

The name "subprogram" emphasizes the fact that a subprogram, though conceptually and procedurally independent, cannot run by itself. It is specifically intended for use by another program. That is, it operates at the request of another program; when it has concluded the operations it is designed to perform, the program that made the request picks up where it left off. The structural connections between the two programs are such that this process will occur each time such use is requested, regardless of the number of requests there are or the number of different places from which the requests are made.

### 11.1.1 How Subprograms Work

The major reason for writing and using subprograms stems from the fact that a subprogram can be viewed as a prepackaged unit. It is a process with its own identity and its own existence. Its mechanics are such that it can be designed, developed, and perfected on its own, without our even needing to know which programs actually will be using it. Then, once a subprogram has been completed, it can be made available to any program that has a use for it. Thus, it is possible to produce large, complex programs by combining a selection of pretested building blocks (i.e., subprograms). This organizational concept has exerted the most profound influence on the way programs are designed and developed. Many large successful programs are produced by several people, each working on one or more subprograms. Accordingly, the next section will look more closely at the structural relations between subprograms and the programs that use them.

**11.1.1.1 Subprograms and the Main Program** Organizationally, every program consists of a *main program* which may or may not use subprograms. Thus, every program we have written thus far is a main program. There is no rule that requires the use of subprograms, and there is no limit to the number of subprograms that can be used by a main program.

When a subprogram is going to be used (*invoked*) by a main program, a physical copy of the subprogram is included as part of the overall program. If the subprogram is one of those in FORTRAN's library (a list is given in Chapter 6), the process of providing a copy is handled automatically as part of the system's duties. Otherwise, it is up to the

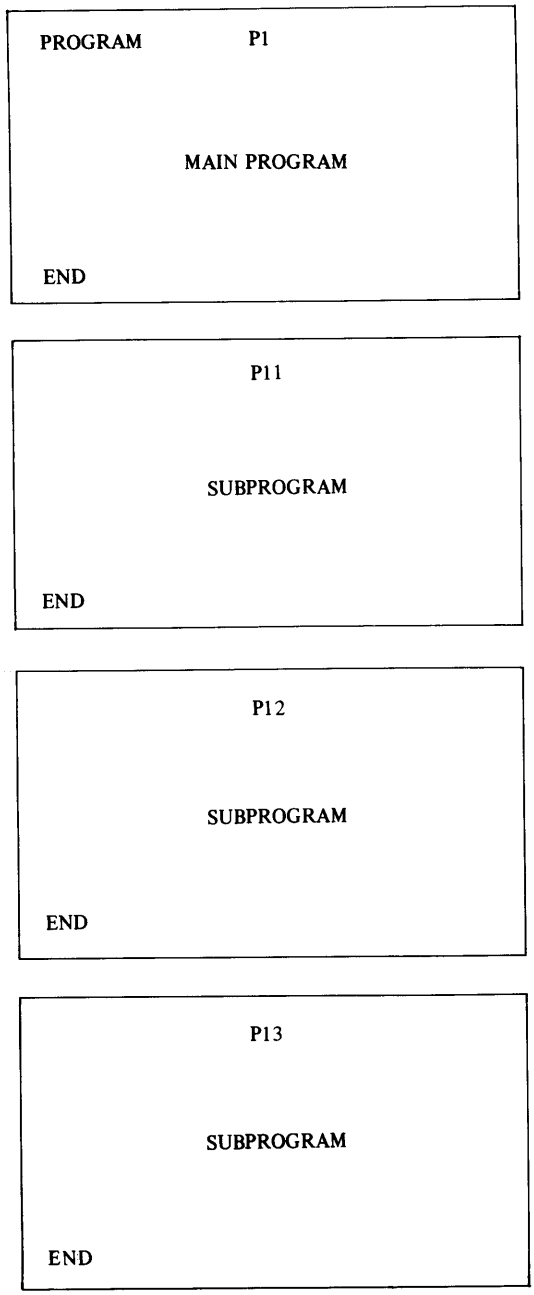



**FIGURE 11.1** General Program Structure.

programmer to include a physical copy as part of the sequence of statements submitted to the FORTRAN compiler. In either case, only a single copy of each subprogram is required for a given program regardless of the number of times the subprogram is used.

**11.1.1.2 Synthesis of Programs** Figure 11.1 shows a structure consisting of a main program P1 and three subprograms P11, P12, and P13. Note that the main program is physically the first program in the structure. (This is not compulsory in FORTRAN, but it is a practice we shall follow.) Regardless of the number of subprograms, each one follows the END statement of the previous one, and the first subprogram follows the END statement of the main program.

```
   PROGRAM     P1        (MAIN PROGRAM)

 1             statement
 2             statement
 3             INVOKE P12
 4             statement
 5             INVOKE P11
 6             INVOKE P12
 7             statement
 8             USENVOKE P13
 9             statement
10             INVOKE P11
11             STOP
               END
```

```
     P11       (SUBPROGRAM)
111            statement
112            statement
                 . . .
114            INVOKE P13
115            statement
                 . . .
117            RETURN
               END
```

```
     P12       (SUBPROGRAM)
 21            statement
                 . . .
                 . . .
 25            statement
                 . . .
 28            RETURN
               END
```

```
     P13       (SUBPROGRAM)
                 . . .
 41            statement
                 . . .
 55            RETURN
               END
```

**FIGURE 11.2** Program Structure and Execution Sequence.

**11.1.1.3 Sequence of Execution** The control exercised by a main program over a subprogram is seen by the fact that the point at which a subprogram runs has nothing to do with its physical placement. Regardless of where the subprogram actually appears, it will run only when the controlling program says it does, i.e., when the subprogram is *invoked*. Each request (*invocation*) is an explicit one. When such a request is made, the program goes directly to the subprogram's first statement and executes from that point. At the completion of the subprogram, execution automatically goes back to the invoking program and continues from the point immediately after the request.

This concept is demonstrated by the program structure in Figure 11.2. To avoid unnecessary confusion, we shall make up a type of statement, i.e.,

INVOKE *subprogram*

to indicate a general invocation. This will give us an opportunity to follow the sequence of events without having to deal with the details. Here is what happens in this program:

1. The main program (P1) starts; statement 1 is executed, followed by statement 2.
2. Statement 3 invokes subprogram P12, and all of its statements (21 through 28) execute. The flow of events then *returns* to the main program at statement 4. (The RETURN statement accomplishes this in FORTRAN.)
3. Statement 4 is executed.
4. Statement 5 invokes subprogram P11, and it begins execution from statement 111, continuing in sequence till statement 114.
5. Statement 114 invokes subprogram P13, thereby transferring attention to the first statement in P13.
6. The statements in P13 (i.e., 41 through 55) execute in sequence, and the program returns to the invoking program or subprogram, in this case P11.
7. Execution in P11 continues in sequence from statement 115 and proceeds through statement 117, at which point P11 returns to the program that invoked *it*, namely P1. The return is to statement 6.
8. Statement 6 (the next one to execute) invokes subprogram P12. (This is the second invocation of this subprogram; the first time, it was invoked by statement 3.) Accordingly, statements 21 through 28 execute again, after which the program returns to statement 7.
9. Statement 7 is executed.
10. Statement 8 invokes subprogram P13, so that statements 41 through 55 are executed next, in sequence. Processing then continues with statement 9 in the main program.
11. Statement 9 executes.
12. Statement 10 invokes subprogram P11, thereby causing execution in that subprogram starting with statement 111 and proceeding through statement 114.
13. Statement 114 invokes subprogram P13, thereby causing statements 41 through 55 to execute, after which subprogram P11 resumes. This subprogram completes its work with statement 117 and returns to the main program at statement 11.
14. Statement 11 executes, and the program ends.

### 11.1.2 Subprograms and Program Development

Now that the basic flow sequences have been established, we can discuss the subprogram as a powerful aid in the program development process. Earlier, mention was made (Section 11.1.1) of the subprogram's usefulness as a building block that can be designed, developed, and tested separately, away from any programs in which it might be used. Then, when it is ready, it can be "plugged into" any program that needs it. Entire companies of professional program designers have built operations around this idea, selling a wide variety of useful subprograms to customers they will never meet. These

customers design *their* programs to "expect" the subprograms, so that their inclusion is simple and painless. This is not different from the builder who constructs a house with the expectation that a furnace will be installed in a certain place, in a certain way. The furnace may not be there yet, but the house is ready for it. Then, when the furnace shows up, it is a unit, preassembled and pretested. If the design was done properly, its installation is simple and painless.

When applied to the program development process, this idea leads to two powerful techniques. The first of these, already mentioned, centers on the systematic, independent development of subprograms for eventual use as building blocks in other programs. A second technique relates to the systematic preparation of main programs that are ready to accept subprograms even though those subprograms may not be available yet. These techniques are introduced and outlined in the next two sections.

**11.1.2.1 Separate Development of Subprograms** In the introduction to Section 11.1 it was established that a subprogram, by its nature, cannot operate alone. It is intended for use as a component in some larger program structure. Consequently, any process for the independent development of a subprogram must involve some kind of main program.

One way to approach this development, then, is to turn the subprogram into a main program until we are convinced that it works. Once its proper operation is assured, we can reconvert it into its final form, i.e., a subprogram. This is undesirable for a variety of reasons, not the least of which is that we are not really developing the component we set out to develop; the processes involved in converting the working program into an "equivalent" subprogram introduce too many possible sources of errors.

Consequently, a much better technique for independent subprogram development starts with the subprogram in its eventual form and equips it with a main program. This main program is *not* the same as the one with which the subprogram eventually will be used. (After all, we are not expected to know what *that* main program (or those main programs) will look like.) Instead, this special main program (called a *driver*) has three primary purposes:

1. To provide a vehicle for running the subprogram.
2. To invoke the subprogram in the same way it will be invoked in "actual" use. This provides a set of realistic conditions on which the subprogram can operate.
3. To provide a place to which the subprogram can return.

By doing this from the outset, we provide the subprogram with a realistic environment in which it can be developed. At the same time, we give ourselves the opportunity to examine the subprogram's behavior.

It is important to understand that a driver need not be complicated. In fact, there is every reason to make the driver as small and simple as possible. Specifically, the driver should not do any more than the minimum it has to do to satisfy the three requirements defined above. When we do that, we can eliminate the possibility of introducing mistakes in the driver. (The whole idea is to be able to concentrate on the subprogram.) Furthermore, a simple driver is less painful to throw away when we are finished with it, and that is exactly what should be done with a driver that has fulfilled its purpose. Turning to the newly built house once more as an example, it is standard practice to help the development process by erecting scaffolding or other supporting structures. Once they have served their purpose, they are taken away, with no tears shed. This situation, basically, is no different.

**11.1.2.2 Expectant Main Programs** This second technique is based on the idea that, once a program's overall structure has been defined (i.e., we know what its major

components are and what each one does), we can duplicate that structure. This is true even though the subprograms may not be developed yet. Note that we are not duplicating the subprograms' details; we cannot. Instead, we are providing the same structure by including a *representative* for each subprogram that the program will use. That representative may be nothing more than a skeleton, just enough to provide a physical presence. This usually is known as a *stub*. In general, its single purpose is to be there, so that there is something to invoke in the same way that the final subprogram will be invoked. For many situations, this purpose is fulfilled quite adequately by a subprogram consisting of an initial statement, some declarations, a RETURN statement, and an END statement. The point is that a stub does not have to do anything. Its being there often is enough to establish the overall flow of the final program. Then, when the corresponding subprogram (developed independently) is ready, it can replace the stub without changing anything else.

A little later in this chapter, after the details of subprogram construction have been introduced, we shall put these techniques to work.

### 11.1.3 Types of FORTRAN Subprograms

FORTRAN recognizes three types of subprograms: *statement functions*, *functions*, and *subroutines*. Each type has its specific uses and advantages. These will become clear once the basic techniques are defined for the construction and invocation of each type.

**11.1.3.1 The Function Subprogram** A *function* is a subprogram that returns one value, i.e., it produces a single result each time it is invoked. It may be as long and complex as it needs to be. The number of data items that it can receive from the program that invokes it (i.e., the number of *arguments*) depends only on what the function is designed to do and not on any restrictions in the language. Moreover, it can include decision mechanisms that influence the way it prepares its result. However, when its processing is finished, the outcome still is a single value.

Figure 11.3 shows the general framework for a function subprogram. Since the function is separate from (i.e., not contained in) the main program, it is considered to be an *external subprogram*. The subprogram itself begins with a FORTRAN statement that *defines* the name of the function. Attached to the function name is a parenthesized list of

```
PROGRAM     P1
statement
    . . .
    . . .
variable = fname (arg1,arg2,...,argn)
statement
    . . .
END

FUNCTION fname (dmarg1,dmarg2,...,dmargn)
statement
    . . . .
    . . . .
statement
fname = expression
RETURN
END
```

**FIGURE 11.3** Structural Relations Between a Main Program and a Function.

names called *dummy arguments*. This term emphasizes the fact that these names do not represent actual values on which the function is to operate. Instead, the function name and the list of dummy arguments provide a model of how the function is to be invoked.

1. The function name (`fname` in Figure 11.3) gives the function an identity that distinguishes it from all other functions used by the program. It is this name that is specified when the function is invoked. This is seen in the example usage by the main program (`P1`) of Figure 11.3.

2. The list of dummy arguments indicates the number of *actual* arguments to be supplied when the program is invoked, as well as the type of each argument. In addition, the list guides FORTRAN in determining the sequence in which the actual arguments are to be used when applied to the operation within the function.

3. The statements in the subprogram itself describe the processing that the function will perform *when it is invoked.* Remember that the activities described by the function do not take place until some other program requests them.

4. When the final result is computed, it is assigned to a variable whose name is the function name. This variable (`fname` in Figure 11.3) is created automatically as part of FORTRAN's processing of the `FUNCTION` statement.

As seen in Figure 11.3, a function subprogram is invoked by using its name, in an expression, along with a list of actual arguments. The value developed by the function is returned to the expression for further use. General rules for invocation are as follows:

1. A function may be invoked by a main program or by an external subprogram.
2. A function may not invoke the main program.
3. A function may not invoke another subprogram which, in turn, invokes that function.
4. The actual arguments supplied to a function should not be changed by the action of that function. They should be used to prepare the single result delivered in the variable carrying the function's name.

Other features will become apparent when we look at functions in more detail.

**11.1.3.2 Statement Functions** This type of subprogram is considered to be a function because it behaves like a function: it delivers a single result regardless of the number of arguments supplied to it, and it is invoked just like a function (as described in the previous section). However, three basic restrictions make this a special type of function:

1. The description of the function is limited to a single statement.
2. The statement describing the function must be embedded in a main program or subprogram, and only that program or subprogram is able to invoke that function directly.
3. The description of the function must appear in the program before any of that program's executable statements and after its declarations.

Because of the second restriction, the statement function is termed an *internal subprogram*. Figure 11.4 shows a main program in which a statement function is defined and then invoked.

**11.1.3.3 The Subroutine Subprogram** This is the most general type of subprogram, in that there are no particular restrictions with regard to the number of arguments it will accept or the number of results it can return. Like the function, it is an external subprogram whose length and/or complexity is limited only by the programmer's requirements.

```
PROGRAM     MAINPG
statement
    ...
stfunc (dmarg1,dmarg2,.....,dmargn)
    ...
statement
statement
    ...
variable = stfunc (arg1,arg2,.....,argn)
statement
    ...
END
```

**FIGURE 11.4** Structural Relation Between a Main Program and a Statement Function.

```
PROGRAM     MAINPR
statement
statement
    ....
    ....
CALL subnam (arg1,arg2,arg3,.....,argn)
    ....
statement
END

SUBROUTINE
        subnam (dmarg1,dmarg2,dmarg3,.....,dmargn)
statement
statement
statement
    ....
    ....
dmarg2 = expression
dmarg3 = expression
RETURN
END
```

**FIGURE 11.5** Structural Relation Between a Main Program and a Subroutine.

Figure 11.5 shows the basic structural relationships between a subroutine subprogram (which we shall call a subroutine from now on) and the main program with which it is associated. Each subroutine begins with a SUBROUTINE statement that gives the name of the subroutine as well as a list of dummy arguments. This opening statement serves exactly the same purpose as the FUNCTION statement does for the function subprogram. However, one important difference is that the subroutine name is *not* used by FORTRAN as a variable in which to store a result computed by the subprogram. To do such a thing would make no sense: a subroutine may produce any number of results, and there is no consistent reason to select any particular one. Accordingly, the results, when computed, are stored in arguments provided for that purpose. This storage of computed results is illustrated, in general terms, in Figure 11.5, where the results of some computations are assigned to two of the arguments (arg2 and arg3). Then, when the subroutine concludes by returning to the point of invocation, the values in those arguments are available for further use.

Unlike a function, a subroutine is invoked by a separate statement:

CALL *subnam* (argument list)

This statement transfers control to the subroutine whose name matches that specified after the word CALL. The return (after completion) is to the statement in the invoking program or subprogram immediately after the CALL. Limitations on who may invoke whom are similar to those listed for functions. Subroutines and functions both may be associated, in any mixture, with a given main program. Moreover, subroutines may invoke functions and vice versa.

## 11.2 CONSTRUCTION OF SUBPROGRAMS

Using the general concepts discussed earlier, this section develops the detailed techniques for building subprograms.

### 11.2.1 Definition of Functions

One of the basic concerns in designing a function is to specify the type of data handled by that function. This must be done so that the values (arguments) supplied during invocation can be checked for consistency against the types expected by the function. For example, it is helpful to be able to detect an attempt to use a character string as an argument in a function designed to compute its logarithm. Such definitions are established by means of the function name and its list of dummy arguments.

**11.2.1.1 The Function Name** The data type of the value *returned* by a function is determined by the function name. That name is formed according to the same rules (Section 4.2.1) that govern variable names. One of these rules (Section 4.2.2.6) allows an automatic mechanism (a *default*) to associate a name with the REAL or INTEGER data type depending on the name's first letter. Consequently, if we allow the default to operate, the functions whose names begin with the letters A through H or O through Z (for example, PCOMP) return a REAL result, and those whose names begin with I through N (for example, NFLAG) will return an INTEGER result. However, we shall declare function names explicitly, as has been our practice with other names. Moreover, the IMPLICIT NONE declaration will provide an additional reminder to do so.

For external functions, this definition is given by specifying the data type as part of the FUNCTION statement:

data type FUNCTION name (dummy argument list)

For instance, the statement

```
INTEGER*2 FUNCTION SETFLG (dmarg1,dmarg2,....,dmargn)
```

announces the fact that the function named SETFLG will return a sixteen-bit integer value. Similarly, the statement

```
CHARACTER*20 FUNCTION WORD_BUILD (dmarg1,dmarg2,....,dmargn)
```

proclaims that WORD_BUILD returns a 20-character string as its result.

Since a statement function looks like an assignment statement, there is no place to specify a data type for the returned value. Instead, the type is defined along with the other declarations in the program. For instance, the sequence

```
REAL          CORRECT_WT, OLDVAL, NEWVAL
INTEGER*2     BALANCE, SERIAL
CHARACTER     CODE_NAME*15, PART_ID*8
. . . . . .
. . . . . .
BALANCE (dmarg1,dmarg2,...,dmargn) = expression
CODE_NAME (dmarg1,dmarg2,...,dmargn) = expression
```

defines BALANCE as being a statement function that returns an integer value, and CODE_NAME as being a statement function that returns a 15-character string. The other declared variables (CORRECT_WT, OLDVAL, NEWVAL, SERIAL, and PART_ID) are included just to show that the statement function names are not declared any differently from any other names.

**11.2.1.2 The Dummy Argument List** The data type for each of a function's arguments is defined by the data type of each corresponding item in the dummy argument list. Note that the data type of a function name (and, therefore the data type of the delivered result) has nothing to do with the data types of any of the arguments. All of those items are independent of each other, and they are individually defined.

An external function's argument types are described by declarations inside that function. For instance, the statements

```
REAL FUNCTION   AMTDUE (EXDAYS)
INTEGER   EXDAYS
```

tell us (and FORTRAN) that the function AMTDUE expects one integer argument and returns a real value.

When it comes to statement functions, the data types for the dummy argument names are specified along with the other declarations in the same way as we handle the function name. For example, the sequence

```
REAL          XVAL, YVAL, PAYVAL
INTEGER       NREG, NSPECL
PAYVAL   (XVAL, NREG, NSPECL)  = expression
```

defines a statement function PAYVAL that expects three arguments. The first is real and the last two are integer values. The returned value is real.

**Example 11.1** We shall design a function subprogram that computes the amount of money to be charged for an overdue rental item. The function, to be named AMTDUE, is given an integer indicating the number of overdue days and the computation is done according to the following schedule:

1.00 dollar per day for the first 10 overdue days
1.50 dollars per day for the next 5 overdue days
1.75 dollars per day for the next 5 overdue days
2.00 dollars per day for the remaining overdue days

We shall specify this process using the name EXDAYS as the dummy argument. The first version will be a relatively simple-minded "brute force" implementation to illustrate the construction of a complete function involving several processing statements:

```
REAL FUNCTION   AMTDUE (EXDAYS)
INTEGER EXDAYS
IF (EXDAYS . LE. 10)   THEN
  AMTDUE = 1.0*EXDAYS
ELSE IF (EXDAYS . LE. 15)   THEN
       AMTDUE = 10.0 + 1.50*(EXDAYS-10)
     ELSE IF (EXDAYS . LE. 20)   THEN
          AMTDUE = 17.5 + 1.75*(EXDAYS-15)
        ELSE
          AMTDUE = 26.25 + 2.0*(EXDAYS-20)
        END IF
     END IF
END IF
RETURN
END
```

Now that we have this sequence of statements, there is an excellent opportunity to design a driver that will provide a simple but realistic environment for this function. Recall (Section 11.1.2.1) that we want the simplest possible vehicle that will do the job. Consequently, we need something that will invoke AMTDUE with a single integer argument. We do not care where that argument comes from or how it is produced. The point is that AMTDUE needs an integer argument, and we shall supply one in the easiest way we can. One possibility is simple assignment; another is initialization. Accordingly, our driver can look something like the one shown below. (Recall that a function is invoked in the same way as one of FORTRAN's built-in functions, described in Chapter 6.)

```
PROGRAM         DRVAMT
REAL            FINE
INTEGER*2       DAYS
DAYS = 12
FINE = AMTDUE (DAYS)
PRINT *, 'DAYS = ',DAYS,' PENALTY =',FINE
STOP
END
```

This is a simple driver, but it does not offer a particularly complete test of AMTDUE. The result will tell us how AMTDUE behaves with an argument of 12. A more revealing driver (for this example) would include a test for each of the ranges specified in the function. We can do that without really complicating the driver by making DAYS a four-element array, with each value falling inside one of the prescribed ranges:

```
PROGRAM         DRVAMT
REAL            FINE
INTEGER*2       DAYS(4),I
DATA            DAYS /8,11,17,32/
DO I=1,4
  FINE = AMTDUE (DAYS(I))
  PRINT *, 'DAYS = ',DAYS(I),' PENALTY = ',FINE
END DO
STOP
END
```

With a little further thought, we can produce the same result without the complications of the ELSE IF statements. Instead of testing the argument to see in which range it falls, we can compute the amount another way, by recognizing the fact that there will be at least a 1.00 daily charge for *all* of the overdue days, an additional 0.50 daily charge for *all* days over 10, an additional 0.25 daily charge for *all* days over 15, and a final addition, i.e., 0.25, for all days beyond 20. Following this approach, our function, greatly simplified, looks like this:

```
 REAL FUNCTION   AMTDUE (EXDAYS)
 INTEGER*2   EXDAYS
 AMTDUE = 1.0*MAX (EXDAYS,0) + 0.5*MAX (EXDAYS-10,0)
1          + .25*MAX (EXDAYS-15,0) + 0.25*MAX (EXDAYS-20,0)
 RETURN
 END
```

This still is a function named AMTDUE that expects a single integer argument and returns a real value. Note also that its behavior can be observed using the same driver that we specified before, without any changes.

Since the actual computation (in this revised version) requires only a single expression, we can recode it as a statement function. Of course, we must recognize that this would restrict its use to the program in which it is placed. However, our purpose here merely is to show the physical possibilities. Accordingly, the part of the program of interest to us would appear as follows:

```
 PROGRAM      EX1101
 REAL         AMTDUE
 INTEGER*2    EXDAYS
 . .
 . .
 AMTDUE (EXDAYS) = 1.0*MAX (EXDAYS,0) + 0.5*MAX (EXDAYS-10,0)
1          + .25*MAX (EXDAYS-15,0) + 0.25*MAX (EXDAYS-20,0)
 . . . . . .
 . . . . . .
```

### 11.2.2 Definition of Subroutines

The SUBROUTINE statement that begins each subroutine conveys the same type of information that the FUNCTION statement does for function subprograms. The only difference is that there is no particular connection between the subroutine's name and the type of information that it delivers. Accordingly, the general form for the SUBROUTINE statement is as follows:

SUBROUTINE name (dmarg1,dmarg2,...dmargn)

As is true for a function, the list of dummy arguments indicates the number and types of actual arguments to be supplied when the subroutine is invoked. (That number may be zero, in which case no dummy list is included.) Since the subroutine's name is not associated with a location in which a data value is to be stored, it does not have a specific data type and the name is not declared except in the SUBROUTINE statement.

**Example 11.2** We shall rewrite the simple computation in Example 11.1 just to show the basic differences when the process is implemented as a subroutine:

```
      SUBROUTINE   AMTDUE (EXDAYS, OVRAMT)
      REAL         OVRAMT
      INTEGER*2    EXDAYS
      OVRAMT = 1.0*MAX (EXDAYS, 0) + 0.5*MAX (EXDAYS-10, 0)
     1         + .25*MAX (EXDAYS-15, 0) + 0.25*MAX (EXDAYS-20, 0)
      RETURN
      END
```

Note that the subroutine version shows two dummy arguments and, therefore, expects two actual arguments. The first one represents the number of overdue days (as it did in the function). Now, in addition, a second argument must be provided in which the subroutine will store the computed result.

A driver for this implementation need not be any more complicated than the one we wrote for the function in Example 11.1. The invocation changes, but that really is all. (A subroutine is invoked by means of a CALL statement, as mentioned earlier.)

```
PROGRAM         DRVAMT
REAL            FINE
INTEGER*2       DAYS(4), I
DATA            DAYS /8, 11, 17, 32/
DO I = 1, 4
  CALL AMTDUE (DAYS(I), FINE)
  PRINT *, 'DAYS = ', DAYS(I), '  PENALTY = ', FINE
END DO
STOP
END
```

## 11.3 INVOCATION OF SUBPROGRAMS

The use of subprograms must be part of any discussion of their structure and properties. Since there already has been an opportunity to become acquainted with the basic forms for invoking the various types of subprograms that FORTRAN recognizes, we shall now examine these capabilities for invocation more closely.

When a subprogram is invoked, FORTRAN sets up an association between each item in the dummy argument list and the corresponding value in the actual argument list. Then, every time a particular dummy argument name appears in the body of the subprogram, the program uses the appropriate actual argument.

We shall illustrate this general association by defining and then using a simple statement function. The program segment is shown below. As in other examples, we

assume that the variables have values assigned to them as a result of activities not shown here:

```
      PROGRAM   VECTCP
      REAL      VSUM1, DARG1, DARG2, HORIZ, VERT,
     1          FHZ, FVT, W, MAXV, FINALV
      VSUM1(DARG1, DARG2) = SQRT(DARG1**2 + 0.8*DARG2**2)
                .....
                .....
      MAXV = 3.6/W + VSUM1(HORIZ, VERT)
                .....
      FINALV = MAXV - 0.72*VSUM1(FHZ, FVT)
```

For a statement function, the body of the subprogram consists merely of the expression on the right hand side of the assignment operator. In this example, the computations tell us that, when VSUM1 is invoked, the function will:

1. Square the value in the first argument.
2. Square the value in the second argument.
3. Multiply the squared second argument by 0.8.
4. Add the squared value of the first argument to the product developed in 3.
5. Invoke the SQRT built-in function to compute the square root of the result obtained in 4.
6. Store the result from 5 in VSUM1, thereby making it available to the expression from which the function was invoked.

In response to the first invocation, the program associates HORIZ with DARG1 and VERT with DARG2. As a result, the program does the computations by using HORIZ where DARG1 appears in the description and VERT where DARG2 appears. Thus, it is

```
SQRT(HORIZ**2 + 0.8*VERT**2)
```

that is added to 3.6/W to produce the value eventually assigned to MAXV. Similarly, when the function is invoked for the second time, the program associates FHZ with DARG1 and FVT with DARG2, and the computations take place accordingly. That is, SQRT(FHZ**2 + FVT**2 + FVT**2) is multiplied by 0.72 and the result subtracted from MAXV.

With this understanding, we can look at the invocation for each type of subprogram.

### 11.3.1 Invocation of Functions

Previous illustrations already have given us a reasonable grasp of the basic mechanism involved in using a function. Consequently, this section will serve to reinforce these concepts by means of an example.

**Example 11.3** The flow of a fluid through a pipe of some kind can be computed by the formula

wtflow = density × area × velocity

where wtflow is the flow in mass per unit time (e.g., lbs/second or kgms/minute), density is the fluid density in mass per unit volume (e.g., lbs/cubic foot, gms/cubic centimeter, etc.), area is the effective flow area of the pipe, and velocity is the fluid velocity (e.g., feet/second, meters/second, etc.), all in consistent units. We shall write a program that computes such flows for a variety of conditions in which the flow area is in the shape of an annulus (Figure 11.6) having outer diameter D2 and inner diameter D1. Each set of input consists of a density (DENSITY), velocity (VELOCITY), D2, and D1, in that order. The program will use these values to compute and print a mass flow, WT_FLOW. The area of an annulus, of

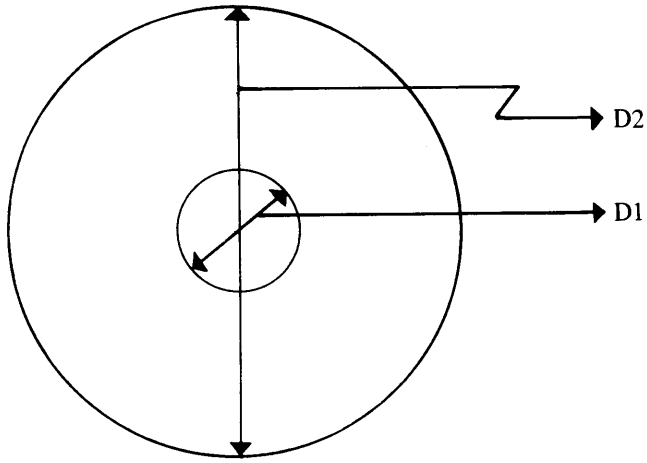
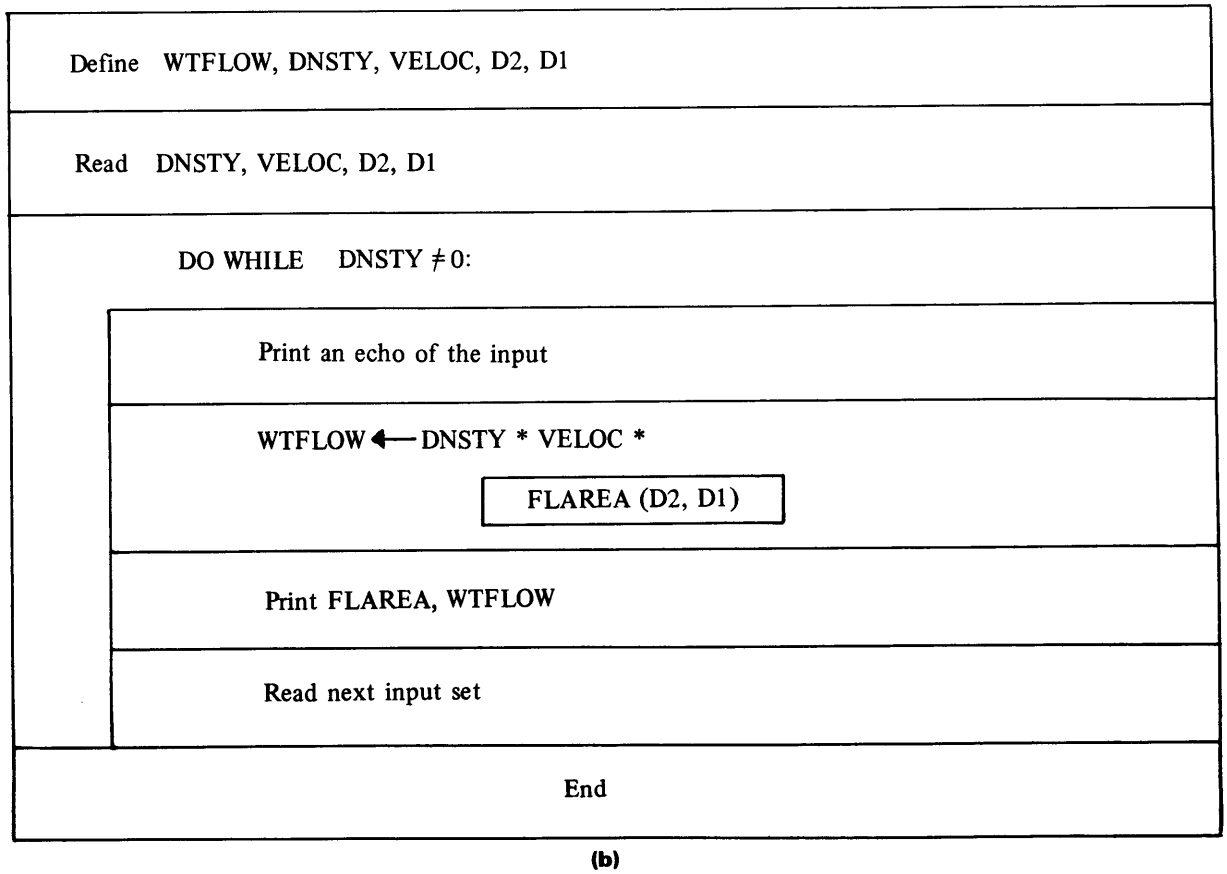



**FIGURE 11.6** Subroutine Diagram of Annulus Area, the Difference Between the Areas of Two Concentric Circles.

"Define WTFLOW, DNSTY, VELOC, D2, and D1."
"Read the first values for DNSTY, VELOC, D2. and D1."
**while**the density is not zero:
    "Echo the input values just read."
    "Compute WTFLOW = DNSTY * VELOC * area; obtain the area by using the function FLAREA on D2 and D1."
    "Print the effective area (FLAREA) and WTFLOW."
    "Read the next set of values for DNSTY, VELOC, D2, and D1."
**endwhile**
"Stop."

**(a)**




**(b)**

**FIGURE 11.7** **(a)** Pseudocode for Example 3's Main Program. **(b)** N-S Description of the Main Program for Example 11.3.

course, is the difference between the area of the circles formed by the outer and inner diameters:

annulus area = 0.7854 * (D2**2 - D1**2)

The *effective* area is that area actually available for flow. For our purposes, we shall assume that the effective area depends on the ratio D1/D2: if D1/D2 is less than 0.7, the effective area is 96.5 percent of the physical area; otherwise it is the same as the physical area.

We shall design our structure so that the main program handles the input, output, and computation of the flow. When it comes to computing the effective area, the job will be turned over to a function FLOW_AREA. This function, using the two diameters as arguments, will determine whether a correction is needed for effective area or not, and the resulting area will be returned in FLOW_AREA. A density of zero will end the run.

Even though this is a rather straightforward program, it still serves as a useful illustration of the techniques described in Section 11.1.2. Accordingly, we shall treat the main program and the function FLOW_AREA as two distinct items, each of which can be developed separately. To begin with, the main

```
************************************************************
*                      EXAMPLE 11.3                        *
************************************************************
*  WT_FLOW:       MASS FLOW IN KGM/SECOND                  *
*  DENSITY:        FLUID DENSITY IN KGM/CUBIC METER        *
*  VELOC:          FLUID VELOCITY IN METERS/SECOND         *
*  D2,D1:          OUTER AND INNER DIAMETER                *
*                            IN METERS                     *
*  FLOW_AREA:     THE NAME OF THE FUNCTION THAT COMPUTES*
*                           THE EFFECTIVE FLOW AREA        *
************************************************************

      PROGRAM          EX1103
      IMPLICIT         NONE
      REAL             WT_FLOW,DENSITY,VELOC,D2,D1

      WRITE (1,*) 'ENTER INITIAL VALUES FOR DENSITY,VELOC,D2,D1'
      READ (1,*) DENSITY,VELOC,D2,D1

      DO WHILE (DENSITY .NE. 0.0)
         PRINT *, '      '
         PRINT *, 'DENSITY = ',DENSITYY,' KGM PER CUBIC METER'
         PRINT *, 'VELOCITY = ',VELOC,' METERS PER SECOND'
         PRINT *, 'OUTER DIAMETER = ',D2,' METERS'
         PRINT *, 'INNER DIAMETER = ',D1,' METERS'
         WT_FLOW = DENSITY * VELOC * FLOW_AREA (D2,D1)
         PRINT *, 'EFFECTIVE FLOW AREA = ',FLOW_AREA,' SQUARE METERS'
         PRINT *, 'MASS FLOW = ',WT_FLOW,' KGMS PER SECOND'
         WRITE (1,*) 'ENTER NEXT SET OF INPUT VALUES'
         READ (1,*) DENSITY,VELOC,D2,D1
      END DO

      STOP
      END

************************************************************
*   THE STUB FOR FLOW_AREA MERELY ACCEPTS THE TWO     *
*   ARGUMENTS AND ASSIGNS A VALUE OF 1 FOR THE AREA.  *
************************************************************

      REAL FUNCTION  FLOW_AREA (OUTERD,INNERD)
      REAL           OUTERD,INNERD

      FLOW_AREA = 1.00
      RETURN
      END
```

**FIGURE 11.8** Statements for Example 11.3 (Full Main Program and Stub for FLAREA).

program is simple enough so that its general description (Figure 11.7) already comes close to a detailed representation. Our attention, then, can focus on providing a complete program structure from the very outset by devising a stub for FLOW_AREA. Since the main program expects to receive a value for the area, the temporary FLOW_AREA must deliver one. An easy way to do that is to assign a value (rather than compute it). We shall assign a value of 1.00 for convenience. Thus, our stub for FLOW_AREA is nothing more than this single assignment, supported by the basic declarations and return. The statements are shown in Figure 11.8, together with those for the main program.

FLOW_AREA needs to check the ratio of the two diameters to determine the coefficient's value. Once that is done, the computation of the effective area is straightforward. Figure 11.9 shows a basic representation of FLOW AREA in pseudocode and N-S forms. FLOW_AREA requires some attention because of some special things that were done in it. Note that, although it is set up with two dummy arguments (OUTERD and INNERD), a third variable name (COEFF) appears in the function's REAL declarations. The fact that COEFF is not in the list of dummy arguments makes it different from OUTERD and INNERD; it is an actual variable that is created for use in and by the function. The information assigned to that variable is needed by the function in preparing its final result, but it is not delivered to the invoking program. In fact, COEFF is not available outside the function (the invoking program does not "know" it is there), and it does not even exist once FLOW_AREA completes its processing. Each time FLOW_AREA is invoked, storage for COEFF is allocated once again, to be used while FLOW_AREA executes, and to be wiped out once FLOW_AREA returns. (The same is true with the parameter PIOVR4.) Such variables (there may be as many as the subprogram needs, in any mixture of types) are called *local variables*. There are ways to prolong the existence of such variables, but they are used for special purposes, and we shall not discuss them until the next chapter. Figure 11.10 gives a more detailed version of FLOW_AREA that leads directly to the statements themselves.

Independent development of FLOW_AREA requires a simple driver. In this case, it needs only to supply FLOW_AREA with two arguments (diameters) and receive an area value from the function. This driver is shown as part of Figure 11.11, along with the statements for FLOW_AREA. (A more elaborate driver might be designed to invoke FLOW_AREA several times with a succession of argument values.) Once we are convinced that the components work, the overall construction of the final program is completed by replacing the FLOW_AREA stub (from Figure 11.8) with the full version (from Figure 11.11) and throwing away the driver (from Figure 11.11).

"Define (dummy) arguments."
"Compute proper coefficient based on ratio of diameters."
"Compute effective area = Pi/4 * coefficient *
((outer diameter)**2—(inner diameter)**2)."
"Return to invoking program."

**(a)**

| Define (dummy) arguments for diameters |
|---|
| Select coefficient based on diameter ratio |
| Compute area = $\frac{\pi}{4}$ (coefficient) (outerd $^2$ - innerd $^2$) |
| Return |

**(b)**

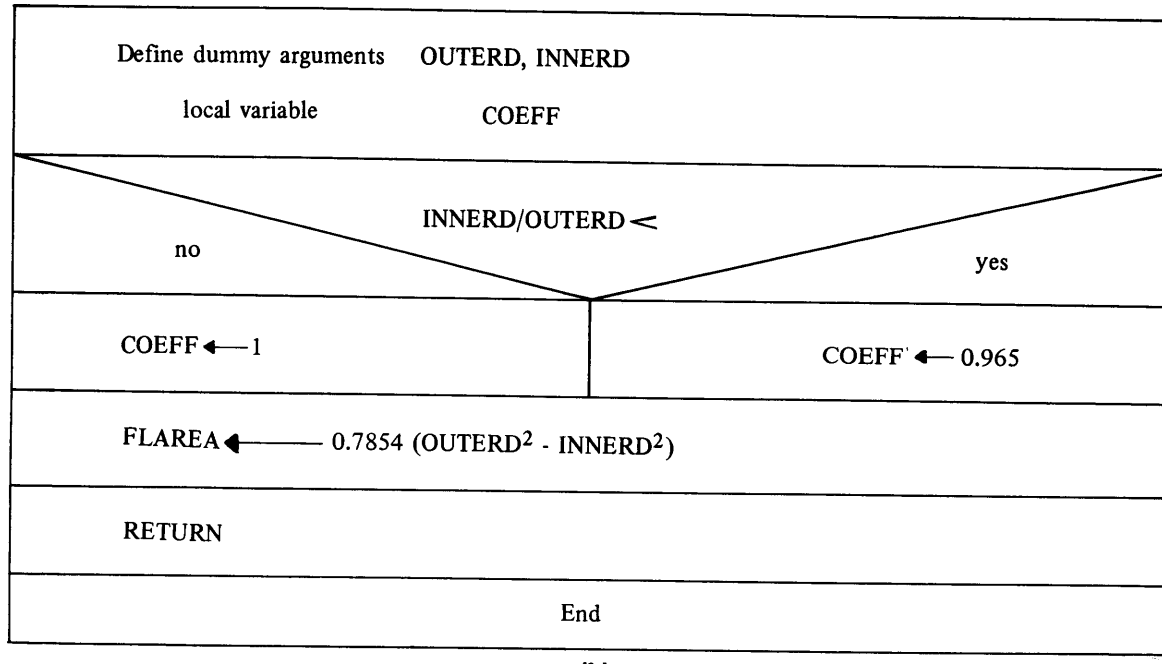
**FIGURE 11.9** **(a)** General Description of FLAREA for Example 11.3 **(b)** Basic N-S Representation of FLAREA.

"Define (dummy) arguments OUTERD and INNERD, local variable COEFF."
**if**
the ratio of inner to outer diameter is less than 0.7
**then**
"Set COEFF to 0.965."
**else**
"Set COEFF to 1.0."
**endif**
"Compute FLAREA = (3.14159/4)*COEFF*(OUTERD+INNERD)*

"Return to invoking program."

**(a)**

FLAREA:




**(b)**

**FIGURE 11.10** **(a)** Detailed Description of FLAREA for Example 11.3. **(b)** Detailed N-S Representation of FLAREA.

### 11.3.2 Invocation of Subroutines

The basic use of the CALL statement already has been introduced, so that our first look at subroutine usage is almost complete. (We shall examine some further concepts relating to subprogram arguments in the next chapter.) Accordingly, we can summarize by constructing an example main program/subroutine structure.

**Example 11.4** We shall expand the procedure in Example 11.3 by imposing a small additional complication with regard to the effective flow area (FLOW_AREA in the previous example). This time, instead of having a coefficient of either 1 or 0.965, the coefficient will depend on the ratio of inner diameter to outer diameter according to the formula

```
COEFF = 1.0 - 0.0637*(INNERD/OUTERD) ** 1.87
```

For each set of input, the program is to print the coefficient, as well as the other items printed in the previous example.

```
************************************************************
*                  DRIVER FOR FLOW_AREA                    *
************************************************************
      PROGRAM       FLADRV
      IMPLICIT      NONE
      REAL          D2,D1,AREA

      D2 = 4.0
      D1 = 2.0
      AREA = FLOW_AREA (D2,D1)
      PRINT *, 'D2:',D2
      PRINT *, 'D1:',D1
      PRINT *, 'AREA FROM FLOW_AREA:',AREA
      STOP
      END

************************************************************
*                       FLOW_AREA                          *
************************************************************
*  THE FUNCTION FLOW_AREA COMPUTES THE EFFECTIVE AREA OF AN*
*  ANNULUS HAVING OUTER DIAMETER OUTERD AND INNER DIAMETER  *
*  INNERD. THE FORMULA IS                                   *
*    FLOW_AREA = COEFF*.7854*(OUTERD**2-INNERD**2)         *
*  .7854 IS EXPRESSED AS THE PARAMETER PIOVR4 AND THE       *
*  DIFFERENCE OF THE SQUARED DIAMETERS IS REWRITTEN AS      *
*  (OUTERD+INNERD)*(OUTERD-INNERD).                         *
************************************************************

      REAL FUNCTION     FLOW_AREA (OUTERD,INNERD)
      REAL              OUTERD,INNERD,COEFF,PIOVR4
      PARAMETER         (PIOVR4=3.14159/4.0)
*
      IF (INNERD/OUTERD .LT. 0.7) THEN
        COEFF = 0.965
      ELSE
        COEFF = 1.0
      END IF
      FLOW_AREA = COEFF * PIOVR4 * (OUTERD+INNERD) * (OUTERD-INNERD)
      RETURN
      END
```

**FIGURE 11.11** Statements for FLAREA and Its Driver.

In line with the new requirement, our subprogram now will be a subroutine that uses the two diameters to determine a coefficient and, in turn, an effective area. Accordingly, the subroutine, which we shall name ANNLAR, will require four arguments: the two diameters, a place to store the computed coefficient, and a place to store the computed effective area. The program is shown in Figure 11.12. (The flowchart and pseudocode change little; the same is true with the driver and stub.)

### 11.3.3 Multiple Entries in a Subprogram

When a multistatement function or subroutine is invoked, execution of the subprogram begins with the first executable statement following the FUNCTION or SUBROUTINE

```
*************************************************************
*                      EXAMPLE 11.4                         *
*************************************************************
*  WT_FLOW:      MASS FLOW IN KGM/SECOND                     *
*  DENSITY:       FLUID DENSITY IN KGM/CUBIC METER           *
*  VELOC:         FLUID VELOCITY IN METERS/SECOND            *
*  D2,D1          THE OUTER AND INNER DIAMETERS,             *
*                   RESPECTIVELY, IN METERS                  *
*  FLOW_AREA:    THE EFFECTIVE FLOW AREA, IN SQ. METERS*
*  FLCORR:        THE FLOW COEFFICIENT                       *
*  ANNLAR:        A SUBROUTINE FOR COMPUTING FLOW AREA       *
*************************************************************
      PROGRAM           EX1104
      IMPLICIT          NONE
      REAL              WT_FLOW,DENSITY,VELOC,D2,D1,FLOW_AREA,FLCORR
      CHARACTER*6       BLANKS
      PARAMETER         (BLANKS='      ')
      WRITE (1,*) 'ENTER INITIAL VALUES FOR DENSITY,VELOC,D2,D1'
      READ (1,*) DENSITY,VELOC,D2,D1

      DO WHILE (DENSITY .NE. 0.0)
         PRINT *, BLANKS
         PRINT *, 'DENSITY = ',DENSITY,' KGM PER CUBIC METER'
         PRINT *, 'VELOCITY = ',VELOC,' METERS PER SECOND'
         PRINT *, 'OUTER DIAMETER = ',D2,' METERS'
         PRINT *, 'INNER DIAMETER = ',D1,' METERS'
         CALL ANNLAR (D2,D1,FLCORR,FLOW_AREA)
         WT_FLOW = DENSITY * VELOC * FLOW_AREA
         PRINT *, 'FLOW COEFFICIENT = ',FLCORR
         PRINT *, 'EFFECTIVE FLOW AREA = ',FLOW_AREA,' SQUARE METERS'
         PRINT *, 'MASS FLOW = ',WT_FLOW,' KGMS PER SECOND'
         WRITE (1,*) 'ENTER THE NEXT SET OF INPUT VALUES'
         READ (1,*) DENSITY,VELOC,D2,D1
      END DO

      STOP
      END

*************************************************************
*  THE SUBROUTINE ANNLAR COMPUTES A FLOW COEFFICIENT*
*  AND AN EFFECTIVE FLOW AREA FOR AN ANNULUS WITH    *
*  OUTER AND INNER DIAMETERS OUTERD AND INNERD,      *
*  RESPECTIVELY. THE FLOW COEFFICIENT (COEFF) IS     *
*  COMPUTED AS 1.0-0.0637*(INNERD/OUTERD)**1.87      *
*************************************************************
      SUBROUTINE ANNLAR (OUTERD,INNERD,COEFF,AREA)
      PARAMETER         (PIOVR4=0.25*3.14159)
      REAL              OUTERD,INNERD,COEFF,AREA,RATIO
      COEFF = 1.0 - 0.0637*(INNERD/OUTERD)**1.87
      AREA = COEFF * PIOVR4 * (OUTERD+INNERD) * (OUTERD-INNERD)
      RETURN
      END
```

**FIGURE 11.12** FORTRAN Statements for Example 11.4.

```
*************************************************************
*  THIS PROGRAM CONSISTS OF A MAIN PROGRAM (CMPADJ)         *
*  THAT USES TWO SUBPROGRAMS: A FUNCTION NAMED COMPAT       *
*  WITH SECONDARY ENTRY POINT COMPAT1, AND A SUBROUTINE     *
*  NAMED EXPLR WITH SECONDARY ENTRY POINT EXPLR1.           *
*  WHEN EXPLR IS INVOKED WITH ITS PRIMARY ENTRY POINT,      *
*  IT REQUIRES THREE ARGUMENTS; A CALL TO EXPLR1 REQUIRES   *
*  FOUR ARGUMENTS. SIMILARLY, AN INVOCATION OF THE          *
*  FUNCTION COMPAT (THE PRIMARY ENTRY POINT) REQUIRES TWO   *
*  ARGUMENTS WHILE AN INVOCATION WITH THE SECONDARY ENTRY   *
*  POINT (COMPAT1) REQUIRES ONLY A SINGLE ARGUMENT.         *
*************************************************************
      PROGRAM                  CMPADJ
      REAL                     OLDVAL,NEWVAL,RES,FACTOR,INTVAL,FINAL
      INTEGER*2                TRYNUM,SEQNUM
   11 statement
   12 RES = FACTOR * COMPAT (OLDVAL,NEWVAL)
   13 statement
         ....
   20 statement
   21 CALL  EXPLR (RES,INTVAL,TRYNUM)
   22 statement
         ....
   28 statement
   30 CALL  EXPLR1 (RES,INTVAL,NEWVAL,TRYNUM)
         ....
   40 FINAL = RES * COMPAT1 (NEWVAL)
         ....
      STOP
      END

      REAL FUNCTION    COMPAT (RVAL1,RVAL2)
      REAL             RVAL1,RVAL2,COMPAT1
  101 statement
  102 statement
        ...
      ENTRY            COMPAT1 (RVAL2)
  111 statement
        ...
      COMPAT = expression
  115 RETURN
      END

      SUBROUTINE       EXPLR (RV1,IN1,IN2)
      REAL             RV1,RV2
      INTEGER*2        IN1,IN2
  201 statement
         ...
      ENTRY            EXPLR1 (RV1,IN1,RV2,IN2)
  205 statement
         ...
         ...
       RETURN
       END
```

**FIGURE 11.13** Subprogram Structure with Multiple Entry Points.

statement and the declarations. Sometimes it is desirable to be able to begin a subprogram's processing from more than one point in that subprogram. This is easily done in FORTRAN by providing a subprogram with as many different *entry points* as the situation requires. Each entry point has its own name and its own argument list, so that the invocation process is the same regardless of which entry point the invoking program specifies.

Figure 11.13 shows exactly how this works. Note that the main program makes two CALLS, each with its list of arguments. The invocations look ordinary enough, and that is exactly what they are. In fact, there is nothing in the CALLS themselves to persuade us that they are not invoking two different subroutines; nothing, that is, until we look at the accompanying subroutine. We see that, in addition to the regular SUBROUTINE statement that begins the subprogram, there is an *ENTRY statement.* This specifies its own *entry name* (which, of course, must be different from the name given in the SUBROUTINE statement) and its own list of dummy arguments. These may or may not be the same as those attached to the SUBROUTINE statement, depending on what the subprogram is designed to do.

The same is true in Figure 11.13 wih regard to the function invocations. The name used in each invocation corresponds to a name associated with the subprogram by declaring it either in the FUNCTION statement that begins the subprogram or in an ENTRY statement somewhere inside it.

Having established this framework, we can take a closer look at the processing in Figure 11.13:

1. COMPAT is invoked as part of the computing in statement 12. Accordingly, processing in that function begins with statement 101 and proceeds until the RETURN is executed (statement 115).
2. The main program continues in sequence, through statement 20.
3. Statement 30 invokes the EXPLR subroutine at its primary entry point, so that this subprogram begins its work at statement 201. Control returns to statement 22 when EXPLR concludes.
4. The main program continues through statement 28.
5. At statement 30 the main program invokes EXPLR a second time. In this case, however, the CALL specifies EXPLR1, a secondary entry point. In response, processing in EXPLR now starts at statement 205 and proceeds from there to the conclusion of the routine. Note that the argument list for EXPLR1 differs from that for EXPLR. This is legal, as long as all the dummy arguments are declared and their use is defined in the routine.
6. The main program continues to statement 40, at which point it invokes COMPAT a second time, using the secondary entry point COMPT1. Processing, then, proceeds from statement 111 through the remainder of the function, followed by a return to the main program.
7. The main program concludes.

Supporting this usage are the declarations in the subprograms themselves. Thus, COMPT1 is defined with the ENTRY statement in the function COMPAT. Note that the name also appears as a REAL declaration to indicate that the function will return a REAL value when invoked as COMPT1. The programmer can design the function so that if it always computes a result of the same data type (regardless of the entry point), that result can be delivered either in the variable having the function name or the entry point name. Doing this is simply a matter of assigning the result to the appropriate variable prior to the RETURN. Of course, if the result's data type is different for each entry point (this is legal), the subprogram must be designed to place the result in the variable declared with the appropriate data type.

An alternate entry point may be placed almost anywhere in a subprogram. There are two basic restrictions:

1. The ENTRY statement cannot be placed in the middle of a DO loop.
2. It cannot be placed in the middle of an IF block.

The point should be made here that, in general, the use of multiple entry points tends to make the use of a subprogram more complicated. That is, the invoking program has to "know" more about how the subprogram works and exactly under what conditions each entry is used. Consequently, an effort should be made to provide a subprogram with a single entry. It often is better to keep such decisions buried inside the subprogram as part of its processing details. What the outside world sees, then, is a simple structure with a single entry point. An example of this practice is seen in Chapter 6, where we discussed the unification of FORTRAN's built-in functions so that each could be invoked with a single entry point (function name) regardless of the arguments' data types.

### 11.3.4 Multiple Returns from a Subprogram

There are many occasions where it is convenient to conclude processing at several different places in a subprogram. A typical case is one in which the processing in the subprogram may or may not be performed depending on the outcome of some test. This is illustrated by the subprogram in Figure 11.14.

The function EVAL takes three real arguments and produces a real number as its result. If either of the first two arguments is zero, or the product of the first two arguments is not less than the square root of the third, no further computations are to be performed. Instead, the program that invokes EVAL decides what to do next depending on the value EVAL delivered. The version shown in Figure 11.14(a) handles this by branching to the RETURN statement, thereby avoiding the computation of EVAL. An alternative (Figure 11.1(b)) replaces the GO TO statement with an immediate return.

From a structural point of view, the version in Figure 11.14(a) is preferred since it provides the subroutine with a single point from which it returns.

```
      REAL FUNCTION    EVAL (DARG1,DARG2,DARG3)
      REAL             DARG1,DARG2,DARG3
      EVAL = 0.0
      IF (DARG1 .EQ. 0.0 .OR. DARG2 .EQ. 0.0
     1    .OR. DARG1*DARG2 .GE. SQRT(DARG3) )  GO TO 100
      EVAL = (DARG1 ** 0.878 + LOG10(DARG2 ** 2))/(DARG1*DARG2+DARG3)
100   RETURN
      END
```

(a)

```
      REAL FUNCTION    EVAL (DARG1,DARG2,DARG3)
      REAL             DARG1,DARG2,DARG3
      EVAL = 0.0
      IF (DARG1 .EQ. 0.0 .OR. DARG2 .EQ. 0.0)
     1    .OR. DARG1*DARG2 .GE. SQRT(DARG3) ) RETURN
      EVAL = (DARG1 ** 0.878 + LOG10(DARG2 **2))/(DARG1*DARG2+DARG3)
      RETURN
      END
```

(b)

**FIGURE 11.14** Use of Multiple Returns in a Subprogram.

### 11.3.5 Multiple Destinations for Returns from a Subroutine

In the previous section we saw that it is possible to return to an invoking program from several different points in a subprogram. Note that, for a given usage of a subprogram, all these points return control to a single place in the invoking program.

FORTRAN has an additional feature that allows the construction of a subroutine (but not a function) in which each RETURN statement may specify a *different* destination in the program that called the subroutine. It is mentioned here to indicate its existence; however, its discussion is deferred to an appendix because it is not considered a particularly helpful feature. In fact, its use encourages a tendency to complicate structures that ought to be kept simple. It is a much sounder practice to design subroutines that return to the place from which they are called, so that we preserve the idea of a subroutine presenting a package whose processing can be thought of as a single operation. Once the subroutine returns, the calling program then can take whatever action it needs to take depending on the outcome of the subroutine's processing.

## 11.4 SUMMARY

We can simplify program construction by treating certain tasks as individual conceptual operations. Once these have been identified, they can be isolated and designed as individual *subprograms*. Production of subprograms need not be tied to any specific program, so that once a subprogram is developed, it can be made available, as a unit, to any number of programs requiring its services.

FORTRAN recognizes three types of subprograms:

1. A *statement function* in which a single assignment statement describes the computations. The statement is placed inside the program using (*invoking*) the function. A statement function may be designed to operate on any number of data items (*arguments*), but it produces a single value each time it is invoked. The general form is

   name (dummy argument, dummy argument, etc.) = expression

2. A *function* consisting of any number of statements. Like a single statement function, this type of subprogram can operate on any number of arguments, producing a single result. Unlike the single statement function, this type of subprogram must appear outside other programs (see Figure 11.1). The general form is

```
FUNCTION name (dummy argument, dummy argument, etc.)
   . . . . . .
   statements
   . . . . . .
name = expression
   . . . . . .
RETURN
   . . . . . .
END
```

3. A *subroutine*, the most general type of subprogram. This structure accepts any number of arguments and can be designed to return any number of results. Subroutines cannot appear inside any other programs and have the following general form:

```
SUBROUTINE   name (dummy arg, dummy arg, etc.)
   . . . . . .
   statements
   . . . . . .
RETURN
   . . . . . .
END
```

Subprograms may be invoked by a main program or by other *subprograms*. A statement function may be invoked only by the program in which it is placed. No subprogram may invoke itself. When a subprogram is invoked, the invoking program supplies a list of *actual arguments*. These are matched, item for item, against the list of dummy arguments given in the subprogram's definition. Thus, the first actual argument is used wherever there is a reference to the first dummy argument, and so on.

A function (of either type) is invoked as part of all of an expression. For example,

```
Y = 2.42 * BVAC(X1, X2, X3)
```

invokes the function BVAC. The result (stored in a variable name BVAC) is delivered to the expression, where it is multiplied by 2.42 and the value thus produced is assigned to Y.

A subroutine is invoked by a separate statement such as

```
CALL subname (argument,argument, etc.)
```

Transfer from a subprogram back to the point at which it was invoked is handled automatically within FORTRAN by means of the RETURN statement.

## PROBLEMS

1. We have the following sequence of statements:

```
REAL      D1, D2, D3, RSLT
READ (1, *) D1, D2, D3
RSLT = SQRT(D1*D1 + D2*D2 + D3*D3)
PRINT *, D1, D2, D3
```

Rewrite the statements so that the same processing is done with the value stored in RSLT after being computed by a statement function named VECTR. (Include the statement function as well as its invocation.)

2. Consider the following sequence of statements:

```
CHARACTER          WD1*4, WD2*6
READ (1, *) WD1
WD2 = WD1//'ED'
PRINT *, WD1, WD2
```

Rewrite the statements so that the same processing is done with the value being stored in WD2 after being developed in a statement function named PAST. (Include the statement function as well as its invocation.)

3. Here is a sequence of statements:

```
REAL                A, B, C, NETVAL
INTEGER*2 N
READ (1, *) A, B, C
N = 2
IF (A/B .LT. 0.41) N=3
NETVAL = SQRT(A**N + B**N + C**N)
PRINT *, A, B, C, N, NETVAL
```

(a) Rewrite the statements so that the same processing is done with the value being stored in NETVAL after it is developed in a function named VADJST.

(b) Rewrite the statements so that the same processing is done with NETVAL being developed in a subroutine named VAJSUB.

4. We have the following statements:

```
INTEGER*2          SUBJCT, AGEYRS, CATEG
CHARACTER*7        CLASS
READ (1, *) SUBJCT, AGEYRS
CATEG = 0
IF (AGEYRS .LT. 7) THEN
  CATEG = 1
  CLASS = 'TOT'
ELSE IF (AGEYRS .LT. 13) THEN
    CATEG = 2
    CLASS = 'SUBTEEN'
  ELSE IF (AGEYRS .LT. 20) THEN
        CATEG = 3
        CLASS = 'TEEN'
      ELSE
        CATEG = 4
        CLASS = 'ADULT'
      END IF
END IF
PRINT * SUBJCT, AGEYRS, CATEG, CLASS
```

(a) Rewrite the statements so that the same processing is done with the values in CATEG and CLASS being assigned in two functions named AGECAT and AGECLS, respectively.
(b) Rewrite the statements so that the same processing is done with the values in AGECAT and CLASS having been computed in a subroutine named AGEGRP.

5. For each of the following program outlines, list the sequences in which the statements are executed. (The statements are all labeled for convenient reference.)

(a)
```
      PROGRAM     P1
    1 SF1(dummy arguments) = expression
    2 SF2(dummy arguments) = expression
    3 statement
    4 statement
    5 result = SF1 (arguments
    6 statement
    7 result = SF2 (arguments)
    8 statement
    9 statement
   10 result = SF2 (arguments) + SF1 (arguments)
   11 statement
      END
```

(b)
```
      PROGRAM     P1
    1 statement
    2 statement
    3 result = F1 (arguments)
    4 statement
    5 result = F2 (arguments) + F1 (arguments)
    6 statement
      END

      FUNCTION F2 (dummy arguments)
   11 statement
   12 statement
   13 statement
   14 result = F1 (arguments)
   15 RETURN
      END

      FUNCTION F1 (dummy arguments)
   21 statement
   22 statement
   23 RETURN
      END
```

(c)
```
      PROGRAM P1
    1 statement
    2 result = F1 (arguments)
    3 CALL S1 (arguments)
    4 result = F1 (arguments)
    5 CALL S1 (arguments)
    6 CALL S1 (arguments)
      END

      FUNCTION F1 (dummy arguments)
   11 SF1 (dummy arguments) = expression
   12 statement
   13 statement
   14 result = SF1(arguments)+SF1(arguments)
   15 statement
   16 RETURN
      END

      SUBROUTINE S1 (dummy arguments)
   21 statement
   22 statement
   23 result = F1 (arguments)
   24 RETURN
      END
```

(d)
```
   PROGRAM P1
 1 statement
 2 statement
 3 result = F1 (arguments)
 4 result = F2 (arguments)
 5 CALL S1 (arguments)
 6 statement
 7 CALL S1 (arguments)
   END

   SUBROUTINE S1 (dummy arguments)
11 statement
12 statement
13 CALL S2 (arguments
14 result = F1 (arguments)
15 RETURN
   END

   FUNCTION F1 (dummy arguments)
21 statements
22 CALL S2 (arguments)
23 result = F2 (arguments)
24 RETURN
   END

   FUNCTION F2 (dummy arguments)
31 statement
32 statement
33 statement
34 RETURN
   END

   SUBROUTINE S2 (dummy arguments)
41 statement
42 statement
43 statement
44 RETURN
   END
```

6. Show the output produced by the PRINT statement(s) in each of the following sequences of statements:

(a)
```
REAL      X, Y, Z, A, B, SQD
SQD (A, B) = 0.2 * (A+B) * (A–B)
X = 27.0
Y = 18.0
Z = SQD (Y, X)
PRINT * X, Y, Z
```

(b)
```
REAL      R, S, T, FWD, BKWD, A, B
FWD (A, B) = 2.0*A + B*B
BKWD (A, B) = 1.0/ (B + A*A)
R = 4.5
S = 10.0
T = FWD (R, S) + BKWD (S, R)
PRINT * R, S, T
```

(c)
```
REAL      X, Y, R
INTEGER*2 J
X = 2.4
Y = 3.6
J = 3
R = 0.5*BELV (X, Y, J) – BELV (Y, X, J)
PRINT * X, Y, J, R
  .....
END
REAL FUNCTION   BELV (Y, J, X)
REAL                 Y, J
INTEGER*2            X
IF (Y .GT. J) THEN
  BELV = (2.0*Y–1.8*J) **X
ELSE
  BELV = (2.2*J–1.8*Y) **X
END IF
RETURN
END
```

(d)
```
REAL      A, B, X, G, H
INTEGER*2   K
A = 3.1
B = 4.0
X = 5.0
K = 3
PRINT * A, B, X, K
CALL CHMPAX (A, B, X, K, G, H)
PRINT * G, H
   .....
END

SUBROUTINE CHMPAX (A1, A2, A3, A4, A5, A6)
REAL      A1, A2, A3, A5, A6
INTEGER*2   A4
A5 = A1 + A2*LOG(A3)
IF (A5 .LT. A1+A2) A4=A4-1
A6 = A2*A5**A4
RETURN
END
```

(e)
```
INTEGER*2      NV
CHARACTER*10      W1, W2
W1 = 'PROVINCIAL'
CALL CHMP (W1, W2, NV)
PRINT *, W1, NV, W2
     ......
END

      SUBROUTINE CHMP (C1, C2, I)
      INTEGER*2      I, J, K, L
      CHARACTER      C1*10, C2*10, CV*5
      I = 0
      CV = 'AEIOU'
      DO 12 J = 1, 10
        K = 11-J
        C2(K:K) = C1(J:J)
        DO 14 L = 1, 5
          IF (C1(J:J) .EQ. CV(L:L)) I=I+1
14    CONTINUE
12  CONTINUE
    RETURN
    END
```

7. Write a statement function named LOG9 that operates on a real argument to produce its logarithm to the base 9. Note that

   $\log_4$ of a number = ($\log_{10}$ of that number)/($\log_{10}$ of 9)

8. Write a statement function LOGB that operates on two arguments: the first, a positive integer, represents a base; the second is a positive real number. LOGB is to return the logarithm of the second argument using the first argument as the base.

9. The specifications in Problem 8 imply that the arguments are guaranteed to make sense. (That is, the base will be a positive integer and so will the number whose logarithm is desired.) In this problem, we remove that guarantee. As a result, the subprogram must do its own testing to make sure that it is computationally possible to perform the requested operations using the arguments given to it. Obviously, then, there will be too much processing to construct this as a statement function. Instead, write a subroutine named ANYLOG that uses the same two arguments as LOGB in Problem 8, along with another two arguments for results: the first of these is a real variable into which ANYLOG will place the results of its computations, and the second is an integer variable to which ANYLOG will assign a value that indicates what happened.

   (1) If the first two arguments are positive, the third argument will contain the computed logarithm, and the fourth argument will contain the integer value 0 to indicate that the computation proceeded normally.

   (2) If the first argument (the integer defining which base to use) is positive and the second argument is not, the third argument is set to −1.0E−20 and the fourth argument's value will be −1.

(3) If the first argument is not positive and the second argument is, the third argument is set to −1.0E−20 and the fourth argument's value will be −2.
(4) If both of the first two arguments are not positive, the third argument (as in the previous two cases) is to be −1.0E−20 and the fourth argument's value will be −3.

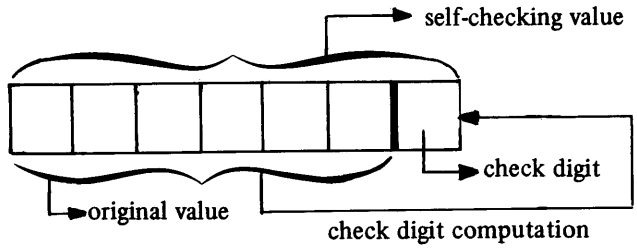
Test your subroutine with the following main program:

```
      PROGRAM        CH11P9
      IMPLICIT       NONE
      REAL           VALUE, LOGVAL
      INTEGER*2      BASE, STATUS
      READ (1,*) VALUE, BASE
      DO WHILE (VALUE .NE. 0.0 .AND. BASE .NE. 0)
        CALL ANYLOG (BASE, VALUE, LOGVAL, STATUS)
        PRINT *, BASE, VALUE, LOGVAL, STATUS
        READ (1,*) VALUE, BASE
      END DO
C
      PRINT *, 'END OF RUN.'
      STOP
      END
```

Here are some suggested input values:

| | |
|---|---|
| 2.0 | 2 |
| 2.0 | 4 |
| 4.0 | 11 |
| 12.5 | 0 |
| −61.2 | 12 |
| −3.0 | −6 |
| 0.0 | 0 |

10. There are many situations in which data require extra protection. For example, a credit card system must make sure that transactions are accepted for valid account numbers and rejected for those that are not valid. One technique that is used for this purpose converts data values to *self-checking numbers*. A self-checking number is formed by using the original data value to compute an extra digit. This *check digit* is attached to the original value and the new, self-checking number now becomes the data value that is stored and used:




Then, when a self-checking number is submitted for validation, the program removes the check digit and uses the remaining value to recompute the check digit. If the newly computed check digit matches the one that was submitted as part of the data, the number is validated; otherwise, it is rejected.

There are many ways to compute check digits. One that finds frequent use is the modulus 11 method. Each digit in the original value is given a weight, with the rightmost digit receiving a weight of 2, the next one receiving a weight of 3, and so on. The computation proceeds as follows:

(1) Each digit is multiplied by its weight.
(2) The results from step 1 are added together. (We shall call this sum SUMWTS for convenience.)
(3) The value MOD (SUMWTS, 11) is computed. (We shall call this value MODSUM for convenience.)
(4) The check digit is 11−MODSUM. If this result is greater than 9, the check digit for such cases is obtained by adding the two digits together. Thus, if MODSUM comes out to be 0, 11−0=11, and the final check digit is 1+1 or 2.

To see how this works, suppose a credit company assigns a five-digit identification number to each of its cardholders. Let us pick one, say, 20781. Following the steps outlined before,

```
SUMWTS = 2*1 + 3*8 + 4*7 + 5*0 + 6*2 = 66
MODSUM = MOD(66, 11) = 0
```

Check digit: 11−0 = 11; 1+1=2.

Thus, the 2 is attached to the original value and the self-checking identification number becomes 207812. This is the number that the cardholder would receive and use. Then, when the cardholder charges something to that account, the program reads the 207812, removes the (rightmost) 2, and recomputes the check digit. Since result (2) agrees with the value computed originally, the number is validated. A counterfeiter trying to use a fake card with a number like 207813, for instance, would fail because the check digit would be wrong.

Write a function named CHKDGT that performs the modulus−11 computation on a six-digit positive integer and returns a one-digit integer representing a check digit. To enable your function to execute, run it with the following main program. Note that this program asks for a check digit and uses it to build a seven-digit value:

```
PROGRAM          C11P10
IMPLICIT         NONE
INTEGER*2        OLDNUM, NEWNUM
READ (1, *) OLDNUM
DO WHILE (OLDNUM .NE. 0)
  NEWNUM = 10*OLDNUM + CHKDGT(OLDNUM)
  PRINT *, OLDNUM, NEWNUM
  READ (1, *), OLDNUM
END DO
PRINT *, 'END OF RUN. '
STOP
END
```

Here are some suggested input values:

```
111111
000369
707070
707072
000000
```

11. Using the technique described in the previous problem, write a function named CHKVAL that is designed to report on the validity of a seven-digit self-checking number. Each time it is invoked, CHKVAL returns a value of 1 if its seven-digit argument is valid and a 0 if it is not. For instance, CHKVAL(7070748) would return a value of 1, and CHKVAL(7070346) would return a value of 0. (NOTE: a rather effective way to construct CHKVAL is for CHKVAL to remove the rightmost digit from its argument and use the resulting six-digit value as an argument when it invokes CHKDGT, the function described in Problem 10.) Here is a main program that can be used for testing CHKVAL:

```
PROGRAM          C11P11
IMPLICIT         NONE
INTEGER*2        NEWNUM, STATUS
CHARACTER*7      REPORT
READ (1, *) NEWNUM
DO WHILE (NEWNUM .NE. 0)
  STATUS = CHKVAL(NEWNUM)
  IF (STATUS .EQ. 1) THEN
    REPORT = 'VALID'
  ELSE
    REPORT = 'INVALID'
  END IF
  PRINT *, NEWNUM, REPORT
  READ (1, *) NEWNUM
END DO
PRINT *, 'END OF RUN. '
STOP
END
```

12. Not all systems place the check digit at the end of a self-checking number. To help make things a little more complicated for the potential counterfeiter, rewrite the *CHKDGT* function (Problem 10) so that the computed check digit becomes the fourth digit in the new value. For instance, an argument of 207416 would produce a check digit of 8, and the final (self-checking) value, then, would be 2078416.

13. Write the function described in Problem 11 as a subroutine named VALID whose first argument is a seven-digit integer variable to be validated. The check digit is returned in the second argument and the status (1 for valid, 0 for invalid) is returned in the third argument. Change the main program given in Problem 11 so that this subroutine can be run under it.

14. Generalize the function described in Problem 10 so that it will produce a self-checking number for an argument whose size (i.e., number of digits) is not known until the function actually is invoked. This more versatile function, named ANYCHK, will have two integer arguments: the first specifies the original data value and the second specifies the number of digits in the first argument. Change the main program so that it can be used to test ANYCHK.

15. We can make the subprogram in Problem 14 even more general: Design a subroutine named GENCHK that returns a single check digit (as before) and the resulting self-checking number as well. GENCHK is to use the following five arguments:

    (1) The value to be processed.
    (2) The number of digits in the value.
    (3) The desired position of the check digit in the resulting self-checking value.
    (4) The variable in which the check digit is to be delivered.
    (5) The variable in which the resulting self-checking value is to be delivered.

    Thus, the statement

    ```
    CALL GENCHK (IDNUM, NUMDIG, CHKPOS, DIGIT, NEWID)
    ```

    says that the value to be processed is in IDNUM and its length is in NUMDIG. If the value in CHKPOS happened to be 3, then GENCHK is to store the computed check digit in DIGIT and also as the third digit of NEWID. That is, the third digit of IDNUM becomes the fourth digit of NEWID, and so on. Note that GENCHK must make sure the value in CHKPOS makes sense when compared to NUMDIG. For instance, if NUMDIG is 7, the addition of a check digit will make NEWID 8 digits long, so that a CHKPOS value of, say, 11 would not mean much.

    Write a simple (but suitable) main program for testing GENCHK.

16. Remember the glory of ancient Rome? At least part of that glory was mentioned in Problem 10 of Chapter 10. Using the information in that problem, design a function named ROMNUM that operates on a character string representing a Roman integer value and returns the equivalent value as an Arabic integer.

17. Write a function ARBNUM that is the reverse of the one specified in Problem 16. That is, ARBNUM operates on an integer value and returns a character string representing a Roman number of the same value.

18. Write a function named TRIM that operates on a 64-character string consisting of words separated by single blanks. The first word always starts in position 1 of the string and there always is exactly one blank between words. However, only part of the string may be occupied by the words. The unused portion of the string (the rightmost part of the string) is filled with blanks. Thus, in a fully occupied string, position 64 will have a letter in it.

    The function is to return an integer specifying the length of the occupied portion of the string, i.e., the number of characters up to and including the last letter of the last word in the string. This length includes the blanks between the words (but, obviously, does not include the blanks after the last word). Write a suitable main program to be used for testing TRIM.

19. Write a subroutine named NXTWRD that finds the next word in a string of words. Imagine that the invoking program has a character string containing so many words, with a single blank separating each word from the next one. This program would CALL the subroutine, telling it the name of the character string to be examined and the position at which the examination should start. You should assume that the starting position given to NXTWRD may not be consistent. It could contain the first letter of the next word or it could be the position of the blank just ahead of the next word. Note that there also is the possibility that there is no next word, and the subroutine must handle this possibility.

When its processing is completed, NXTWRD returns the word itself, its length, and the position of the *last letter* of the word it just found. If there was no next word, NXTWRD is to return blanks in the character string that would have contained the word, a value of zero for the word length, and a value of zero for the position of the "last letter." Assume that the string to be searched never will be longer than 64 characters, and that the longest word in the string will be 15 characters.

Write a simple main program to test NXTWRD. (NOTE: If you wish, this main program need not even perform any input. It may be sufficient to set up (via assignment, DATA, or PARAMETER statements) two or three character string constants and let NXTWRD search those. Then, the main program can print the values returned by NXTWRD, along with the original character strings.)

20. The arithmetic mean of a collection of values is defined as

$$\text{MEAN} = \frac{\text{SUMVAL}}{\text{NUMVAL}}$$

where SUMVAL is the sum of the individual values and NUMVAL is the number of values. Another statistical quantity, the *variance*, describes the dispersal of a collection around its arithmetic mean. This value can be computed as follows:

$$\text{VARNC} = \frac{\text{SMSQVL} - \text{SMVLSQ}}{\text{NUMVAL}}$$

where SUMVAL and NUMVAL have their previous meanings, SMSQVL is the sum of the individual squared values, and SMVLSQ is the square of SUMVAL. A third useful statistic that describes a collection of values is the *standard deviation*, computed as the square root of the variance:

$$\text{SD} = \sqrt{\text{VARNC}}$$

Write a subroutine named STATS that receives integer values for the sum of a collection of values, the sum of the squared values and the number of values. After its processing is completed, STATS returns the mean, variance, and standard deviation (all as real numbers).

Test STATS by invoking it from a simple main program. For instance, the main program need not compute SUMVAL, SMSQVL or NUMVAL; these simply can be read as input values and used directly as arguments to STATS.

21. In this problem we can use the subprograms developed for Problems 18 through 20 as building blocks to produce a grand construction that operates as follows: Input consists of any number of 64-character strings whose contents and organization are the same as described for Problem 18. The last input set is followed by a special set containing a character string whose first five positions are blank. This set signals the end of the run and is not to be processed as part of the data. For each set processed, the program is to print an echo of those data along with the number of words in that set. After all of the input has been processed, the program is to leave two blank lines and print the total number of sets processed, the total number of words found, the average word length, and the variance and standard deviation for the entire collection of input words.

22. This is a slightly more extensive version of the program described in Problem 21: input is the same as before. This time, for each set processed, the program is to print an echo of those data, the number of words in that set, and the average word length *for that set*, rounded to the nearest tenth of a letter. The summary output is the same as before.

23. Write a function named NDAYS that operates on two dates submitted to it as six integer arguments (month2, day2, year2, month1, day1, and year1). After operating on these values, NDAYS returns an integer indicating the number of days between the two dates. For example, NDAYS(6, 18, 1980, 6, 7, 1979) will return a value of 366 (remember that February of 1980 had a 29th day) plus 11 or 377. You may assume that both year2 and year1 will be in the twentieth century and that the date represented by the first three arguments always will be later in time than the date represented by the fourth, fifth and sixth arguments. Use an input date (month2,day2,year2) of zero to stop the run.

Write a simple main program for use with NDAYS. Here are some suggested lists of arguments for testing purposes:

```
10, 10, 1977, 10, 10, 1974
10, 10, 1977, 10, 10, 1957
 3, 16, 1981, 12, 31, 1972
12, 31, 1980,  3,  1, 1979
 0,  0,    0,  0,  0, 0000
```

24. In this more extensive version of Problem 23, design NDAYS so that it can handle improper arguments: all the values given to it still will be positive integers, but they may not be legitimate dates. For instance, values of 41 and 38 for month2 and day2, respectively, are positive integers but they do not make much sense. Also, NDAYS must recognize (and handle) a situation where the dates are reversed in time. (That is, the date that is supposed to be later in time is actually earlier.)

25. When a borrower returns books to the East Flapville Public Library, the computer department finds the data that it prepared when the books were taken out and adds them to the day's batch. At the end of the day, this collection is processed by a program that reports on the day's returns.

    The library in the city of Cranky Harold wants to do the same thing, but East Flapville would not give, rent, or sell it the program. However, enough information has been gathered so that the program's requirements can be defined. We know that each line contains information about a book and its borrower:

    (1) The borrower's name (up to 20 characters)
    (2) The borrower's card number (7 digits).
    (3) The book's call number ( a number whose integer portion can be anything from 0–999 and whose fractional part may have no less than one and no more than six decimal places).
    (4) The month in which the book is due.
    (5) The day of the month on which the book is due:
    (6) The year in which the book is due.

    A typical line looks like this:

```
'SMEDLEY, HEDLEY K. '     026414     817.4167     6     30     1981
```

    Each day, the day's data are arranged so that all data from the same borrower are together. Then, a line is placed at the front with that day's date (month, day, year) and a special (fake) input line is placed at the back with a borrower's number of 0000000 on it. The program produces the following output:

    (1) The program separates the information for each borrower by leaving two blank lines between borrowers and starting the little section for each borrower by printing a line with the borrower's name and number.
    (2) For each line (i.e., each book returned by a particular borrower), the program prints a line showing the call number, the number of overdue days (if any), and the amount (if any) of overdue charges.
    (3) After all of a borrower's data have been processed, the program leaves a blank line and prints a summary line showing the number of books returned by that borrower and the total amount of overdue charges. Then it leaves the two blank lines mentioned before and goes on to the next borrower (if there is one).
    (4) After the last borrower's data have been processed, the program prints a summary showing (on separate lines) the number of borrowers, number of books returned, and the total amount of overdue charges collected that day.

    The city of Cranky Harold allows borrowers to take out as many books as they can carry, and overdue charges are determined as follows:

    (1) Five cents per day for the first nine days.
    (2) Seven cents per day for the next five days.
    (3) Ten cents per day for every overdue day beyond that.
    (4) The Cranky Harold Library is open seven days a week.

26. Write the program described in Problem 25 with the following additional wrinkle: The Cranky Harold Library has changed its policy somewhat in that it will be closed on July 4th, Christmas Day, New Years Day, and March 9th, the birthday of the city's founder, Harold C. Shinsplint. These are free days, not to count in the computation of the overdue charges.

27. Looking to outdo East Flapville, Cranky Harold's Board of Alderpersons wants the library program expanded so that it produces a more elaborate summary. In addition to the information described in Problem 25, the expanded version is to print:

    (1) The average number of books returned by each borrower (rounded to the nearest tenth of a book).
    (2) The average overdue fine paid by a borrower (to the nearest cent).
    (3) The name of the borrower paying the largest overdue fine that day, and the amount of that fine. You are guaranteed that there never will be more than one such winner on any given day.

# 12 *Data for Arguments to Subprograms*

In looking at the mechanisms for defining and invoking subprograms, we used simple variables as arguments. This is just one of several forms such arguments may take. This chapter explores the other possibilities.

## 12.1 SINGLE DATA VALUES AS ARGUMENTS

To establish the mechanisms for defining and using subprograms, we illustrated the transfer of data to a subprogram by specifying actual arguments in the form of variables. Thus, when some function PCOMP is invoked as part of an assignment statement such as

```
FINAL = START_VAL + 0.012 * PCOMP (BASE, CHANGE, UPPER_LIMIT) **1.4
```

PCOMP computes its result using the three individual data values taken from the places named BASE, CHANGE, and UPPER_LIMIT. Note that the function itself does not "know" how the values were obtained or from where they were taken. It is this framework that enables us to expand the variety of ways in which argument values are produced.

### 12.1.1 Constants as Arguments

FORTRAN will accept a constant value as an argument. For example, suppose a subroutine named CHEMVAL is defined as follows:

```
SUBROUTINE  CHEMVAL (COEF, VISC, ACTIV, ADJ, MIX, FINAL)
IMPLICIT    NONE
REAL        COEF, VISC, ACTIV, ADJ, FINAL, REDUC
INTEGER*2   MIX
IF (VISC .LT. 0.64/ACTIV) THEN
  REDUC = 0.0
ELSE
  REDUC = 0.0183
END IF
MIX = INT( (ACTIV-ADJ) / (1.0-REDUC) )
FINAL = COEF * VISC ** (ACTIV+ADJ) - REDUC
RETURN
END
```

Now we shall set up an IF-THEN-ELSE construct that invokes this subroutine in one of two ways depending on the value of some real variable MW:

```
  .....
IF (MW .LT. 250.0) THEN
  CALL CHEMVAL (FACTOR, VISC, ENRG, REFIN, MIXVAL, OUTCOM)
ELSE
  CALL CHEMVAL (1.0, VISC, ENRG, 0.0, MIXVAL, OUTCOM)
END IF
GLBNUM = LOG (MW) * OUTCOM
  .....
```

Although the exact nature and meaning of these computations is not defined, we can make certain observations about them just by looking at the decision structure:

1. When some variable MW is less than 250.0, CHEMVAL depends on four variables and is subject to further adjustment by a local variable REDUC within the subroutine itself. The fifth and sixth arguments do not affect the results. Rather, they are variables to which the results are assigned.

2. Once MW reaches 250.0, the equation settles down in that the first and fourth arguments no longer affect the computations. Thus, instead of assigning values of 1.0 and 0.0 to FACTOR and REFIN, respectively, and using the variable names as arguments, we show these arguments directly as constants, so that the program gives a clearer indication of what is going on.

The result is that, in those cases where MW is not less than 250.0, invocation of CHEMVAL will cause the function to associate the constant 1.0 with the first dummy argument (COEF), and the constant 0.0 with the fourth dummy argument (ADJ). This is completely consistent with the framework mentioned earlier. Whenever a subprogram is invoked with single-valued arguments, FORTRAN sets up an association between each name in the dummy argument list and the value of the corresponding actual argument supplied to the subprogram.

The example given above also emphasizes the fact that the names used in the dummy argument list do not interfere with the actual arguments given to a subprogram when it is invoked. There is no confusion, for instance, even though the second actual argument happens to have the same name (VISC) as the second member of the dummy argument list. The dummy argument name VISC is strictly local to the subroutine and has no existence outside the subroutine. When the *actual argument* VISC is specified in the invocation, the value currently stored in VISC is transmitted to the subroutine and used there. Consequently, the subroutine does not "know" that the value came from a variable named VISC. This separation makes it possible for somebody to develop a subprogram, using any dummy argument names he or she chooses, without having to worry about the *actual* argument names that will be specified for a given invocation.

As a result of this flexibility, there is an entire industry (and not a little one, at that) built around the design, development, and sale of subprograms. This commerce goes on successfully, even though the seller has no idea when, where, and with what argument names the subprogram will be used.

FORTRAN places no particular restrictions on where and when to use constants as actual arguments. Therefore, it is up to the programmer to make sure that their use makes sense. The guideline is a simple one. Since a constant is meant to be a value not subject to change, it should be used in those instances where the subprogram's computations do not alter it. Applying this guideline to the previous example, we see that it would not make sense to specify a constant value as either the fifth or sixth actual argument. Values produced by the subroutine are transmitted to these arguments rather than being taken from them.

### 12.1.2 Expressions as Arguments

Chapter 5 discusses the idea that a constant and a single-valued variable are themselves simple forms of expressions. Therefore, FORTRAN's ability to accept expressions as actual arguments is just a continuation of the normal process for transferring information when a subprogram is invoked. Thus, when a subprogram specifies a dummy argument of a particular type, the actual argument can be any legitimate expression whose result is a value of the same type.

For example, the subroutine CHEMVAL used in the previous section required six arguments, four of which were not changed as a result of the subroutine's activities.

Consequently, we can invoke CHEMVAL using an expression for any or all of those first four arguments. For instance, the following invocation

```
      CALL CHEMVAL (FACTOR-0.0082, VISC,
     1                          SQRT(1.0-ENRG)/MW, REFIN, MIXVAL, FINVAL)
```

is perfectly acceptable. When the invocation occurs, FORTRAN evaluates each expression and associates the result with the corresponding entry in the dummy argument list, as it does in any other invocation. In this case, the association is as

| *dummy argument* | *corresponding actual argument* |
|---|---|
| COEF | result of (FACTOR-0.0082) |
| VISC | VISC |
| ACTIV | result of (SQRT((1.0-ENRG)/MW)) |
| ADJ | REFIN |
| MIX | MIXVAL |
| FINAL | FINVAL |

There are two basic restrictions when it comes to using expressions as actual arguments. Both have been mentioned in passing, but it will be helpful to reemphasize them:

1. The expression, when evaluated, must produce a result having the same data type as the corresponding dummy argument.
2. An expression other than a single-valued variable or array element by itself *cannot* be used for an argument whose value will be changed by the subprogram.

The second restriction applied specifically to subroutines where the value (or values) produced by the subroutine are delivered in arguments designated for that purpose. Going back to the previous example, then, it would be inconsistent (and illegal) to use expressions for the fifth and sixth arguments of CHEMVAL.

A special note should be made about character string arguments. Since it is legal to specify a character string expression as an actual argument (assuming, of course, that the corresponding dummy argument is declared as CHARACTER), evaluation of such an expression could produce a string having a length that is different from that given in the subprogram's declaration. FORTRAN will accept the string as long as it is *not less than* the length given for the dummy argument. (The same holds true for any single-valued character argument.) If the actual argument is longer than the dummy argument, the string still will be accepted; however, only part of it will be used. For instance, if a 10-character string is given to a subprogram whose dummy argument specifies a length of 7, only the 7 leftmost characters of the actual string value will be supplied.

### 12.1.3 Array Elements as Arguments

Since each element of an array can be isolated (by means of subscripts) and used just like a single-valued variable, its value can be made available as an argument to a subprogram just as conveniently as any other single value. Thus, assuming everything has been properly declared, we can invoke good old CHEMVAL with a statement such as

```
CALL CHEMVAL (FACTOR-0.0082, VISCO(3), ENRG, 0.0, MIXERS(4), OUTCOM)
```

Note that two of the actual arguments are array elements. As a result of this particular CALL, CHEMVAL will use values obtained from the computation of FACTOR-0.0082, the third element in the array VISCO, the single-valued variable ENRG, the constant 0.0 (and the local variable REDUC) to compute two results. One of these (the integer value represented by the dummy argument MIXV) will be delivered to the fourth element of array MIXERS, and the other to the single-valued variable OUTCOM.

### 12.1.4 Character Arguments with Adjustable Lengths

Since each character in a string can be manipulated separately, there can be some additional flexibility with regard to actual character arguments. Specifically, it is not necessary for the string length in the actual argument to be the same as that in the dummy argument. Different lengths are handled by specifying an asterisk (*) as the length of the dummy character argument. Then, when the subprogram is invoked, it uses the length declared for the actual character argument. This length does not itself have to be specified as an argument; it is determined from the declaration in the invoking program.

For example, suppose subroutine HDG uses a character string as a label of some kind, and its statements include the following:

```
SUBROUTINE  HDG (HVAL, LVAL, NUMVAL, LABEL)
REAL        HVAL, LVAL
INTEGER*2   NUMVAL
CHARACTER   LABEL*(*)
   .......
   .......
PRINT *, LABEL
   .......
RETURN
END
```

Then, for instance, some variable HEAD declared in the invoking program as

```
CHARACTER*20    HEAD
```

can be used as an actual argument in a statement such as

```
CALL HDG (VMAX, VMIN, NV, HEAD)
```

and, for this invocation, HDG will print 20 characters.

Sometimes the length needs to be specified as an argument, but not because FORTRAN does not "know" what it is. Rather, there may be an occasion where it must be referred to explicitly. A case like this is seen in the next example.

**Example 12.1** We shall design a function RVRS which takes a character string of any length and returns a string with the characters in reverse order. Since each character has to be moved separately, the loop that does this needs a limiting value, i.e., the string length. Consequently, the second argument will be the string length. The processing itself is quite simple, as Figure 12.1 shows.

```
CHARACTER*(*) FUNCTION      RVRS (STRING,NCHRS)
INTEGER*2                   NCHRS,I,J
CHARACTER                   STRING*(*)

DO I = 1,NCHRS,1
  J = NCHRS+1-I
  RVRS(J:J) = STRING(I:I)
END DO

RETURN
END
```

**FIGURE 12.1** Character Reversal Function.

An invoking program, then, could use RVRS as follows:

```
  .......
CHARACTER  HDLIN1*25, HDLIN2*30
  .......
  .......
HDLIN1 = RVRS(HDLIN1, LEN(HDLIN1))
  .......
  .......
HDLIN2 = RVRS(HDLIN2, LEN(HDLIN2))
  .......
```

## 12.2 ARRAYS AS ARGUMENTS

Functions and subroutines can be designed to process entire arrays. Association between an actual array argument and the corresponding entry in a dummy argument list follows the same pattern described for single-valued arguments. The array name given in a dummy argument list is defined as an array in a declaration statement within the subprogram, and a similar declaration appears in the invoking program for the array name to be used as an actual argument.

As is the case with single-valued arguments, it still is necessary to make sure that the declared array type is the same for both array names. For instance, if a main program has the following statements:

```
INTEGER*2    POINTS(24)
  .....
  .....
CALL STATS (POINTS)
  .....
```

the subroutine itself must include a similar declaration:

```
SUBROUTINE  STATS (SCORES)
INTEGER*2   SCORES (24)
  .....
  .....
RETURN
END
```

If the subroutine is designed to process an array of character strings, the declared lengths must match as well.

**Example 12.2** To illustrate the use of subprograms for array processing, we shall design a program in which each set of input data consists of a four-digit run number followed by a list of 40 words having various lengths, none of which is greater than ten letters. Each word is left justified, i.e., it starts in its leftmost position. A run number of zero ends the run. After printing an input array (four words per line), the program is to determine the number of one-letter words, two-letter words, and so on. These results are to be printed as a two-column frequency table in which the first column shows the word length and the second column shows the number of words having that length. Finally, the program is to print the list of words rearranged so that they are in order by increasing length (shortest length first). When there are several words with the same length (as there are bound to be), the order within that group does not matter. This final list is to be printed four words per line.

Since the problem requires us to rearrange the words in order of increasing length, it will be necessary to store all of the words for a given input set in an array (which we shall call WORDS). Moreover, we shall need to build an additional array of 40 integers (WORD_SIZE) in which the lengths will be stored. When we perform the sorting operation, we shall be sorting WORD_SIZE rather than WORDS. After all, it is WORD_SIZE that will contain the information on which the rearrangement is to be based. Then, every

time we need to reposition an element in WORD_SIZE, we shall automatically reposition the corresponding element from WORDS in the same way, thereby forcing the ordering of the words to imitate the ordering of their lengths. The need to declare character strings with fixed lengths leads to the use of the maximum possible length (10) for the elements in WORDS. Then, once the array has been read in, the program can look at each word and determine its length. This will be done by using the INDEX built-in (intrinsic) function to locate the position of the first blank in the word. The word length, then, will be one less than the position at which that blank was found.

If the search for a blank was unsuccessful, we know that the length of that particular word is 10. (We could be cute and declare the length of the elements in WORDS as 11, thereby guaranteeing that every element will have at least one blank in it. However, cuteness is not the current subject.) The resulting length then is used to determine which of the counters to increase. These counters, each of which keeps track of the number of words having a particular length, are stored in a ten-element integer array called FREQ. Printing of the output is straightforward enough, so that it need not be discussed.

Now that the basic processing has been outlined, we can turn our attention to the major focus, which is the program's overall organization. It is important to note that there is no one "correct" way to organize a processing task into a main program and supporting subprograms. Instead, we take advantage of the lessons learned from successful programming projects which, together with a helping of common sense, make it possible to identify some guidelines that can help determine how big a subprogram "ought" to be and how much processing "should" be included in it:

1. A subprogram ought to perform *one job.* The nature of that job should be clearcut, so that there is no doubt about what it is. Some people insist that the job should be sufficiently small and elementary so that a person can describe it in a *single, relatively simple sentence.* This does not mean that the job cannot be complex; it *does* mean, however, that the processing clearly represents one step in a solution procedure.

2. Regardless of its complexity, the job handled by a subprogram should be small enough to be understood completely by a person looking at the statements of that subprogram. For many practitioners this means that, *in general,* if a subprogram's length is such that a printout of its statements (i.e., a *listing*) requires more than a single page, the function or subroutine is too long. The designer, then, would be well advised to reexamine the subprogram and see whether its processing can be restated in terms of two (or more smaller processes, each of which then would become a separate subprogram.

Applying these guidelines to our example, we identify the following subprograms as being appropriate in the light of the guidelines given before:

1. A subprogram named WORD_LENGTH that examines a word and determines its length. Since only a single result is produced (the word being examined is not changed in any way), WORD_LENGTH will be constructed as a function.

2. A subprogram named SORT_LIST that sorts the array WORD_SIZE by word length. As part of this process, it also will sort WORDS, the array we actually want rearranged. SORT_LIST, obviously, will be a subroutine since the values in any (or all) of the two arrays' elements may change because of the rearrangement.

3. A subprogram named WRITE_WORDS that prints the word list. This will be a subroutine, not because it operates on an entire array, but because it does not change any values at all, nor does it produce any new ones. Recall that a function subprogram, by definition, delivers *exactly one result.* WRITE_WORDS will be invoked twice for each word list: once to print the list in its original order, and a second time to print the sorted list. Both times the name of the actual argument will be the same.

4. A subprogram named WRITE_TABLE to print the frequency table (FREQ). This will be a subroutine for the same reason as described for WRITE_WORDS.

5. A subprogram named INIT that resets the system for a new list of words. Basically, this subroutine (the reason why it must be a subroutine is apparent by now) sets the ten counters in FREQ back to zero.

The identification of these subprograms and their respective duties brings to light an important factor. Even though we have not yet defined the processing details, the fact that we know where one subprogram's job ends and another begins enables us to design the main program. In effect, we have

organized the program's structure so that the main program becomes an *outline of the processing activities.* As seen in the flowchart and pseudocode of Figure 12.2, it is little more than a loop that manages the flow of events and invokes the proper subprogram at the proper time. The only processing that the main program itself does (aside from handling the loop) is the data input, printing of headings, insertion of each new word length, and the addition to the frequency counter. These activities were considered to be too simple to warrant construction of separate subprograms. It may be argued that the same holds true for some of the others, but we must bear in mind that this process of subdividing an algorithm into unit processes is largely a matter of judgment which improves with experience. Also, remember that this is an example.

The simplicity of the main program makes it possible for the statements themselves to be an immediate carryover from the flowchart or pseudocode. This is seen by examining the listing in Figure 12.3.

Because the labor has been divided as indicated above, each of the subprograms is simple. Note that wherever a subprogram needs an extra variable to play some part in its operations, that variable is declared locally, thereby keeping it within that subprogram. This relieves the main program of unnecessary clutter. For instance, in performing the sorting operation, `SORT_LIST` makes use of temporary storage places while it swaps pairs of elements. These are set up as local variables and are not part of the dummy argument list. The entire program is shown in Figure 12.4 and a sample run is seen in Figure 12.5.

### 12.2.1 Size Differences Between Actual and Dummy Array Arguments

It is not always necessary for the size of an actual array argument to match that declared for the corresponding dummy array. FORTRAN will accept an actual array argument *at least as large* as the corresponding dummy array. In the main program of Example 12.2, for instance, the array `WORDS` could have been declared with more elements, but it could not have been declared with fewer than (40) without changing the declarations in `INIT`, `SORT_LIST`, and `WRITE_WORDS`. (`WORD_LENGTH`, of course, would require no change since it deals only with a single element at a time.)

### 12.2.2 Adjustable Array Sizes

Another, more flexible way of using subprograms with actual array arguments of different sizes is to construct the subprogram with a dummy array argument having *adjustable*

```
"Define FREQ(10), WDSIZE(10), WORDS(40), LNUM, RUNNUM."
"Read the first run number."
while RUNNUM is not equal to zero:
    "Initialize the counters."
    "Print the heading for this run."
    "Read a set of 40 input words."
    do for each of the 40 input words:
        "Determine the word length."
        "Increment the appropriate frequency counter."
        "Store the word length."
    enddo
    "Print the input list."
    "Sort the input list by increasing word size."
    "Print the frequency table."
    "Print the sorted word list."
    "Read the next run number."
endwhile
"Print a terminating message."
"Stop."
```

**FIGURE 12.2 (a)** Pseudocode for Example 12.2.

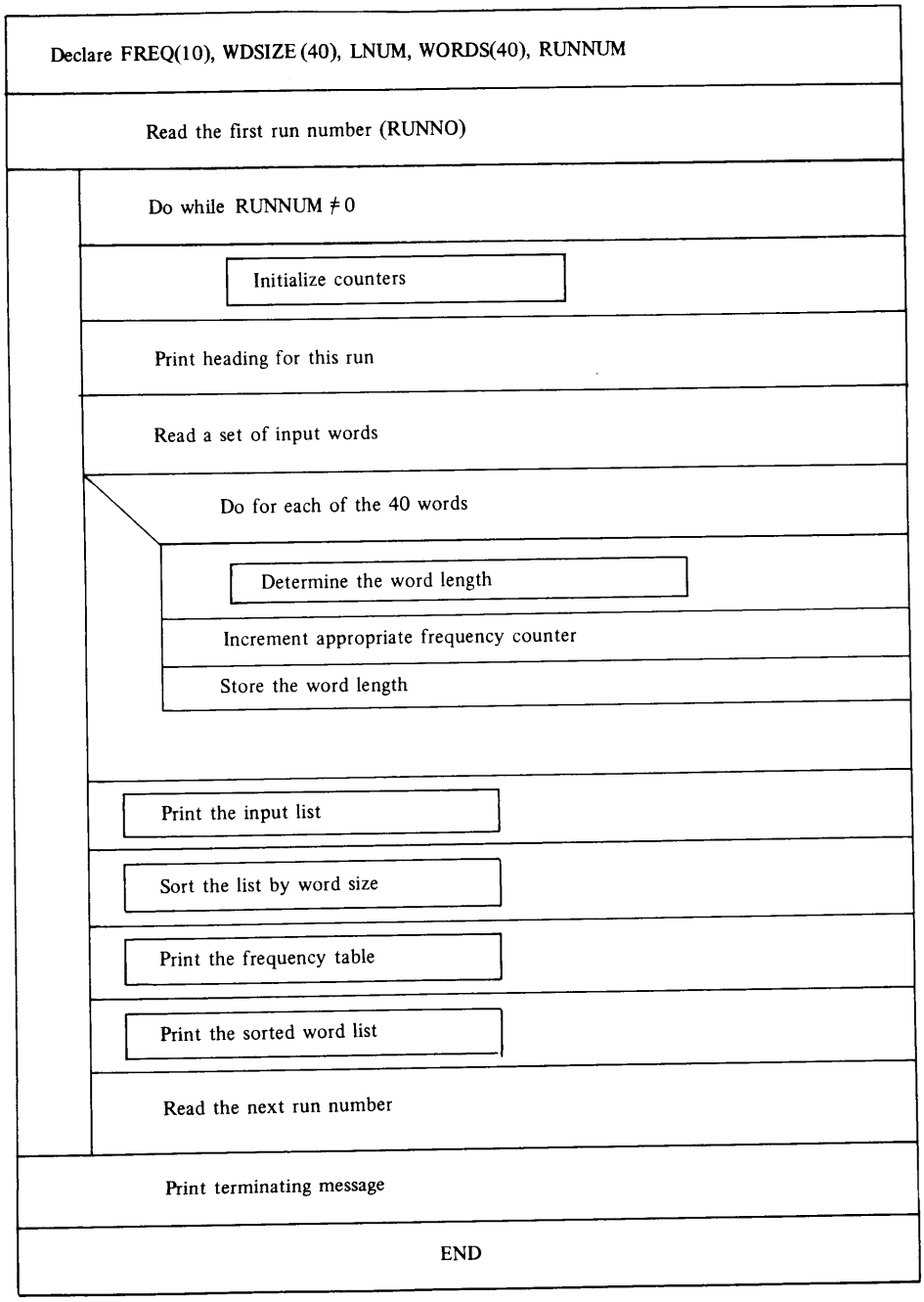



**FIGURE 12.2 (b)** N-S Diagram for Example 12.2.

```
C*************************************************************
C                    EXAMPLE 12.2                             **
C  THIS PROGRAM SORTS A LIST OF 40 WORDS IN ORDER BY          **
C  INCREASING WORD LENGTH AND COMPUTES THE NUMBER OF          **
C  OCCURRENCES OF EACH WORD LENGTH. MAXIMUM WORD              **
C  LENGTH IS TEN LETTERS.                                     **
C*************************************************************
C                   THE MAIN PROGRAM                          **
C*************************************************************
C  WORDS:   A 40-ELEMENT ARRAY FOR THE WORD LIST              **
C  WDSIZE:  A 40-ELEMENT ARRAY FOR THE WORD LENGTHS           **
C  FREQ:    A 10-ELEMENT ARRAY FOR THE COUNTERS               **
C  LNUM:    THE LENGTH OF A PARTICULAR WORD                   **
C  RUNNUM:  THE 4-DIGIT RUN NUMBER                            **
C*************************************************************
      PROGRAM       WRDCNT
      IMPLICIT      NONE
      INTEGER*2     FREQ (10), WDSIZE(40), LNUM, I, RUNNUM
      CHARACTER*10  WORDS(40),BLANKS*7
      PARAMETER     (BLANKS = '       ')

      WRITE (1,*) 'ENTER THE RUN NUMBER'
      READ (1,*) RUNNUM

      DO WHILE (RUNNUM .NE. 0)
         CALL INIT (FREQ)
         PRINT *, BLANKS
         PRINT *, 'RUN NUMBER ',RUNNUM
         WRITE (1,*) 'ENTER THE WORD LIST FOR THIS RUN'
         READ (1,*) WORDS
         DO I=1,40
           LNUM = WORD_LENGTH (WORDS(I))
           WDSIZE(I) = LNUM
           FREQ(LNUM) = FREQ(LNUM) + 1
         END DO
         PRINT *, 'LIST OF WORDS AS RECEIVED:'
         PRINT *, BLANKS
         CALL WRITE_WORDS(WORDS)
         CALL SORT_LIST(WORDS,WORD_SIZE)
         PRINT *, BLANKS
         CALL WRITE_TABLE(FREQ)
         PRINT *, BLANKS
         PRINT *, 'LIST OF WORDS BY INCREASING LENGTH:'
         CALL WRITE_WORDS(WORDS)
         WRITE (1,*) 'ENTER THE NEXT RUN NUMBER'
         READ (1,*,END=199) RUNNUM
      END DO

  199 PRINT *, BLANKS
      PRINT *, 'END OF RUN. NORMAL TERMINATION.'
      STOP
      END
```

**FIGURE 12.3** FORTRAN Statements for Example 12.2's Main Program.

```
C*******************************************************
C                      INIT                             **
C  THIS SIMPLE SUBROUTINE MERELY SETS THE TEN           **
C  COUNTERS IN THE ARRAY FREQ TO ZERO.                  **
C*******************************************************
      SUBROUTINE      INIT (CTS)
      INTEGER*2       CTS(10), J
      DO J=1,10
        CTS(J) = 0
      END DO
      RETURN
      END
```

**(a)**

```
C*******************************************************
C                    WORD_LENGTH                        **
C  THIS FUNCTION OPERATES ON A SINGLE CHARACTER STRING **
C  OF LENGTH 10 CONTAINING A 1-10 LETTER WORD WITH THE **
C  UNUSED POSITIONS (AT THE RIGHT) FILLED WITH BLANKS. **
C  WORD_LENGTH RETURNS THE WORD'S LENGTH.              **
C*******************************************************
      INTEGER*2 FUNCTION     WORD_LENGTH (STRING)
      CHARACTER*10          STRING
      WORD_LENGTH = INDEX(STRING,' ')
      IF (WORD_LENGTH .EQ. 0) THEN
        WORD_LENGTH = 10
      ELSE
        WORD_LENGTH = WORD_LENGTH-1
      END IF
      RETURN
      END
```

**(b)**

```
C*******************************************************
C                    SORT_LIST                          **
C  THIS SUBROUTINE SORTS A LIST OF WORDS IN ORDER OF    **
C  INCREASING WORD LENGTH. THERE IS NO ATTEMPT TO       **
C  SORT WITHIN A GROUP OF WORDS HAVING A GIVEN LENGTH.  **
C  TO DO THIS, SORT_LIST ACTUALLY SORTS THE WORD        **
C  LENGTHS AND MOVES THE WORDS THEMSELVES ACCORDING TO  **
C  THE WAY THE LENGTHS ARE REARRANGED.                  **
C*******************************************************
      SUBROUTINE      SORT_LIST (WDS,SIZES)
      INTEGER*2       SIZES(40),TMPSIZ,K1,K2
      CHARACTER*10    WDS(40),TMPWRD
      DO K1=100,2,-1
        DO K2=1,K1-1,1
          IF (SIZES(K2) .GT. SIZES(K2+1)) THEN
            TMPSIZ = SIZES(K2+1)
            TMPWRD = WDS(K2+1)
            SIZES(K2+1) = SIZES(K2)
            WDS(K2+1) = WDS(K2)
            SIZES(K2) = TMPSIZ
            WDS(K2) = TMPWRD
          ELSE
          END IF
        END DO
      END DO
      RETURN
      END
```

**(c)**

*(continued)*

**FIGURE12.4** **(a)** INIT Subprogram for Example 12.2. **(b)** WDLGTH Subprogram for Example 12.2. **(c)** SRTLST Subprogram for Example 12.2.

```
C*************************************************************
C                      WRITE_WORDS                           **
C  THIS SUBROUTINE PRINTS A 40-ELEMENT ARRAY OF              **
C  CHARACTER STRINGS FOUR TO A LINE.                         **
C*************************************************************
      SUBROUTINE       WRITE_WORDS (STR)
      INTEGER*2        J
      CHARACTER*10     STR(40)
      DO J=1,40,4
        PRINT *, STR(J), STR(J+1), STR(J+2), STR(J+3)
      END DO
      RETURN
      END
```

**(d)**

```
C*************************************************************
C                      WRITE_TABLE                           **
C  THIS SUBROUTINE PRINTS THE FREQUENCY TABLE                **
C*************************************************************
      SUBROUTINE       WRITE_TABLE (TAB)
      INTEGER*2        TAB (10), K
      PRINT *, 'WORD LENGTH','NO. OF OCCURRENCES'
      DO K=1,10,1
        PRINT *, K,TAB(K)
      END DO
      RETURN
      END
```

**(e)**

**FIGURE 12.4** **(d)** WRTWDS Subprogram for Example 12.2. **(e)** WRTTAB Subprogram for Example 12.2.

*dimensions*. This is done by supplying the subprogram with an additional argument that specifies how large the actual array is for that particular invocation. Correspondingly, the subprogram includes an integer argument that appears in the dummy array declaration.

**Example 12.3** In order to look at a particular case, let us suppose that our computer installation requires a general purpose subroutine which takes a list of real numbers and computes their sum, average value, and range (the difference between the largest and smallest values). This subroutine, named MNVAL, eventually will be incorporated into a variety of other programs, each of which may specify an array of a different size.

Accordingly, we shall design MNVAL to expect the following arguments:

1. The real array itself (named ARRAY in the dummy list).
2. The array size (an integer named NUMOBS in the dummy list).
3. The real argument for the array sum (ARYSUM).
4. The real argument in which the average is to be delivered (AVG).
5. The argument in which the range value is to be placed (RANGE).

The subroutine is shown in Figure 12.6. Note that the variables TMPMAX and TMPMIN, used in developing the range, are local to MNVAL since the specifications require only their difference, and not their values.

FORTRAN's rules still require that the actual array argument be declared (in the invoking program) with a *fixed* set of dimensions. However, each program using MNVAL can specify the array size, whatever it may be, as the second actual argument. As far as the subroutine itself is concerned, that argument (NUMOBS in the dummy argument list) is a single-valued integer. This means it can be specified as a constant, variable, or any integer arithmetic expression. Thus, one way of calling MNVAL could be

```
RUN NUMBER 27
LIST OF WORDS AS RECEIVED

COWS            BEARD           BROODS          SCEPTER
ANY             GRASSLAND       BOWLS           TO
GROWTH          TARTNESS        BUILDUP         ELEVATORS
CARDBOARD       SUBROUTINE      ENLARGE         AN
HISTORY         WASHES          OPAQUE          FARTHEST
GOATS           COATS           BOATS           MOATS
TWELVE          SIXES           DUCTILE         MOBILE
EXIT            BE              ENTRANCE        WAY
MAINSTREAM      I               BROKERAGE       SENSITIVE
SHIELDS         OVERTURE        AVERAGE         MALLEABLE

WORD LENGTH          NO. OF OCCURRENCES
1                    1
2                    3
3                    2
4                    2
5                    7
6                    6
7                    7
8                    4
9                    6
10                   2

LIST OF WORDS BY INCREASING LENGTH

I               AN              TO              BE
WAY             ANY             EXIT            COWS
MOATS           BOATS           BOWLS           SIXES
COATS           BEARD           GOATS           MOBILE
OPAQUE          BROODS          WASHES          TWELVE
GROWTH          SCEPTER         AVERAGE         DUCTILE
ENLARGE          BUILDUP         SHIELDS         HISTORY
FARTHEST        ENTRANCE        OVERTURE        TARTNESS
MALLEABLE       SENSITIVE       ELEVATORS       BROKERAGE
GRASSLAND       CARDBOARD       SUBROUTINE      MAINSTREAM

END OF RUN. NORMAL TERMINATION.
```

**FIGURE 12.5** Sample Run for Example 12.2.

represented by the following sequence of statements:

```
PROGRAM          CALLER
REAL             RDNGS(120), TOTAL, AVGRDG, SPAN
  .......
READ (1,*) RDNGS
  .......
  .......
CALL MNVAL (RDNGS, 100, TOTAL, AVGRDG, SPAN)
  .......
  .......
```

```
      SUBROUTINE        MNVAL (ARRAY,NUMOBS,ARYSUM,AVG,RANGE)
      INTEGER*2         NUMOBS,I
      REAL              ARRAY(NUMOBS),ARYSUM,AVG,RANGE,TMPMAX,
     1                  TMPMIN

C*************************************************************
C  TMPMAX AND TMPMIN ARE INITIALIZED TO THE VALUE OF  **
C  THE FIRST ARRAY ELEMENT SO THAT WE HAVE A REALISTIC**
C  REFERENCE. NOTE THAT THE MAX AND MIN INTRINSIC     **
C  FUNCTIONS ARE USED WITH THE TEMPORARY MAXIMUM AND  **
C  MINIMUM VALUES, COMPARING THEM WITH EACH ARRAY     **
C  ELEMENT IN TURN.                                   **
C*************************************************************

      ARYSUM = 0.0
      TMPMAX = ARRAY(1)
      TMPMIN = ARRAY(1)

      DO I=1,NUMOBS
        ARYSUM = ARYSUM+ARRAY(I)
        TMPMAX = MAX(TMPMAX,ARRAY(I))
        TMPMIN = MIN(TMPMIN,ARRAY(I))
      END DO

      AVG = ARYSUM/NUMOBS
      RANGE = TMPMAX-TMPMIN
      RETURN
      END
```

**FIGURE 12.6** FORTRAN Statements for Subroutine MNVAL.

### 12.2.3 A Special Note About Character Arrays

Elements in an actual character array argument may have a length different from that declared for the dummy array. The only restriction is that the *total number of characters* in the dummy array cannot exceed the *total number of characters* in the actual array argument. For example, in the subroutine SORT_LIST from Example 12.2, the dummy array WDS is declared as a 40-element array of 10-character strings. This means that a CALL to SORT_LIST may specify a collection of at least 400 characters, no matter how they are organized: for instance, 80 elements 5 characters long, 20 elements 20 characters long, 25 elements 12 characters long, and 5 elements 80 characters long all are eligible, as well as many other combinations. If there are more characters in the actual array than the number specified for the dummy array, the subprogram will use as many of them (starting from the left) as the dummy array declaration allows. For example, if a calling program includes the following statements:

```
INTEGER*2        LGTHS(40)
CHARACTER*12     VOCAB(40)
  .......
CALL   SORT_LIST(VOCAB, LGTHS)
  .......
```

SORT_LIST would use VOCAB's first 400 characters: 12 each from the first 33 elements of VOCAB, and the leftmost four characters from VOCAB(34).

Alternatively, the use of asterisks for dummy character array specification is simple if the programmer merely wishes to set up associations between array elements, and not between *individual characters*. To provide SORT_LIST with this capability, for example, we would add an integer argument for the array size (say NSZ) and use an asterisk for the string length:

```
SUBROUTINE      SORT_LIST(WDS, SIZES, NSZ)
INTEGER*2       SIZES(NSZ), TMPSIZ, K1, K2
CHARACTER*(*)   WDS(NSZ), TMPWRD
      .......
      .......
```

### 12.2.4 Parts of Arrays as Arguments

When a subprogram includes one or more arrays as dummy arguments, the corresponding actual arguments need not be entire arrays. By using a simple technique based on FORTRAN's capability for handling adjustable dimensions, we can specify parts of arrays as actual arguments.

The subprogram is constructed for an adjustable array, so that the dummy argument list includes an integer variable for each adjustable dimension. Then, the subprogram uses each of those integer variables in the array's declaration. When the subprogram is invoked, the subscript attached to the *actual* array argument specifies the first element in the part of the array to be processed by this invocation. The actual integer value corresponding to the dummy argument used to define the array's size in the subprogram tells how many elements to be processed.

To take a first look at how the array argument specifications work, let us use the subprogram MNVAL designed in the previous section and shown in Figure 12.6. Recall that the second argument for that subprogram defines the array size. If we invoke MNVAL with the statement

```
CALL MNVAL (WTS, 54, WTSUM, WTAVG, WTRNG)
```

(assuming the appropriate prior declarations), MNVAL will process all 54 elements of the entire array WTS. Now, suppose array WTS still is declared (in the invoking program) with 54 elements, but our invocation looks like this:

```
CALL MNVAL (WTS(31), 20, WTSUM, WTAVG, WTRNG)
```

This time there is a subscript (31) attached to the array argument. The 31 tells FORTRAN that MNVAL is being instructed to process that part of WTS starting with element 31. The 20 indicates that 20 elements are to be processed—specifically, the 20 consecutive elements in WTS beginning with the 31st element. In response, the subprogram "pretends" that the dummy array (for this invocation) is declared with a size of 20 elements. As a result, the association set up between the dummy and the actual array arguments is as shown in Display 12.1.

**Display 12.1**

| | | | | | | |
|---|---|---|---|---|---|---|
| ARRAY (DUMMY) | ARRAY(1) | ARRAY(2) | ARRAY(3) | ... | ARRAY(19) | ARRAY(20) |
| WTS (ACTUAL) | WTS(31) | WTS(32) | WTS(33) | ... | WTS(49) | WTS(50) |

Similarly, the statement

```
CALL MNVAL (WTS(26), 14, WTSUM, WTAVG, WTRNG)
```

arranges for MNVAL to use a sequence of fourteen elements from WTS starting with element 26. One of the results, then, would be that WTSUM will contain the sum of WTS(26) through WTS(39).

Care must be taken to use this feature consistently. For example, a CALL of MNVAL with the statement

```
CALL MNVAL (WTS(40), 20, WTSUM, WTAVG, WTRNG)
```

(still assuming WTS to be declared with 54 elements) will be rejected because we would be instructing MNVAL to associate a 20-element dummy array with the elements of WTS starting at WTS(40). Since the actual argument describes only 15 elements (WTS(40) through WTS(54)), there is a mismatch.

**12.2.4.1 A General Rule for Parts of Arrays as Arguments** When a subprogram is set up to use a dummy array having *ds* elements, and we invoke that subprogram with a subscript value of *ad* attached to the actual array name, the size of the *entire actual array* must be *at least*

$$as + ds - 1$$

Thus, for the 54-element array WTS, the first call works because *ds* is 20 and *as* is 31, so that the requirement for actual size (31+20−1 or 50) is met. The same holds true for the second invocation: *ds* is 14 and *as* is 26, giving us a size requirement of 26+14−1 or 39. However, with a *ds* of 20 and an *as* value of 40, we need a minimum actual array size of 20+40−1 or 59, a requirement that the 54-element WTS does not meet.

**12.2.4.2 Another Way to Use Parts of Arrays as Arguments** We can achieve the same effect as that described in the previous section by defining the part of the array that we want in terms of another array. This is done in the invoking program by means of the EQUIVALENCE statement. Then, when the subprogram is invoked, the smaller array name is given as the argument.

For example, suppose WTS is declared as a 54-element array (as before) and we want to process 14 successive elements starting with WTS(26), just as we did before. We can pretend that the 14 elements represent an entire array by including the following declarations in the invoking program:

```
REAL          WTS(54), EWTS(14)
EQUIVALENCE  (WTS(26), EWTS(1))
```

Then, when MNVAL is CALLed, the statement would say

```
CALL MNVAL (EWTS, WTSUM, WTAVG, WTRNG)
```

so that it appears as if we are using an entire array. Note that this technique is limited to those situations in which the part of the array being used does not change in size or position.

**12.2.4.3 Parts of Multidimensional Arrays as Arguments** The same general rule can be applied to multidimensional arrays. Suppose, for example, that the subroutine MNVAL were set up to process two-dimensional arrays with adjustable dimensions. The first three statements, then, might look as follows:

```
SUBROUTINE  MNVAL (ARRAY, NROWS, NCOLS, ARYSUM, AVG, RANGE)
INTEGER*2   NROWS, NCOLS
REAL*4      ARRAY(NROWS, NCOLS), ARYSUM, AVG, RANGE
```

(We shall assume that the rest of MNVAL is adjusted appropriately to take care of the fact that the array is two-dimensional.) Now, let us say that the invoking program defines the following variables as part of its declarations:

```
REAL*4      REFRAC(5,6), RSUM, RAVG, RFRNG
INTEGER*2   RDIM, CDIM
```

Then, the following CALL to MNVAL

```
CALL MNVAL (RFRAC, 5, 6, RSUM, RAVG, RFRNG)
```

will cause MNVAL to use all 30 of RFRAC's elements in its computations. Changing the invocation to

```
CALL MNVAL (RFRAC(1,1), 3, 6, RSUM, RAVG, RFRNG)
```

sets up a 3 by 6 dummy array whose first element (ARRAY(1,1)) is associated with RFRAC(1,1), the second with RFRAC(2,1), and so on, as shown in Display 12.2.

**Display 12.2**

| | | | | | |
|---|---|---|---|---|---|
| RFRAC(1,1)<br>ARRAY(1,1) | RFRAC(1,2)<br>ARRAY(3,2) | RFRAC(1,3)<br>ARRAY(2,4) | RFRAC(1,4)<br>ARRAY(1,6) | RFRAC(1,5) | RFRAC(1,6) |
| RFRAC(2,1)<br>ARRAY(2,1) | RFRAC(2,2)<br>ARRAY(1,3) | RFRAC(2,3)<br>ARRAY(3,4) | RFRAC(2,4)<br>ARRAY(2,6) | RFRAC(2,5) | RFRAC(2,6) |
| RFRAC(3,1)<br>ARRAY(3,1) | RFRAC(3,2)<br>ARRAY(2,3) | RFRAC(3,3)<br>ARRAY(1,5) | RFRAC(3,4)<br>ARRAY(3,6) | RFRAC(3,5) | RFRAC(3,6) |
| RFRAC(4,1)<br>ARRAY(1,2) | RFRAC(4,2)<br>ARRAY(3,3) | RFRAC(4,3)<br>ARRAY(2,5) | RFRAC(4,4) | RFRAC(4,5) | RFRAC(4,6) |
| RFRAC(5,1)<br>ARRAY(2,2) | RFRAC(5,2)<br>ARRAY(1,4) | RFRAC(5,3)<br>ARRAY(3,5) | RFRAC(5,4) | RFRAC(5,5) | RFRAC(5,6) |

Consequently, this invocation will process the first three rows of RFRAC. Let us change the CALL again:

```
CALL MNVAL (RFRAC(2,1), 3, 6, RSUM, RAVG, RFRNG)
```

As shown in Display 12.3, the first element of the dummy array is associated with RFRAC(2,1):

**Display 12.3**

| | | | | | |
|---|---|---|---|---|---|
| RFRAC(1,1) | RFRAC(1,2)<br>ARRAY(2,2) | RFRAC(1,3)<br>ARRAY(1,4) | RFRAC(1,4)<br>ARRAY(3,5) | RFRAC(1,5) | RFRAC(1,6) |
| RFRAC(2,1)<br>ARRAY(1,1) | RFRAC(2,2)<br>ARRAY(3,2) | RFRAC(2,3)<br>ARRAY(2,4) | RFRAC(2,4)<br>ARRAY(1,6) | RFRAC(2,5) | RFRAC(2,6) |
| RFRAC(3,1)<br>ARRAY(2,1) | RFRAC(3,2)<br>ARRAY(1,3) | RFRAC(3,3)<br>ARRAY(3,4) | RFRAC(3,4)<br>ARRAY(2,6) | RFRAC(3,5) | RFRAC(3,6) |
| RFRAC(4,1)<br>ARRAY(3,1) | RFRAC(4,2)<br>ARRAY(2,3) | RFRAC(4,3)<br>ARRAY(1,5) | RFRAC(4,4)<br>ARRAY(3,6) | RFRAC(4,5) | RFRAC(4,6) |
| RFRAC(5,1)<br>ARRAY(1,2) | RFRAC(5,2)<br>ARRAY(3,3) | RFRAC(5,3)<br>ARRAY(2,5) | RFRAC(5,4) | RFRAC(5,5) | RFRAC(5,6) |

As a result, MNVAL still processes three rows (18 elements) of RFRAC, but they are the three rows starting with row 2 (i.e., rows 2–4).

To fasten this idea securely, we shall invoke MNVAL with the statement

```
CALL MNVAL (RFRAC(4, 1), 3, 6, RSUM, RAVG, RFRNG)
```

By now, we can see immediately that this will not work. Even though FRAC has 30 elements, and we are asking MNVAL to process only 18 elements, the association implied by our CALL starts with element RFRAC(4, 1), a point far enough along in the array so that fewer than 18 elements remain. (Specifically, only 12 elements remain.)

It is possible to set up an invocation where the actual and dummy arrays are not aligned as they are in these examples. However, the programmer must be careful to make sure he or she knows exactly how the association will take place. In any situation, the underlying guide is FORTRAN's rule for the sequence of array elements (defined in Section 7.1.3). We shall not explore the details of such associations in this text; however, one example is presented to illustrate how the rule applies. The statement

```
CALL MNVAL (RFRAC(3, 3), 3, 4, RSUM, RAVG, RFRNG)
```

sets up a 3 by 4 dummy array whose first element (ARRAY(1, 1)) is associated with element (3, 3) of array RFRAC. ARRAY's 11 remaining elements are associated, in order, with the next 11 elements of RFRAC. See Display 12.4.

**Display 12.4**

| | | | | | |
|---|---|---|---|---|---|
| RFRAC(1, 1) | RFRAC(1, 2) | RFRAC(1, 3) | RFRAC(1, 4)<br>ARRAY(1, 2) | RFRAC(1, 5)<br>ARRAY(3, 3) | RFRAC(1, 6) |
| RFRAC(2, 1) | RFRAC(2, 2) | RFRAC(2, 3) | RFRAC(2, 4)<br>ARRAY(2, 2) | RFRAC(2, 5)<br>ARRAY(1, 4) | RFRAC(2, 6) |
| RFRAC(3, 1) | RFRAC(3, 2) | RFRAC(3, 3)<br>ARRAY(1, 1) | RFRAC(3, 4)<br>ARRAY(3, 2) | RFRAC(3, 5)<br>ARRAY(2, 4) | RFRAC(3, 6) |
| RFRAC(4, 1) | RFRAC(4, 2) | RFRAC(4, 3)<br>ARRAY(2, 1) | RFRAC(4, 4)<br>ARRAY(1, 3) | RFRAC(4, 5)<br>ARRAY(3, 4) | RFRAC(4, 6) |
| RFRAC(5, 1) | RFRAC(5, 2) | RFRAC(5, 3)<br>ARRAY(3, 1) | RFRAC(5, 4)<br>ARRAY(2, 3) | RFRAC(5, 5) | RFRAC(5, 6) |

The use of the EQUIVALENCE statement to represent part of an array as an entire array (Section 12.2.4.2) applies to multidimensional arrays as well as it does to those having a single dimension.

### 12.2.5 Dimensional differences Between Array Arguments

It is possible to specify a multidimensional array as an actual argument to a subprogram in which the corresponding dummy argument is a one-dimensional array. The rules for association are the same as those discussed and illustrated previously. The only difference is that the programmer must be aware of the details, so that he or she can make sure that the processing resulting from a given invocation matches his or her requirements.

To illustrate, we shall go back to the subroutine MNVAL as it was originally specified for Example 12.2. Recall that the one-dimensional dummy array ARRAY in that example is declared with the adjustable size (NUMOBS) supplied as an actual argument by each invocation. Using the 5 by 6 array FRAC and the other variables from the previous section, we can CALL MNVAL with the statement

```
CALL MNVAL (RFRAC, 30, RSUM, RAVG, RFRNG)
```

in which case all of RFRAC's 30 elements will be processed. The resulting association between the 30-element dummy array ARRAY and the 5 by 6 actual array is shown in Display 12.5.

**Display 12.5**

| | | | | | |
|---|---|---|---|---|---|
| RFRAC(1,1)<br>ARRAY(1) | RFRAC(1,2)<br>ARRAY(6) | RFRAC(1,3)<br>ARRAY(11) | RFRAC(1,4)<br>ARRAY(16) | RFRAC(1,5)<br>ARRAY(21) | RFRAC(1,6)<br>ARRAY(26) |
| RFRAC(2,1)<br>ARRAY(2) | RFRAC(2,2)<br>ARRAY(7) | RFRAC(2,3)<br>ARRAY(12) | RFRAC(2,4)<br>ARRAY(17) | RFRAC(2,5)<br>ARRAY(22) | RFRAC(2,6)<br>ARRAY(27) |
| RFRAC(3,1)<br>ARRAY(3) | RFRAC(3,2)<br>ARRAY(8) | RFRAC(3,3)<br>ARRAY(13) | RFRAC(3,4)<br>ARRAY(18) | RFRAC(3,5)<br>ARRAY(23) | RFRAC(3,6)<br>ARRAY(28) |
| RFRAC(4,1)<br>ARRAY(4) | RFRAC(4,2)<br>ARRAY(9) | RFRAC(4,3)<br>ARRAY(14) | RFRAC(4,4)<br>ARRAY(19) | RFRAC(4,5)<br>ARRAY(24) | RFRAC(4,6)<br>ARRAY(29) |
| RFRAC(5,1)<br>ARRAY(5) | RFRAC(5,2)<br>ARRAY(10) | RFRAC(5,3)<br>ARRAY(15) | RFRAC(5,4)<br>ARRAY(20) | RFRAC(5,5)<br>ARRAY(25) | RFRAC(5,6)<br>ARRAY(30) |

This is straightforward enough: each of the 30 elements in the dummy array ARRAY represents the element in the *corresponding sequential position* of the actual array RFRAC.

Now, using the same values, we can analyze the action of the statement

```
CALL MNVAL (RFRAC(3,2),9,RSUM,RAVG,RFRNG)
```

In effect, the invocation requests MNVAL to process the nine consecutive elements in array RFRAC beginning with element RFRAC(3,2). As a result, MNVAL associates ARRAY(1) with RFRAC(3,2), ARRAY(2) with RFRAC(4,2), and so on, as shown in Display 12.6.

**Display 12.6**

| | | | | | |
|---|---|---|---|---|---|
| RFRAC(1,1) | RFRAC(1,2) | RFRAC(1,3)<br>ARRAY(4) | RFRAC(1,4)<br>ARRAY(9) | RFRAC(1,5) | RFRAC(1,6) |
| RFRAC(2,1) | RFRAC(2,2) | RFRAC(2,3)<br>ARRAY(5) | RFRAC(2,4) | RFRAC(2,5) | RFRAC(2,6) |
| RFRAC(3,1) | RFRAC(3,2)<br>ARRAY(1) | RFRAC(3,3)<br>ARRAY(6) | RFRAC(3,4) | RFRAC(3,5) | RFRAC(3,6) |
| RFRAC(4,1) | RFRAC(4,2)<br>ARRAY(2) | RFRAC(4,3)<br>ARRAY(7) | RFRAC(4,4) | RFRAC(4,5) | RFRAC(4,6) |
| RFRAC(5,1) | RFRAC(5,2)<br>ARRAY(3) | RFRAC(5,3)<br>ARRAY(8) | RFRAC(5,4) | RFRAC(5,5) | RFRAC(5,6) |

## 12.3 SUBPROGRAMS AS ARGUMENTS

A third type of argument that can be specified for an external subprogram is the name of another subprogram. this makes it possible to set up a function or subroutine that uses another subprogram whose identity need not be established until there is an actual invocation. HP FORTRAN 77 includes statement functions as part of this capability.

To provide this facility, a *dummy procedure* name is included in the list of dummy arguments. Then, when the subprogram is invoked, an actual procedure name is specified in the corresponding position of the actual argument list. Suppose we have the following

general subroutine:

```
SUBROUTINE  subnam (darg1, darg2, dsub, darg3)
REAL        darg1, darg2, darg3
   . . . . . .
   . . . . . .
darg3 = darg1 + dsub(darg1,darg2)
   . . . . . .
RETURN
END
```

The dummy argument dsub is *not* the name of a particular subprogram. Instead, it is a symbol (constructed like any other variable name) indicating that a specific subprogram name (a function in this particular example) will be supplied in the same position of the actual argument list when subnam is invoked. Accordingly, a CALL to subnam would take the form

CALL *subnam* (*arg1*, *arg2*, *sub*, *arg3*)

where *sub* is an actual function name in the same way that *arg1*, *arg2* and *arg3* are actual arguments. This may be any subprogram at all. Of course, it is up to the programmer to make sure that the actual subprogram's usage is consistent (i.e., number, type, and sequence of arguments) with that describe¢ for the dummy subprogram.

Note that dsub, though included in the dummy argument list, is not declared in the subprogram. Its meaning is established through its usage. However, it is necessary to provide FORTRAN with a list of specific subprograms that could be used as actual arguments. This list must appear in the invoking program, and there are two types of statements (EXTERNAL and INTRINSIC) provided for that purpose. We shall discuss their use in the next two sections.

### 12.3.1 The EXTERNAL Statement

If a programmer-designed external subprogram is to be used as an actual argument, its name must appear in an EXTERNAL statement in the invoking program. The general form is

EXTERNAL *sname1*, *sname2*, . . . . . .

The names may appear in any order, with functions and subroutines intermixed. This tells FORTRAN that the program or subprogram intends to use the names sname1, sname2, etc. as actual arguments, and that these arguments are subprogram names.

**Example 12.4** We shall illustrate these connections by building a set of procedures for a type of computation used in certain aspects of engineering combustion: a known flow of air coming into a burner of some kind at temperature T1 is presented with a known flow of fuel, and the resulting mixture is burned to produce a temperatue rise to some level T2. The extent of this temperature increase depends on T1 and FA, the relative amount of fuel used (e.g., pounds of fuel per pound of air). Oecause of the complex chemical and physical processes involved, several different equations have to be used to express these relations, as summarized in Table 12.1.We are to read sets of T1 and FA, producing a value of T2 for each set, along with R, the ratio of T2/T1.

To illustrate the use of procedures as arguments, we shall construct a simple main program that invokes a subroutine named COMB to obtain the final temperature. COMB uses one of two functions, HTEMP or LTEMP, to perform the computations. The selection of HTEMP or LTEMP depends on T1, and each function uses FA to determine how to do its computations. A T1 of zero concludes the run.

The only noteworthy feature of the main program (Figure 12.7) is the EXTERNAL statement in which the two functions HTEMP and LTEMP are listed. The subroutine COMB is not listed there because it is *always* invoked—it is not one of several choices. COMB uses FTEMP as a dummy procedure name. There is no subprogram named FTEMP. Instead, when COMB is called, the call will include either HTEMP or LTEMP as an actual argument.

**Table 12.1** Equations for Example 12.4
T1 = Initial Temperature, Degrees R
FA = Lbs. of Fuel per Lb. of Air
TR = Temperature Rise, Degrees R

| *T1* | *FA* | *TR* |
|---|---|---|
| <560 | <0.022 | 0.0892*T1**0.719 + 0.2744*FA |
| <560 | 0.022–0.067 | .1136*T1**1.332+(1.16*1/(560–T1))*FA**.9246 |
| <560 | >0.067 | .1136*T1**1.332+(1.16*1/(560–T1))*FA**.9246 *(FA–0.067)**1.0881 |
| >=560 | <0.0212 | .0505*(T1–560)+0.1092*T1**.757+.2861*FA –.00310FA**1.75 |
| >=560 | 0.0212–0.067 | .1163*T1**.404*(FA/(1–FA))**1.077 –.00287*FA**1.554 |

```
C*********************************************************
C                    EXAMPLE 12.4                        *
C  THIS PROGRAM COMPUTES AND PRINTS FINAL TEMPERATURES   *
C  FOR COMBUSTION PROCESSES OPERATING AT INITIAL TEM-    *
C  PERATURES T1 AND FUEL-TO-AIR RATIOS FA. FOR EACH T1   *
C  AND FA, THE PROGRAM ALSO PRODUCES R, THE RATIO OF     *
C  FINAL TEMPERATURE (T2) TO INITIAL TEMPERATURE.        *
C  THE TEMPERATURE RISE IS COMPUTED USING BLIVVOOLY'S    *
C  EQUATIONS, THE PROPER ONE DEPENDING ON T1 AND FA.     *
C*********************************************************

      PROGRAM          EX1204
      IMPLICIT         NONE
      REAL             T1,FA,T2,R,ZERO
      CHARACTER*7      BLANKS
      EXTERNAL         HTEMP,LTEMP
      PARAMETER        (ZERO=0.0,BLANKS='       ')

      PRINT *, 'T1','FA','T2','R'
      WRITE (1,*) 'ENTER VALUES FOR T1 AND FA'
      READ (1,*) T1,FA

      DO WHILE (T1 .NE. ZERO)
         IF (T1 .LT. 560.0) THEN
           CALL COMB(T1,FA,LTEMP,T2,R)
         ELSE
           CALL COMB(T1,FA,HTEMP,T2,R)
         END IF
         PRINT *, T1,FA,T2,R
         WRITE (1,*) 'READ THE NEXT SET OF VALUES FOR T1, FA'
         READ (1,*,END=199) T1,FA
      END DO

  199 PRINT *, BLANKS
      PRINT *, 'END OF RUN. NORMAL TERMINATION.'
      STOP
      END
```

*(Continued)*

**FIGURE 12.7 (a)** FORTRAN Statements for Example 12.4's Main Program.

```
C*******************************************************
C                        COMB                           *
C  THIS SUBROUTINE COMPUTES THE FINAL TEMPERATURE OF A  *
C  COMBUSTION PROCESS GIVEN THE INITIAL TEMPERATURE TA  *
C  AND THE FUEL/AIR RATIO F. IN DOING SO, IT INVOKES    *
C  ONE OF TWO FUNCTIONS, THE CHOICE BEING DETERMINED    *
C  BY THE PROGRAM CALLING COMB.                         *
C*******************************************************

      SUBROUTINE        COMB(TA,F,FTEMP,TB,RATIO)
      REAL              TA,F,TB,RATIO

      TB = TA+FTEMP(TA,F)
      RATIO = TB/TA
      RETURN
      END
```

**(b)**

```
C*******************************************************
C                      LTEMP                            *
C  THIS FUNCTION COMPUTES THE FINAL TEMPERATURE FOR     *
C  COMBUSTION PROCESSING IN WHICH THE INITIAL TEMPERA-  *
C  TURE IS BELOW 560 DEGREES RANKINE. THE FAMOUS AND    *
C  INTERESTING BLIVVOOLY EQUATIONS ARE USED.            *
C*******************************************************

      REAL FUNCTION     LTEMP(TIN,FL)
      REAL              TIN,FL
      IF (FL .LT. 0.0022) THEN
        LTEMP=0.0892*T1**0.719+0.2744*FL
      ELSE IF (FL .LT. 0.067) THEN
            LTEMP=0.1136*T1**1.332+(1.16*T1/(560.0-T1))*FL**0.9246
          ELSE
            LTEMP=0.1136*T1**1.332+(1.16*T1/(560.0-T1))*FL**0.9246
     1                          *(FL-0.067)**1.0881
          END IF
      END IF
      RETURN
      END
```

**(c)**

```
C*******************************************************
C                       HTEMP                           *
C  THIS FUNCTION COMPUTES A FINAL COMBUSTION TEMPERA-   *
C  TURE GIVEN AN INITIAL TEMPERATURE AND FUEL/AIR RATIO*
C  FOR PROCESSES IN WHICH THE INITIAL TEMPERATURE IS AT*
C  LEAST 560 DEGREES RANKINE. THE ILLUSTRIOUS AND       *
C  IMPRESSIVE BLIVVOOLY EQUATIONS ARE USED.             *
C*******************************************************

      REAL FUNCTION         HTEMP(TSTR,FLR)
      REAL                  TSTR,FLR

      IF (FLR .LT. 0.0212) THEN
        HTEMP=0.0505*(TSTR-560.0)+0.1092*T1**0.757+0.2861+FLR
     1                          -0.00310*FLR**1.75
      ELSE IF (FLR .LT. 0.067) THEN
            HTEMP=0.1163*T1**0.404*(FLR/1.0-FLR))**1.164
          ELSE
            HTEMP=(1.1083*T1/(T1-560.0))**0.874 *
     1            (FLR/(1.0-FLR))**1.077-0.00287*FLR**1.554
          END IF
      END IF
      RETURN
      END
```

**(d)**

**FIGURE 12.7** **(b)** COMB Subprogram for Example 12.4. **(c)** LTEMP Subprogram for Example 12.4 **(d)** HTEMP Subprogram for Example 12.4.

### 12.3.2 The INTRINSIC Statement

When the subprogram to be used as an actual argument is one of FORTRAN's built-in functions (Chapter 6), its name must appear in an INTRINSIC statement in the invoking program or subprogram. For example, the statement

```
INTRINSIC   SQRT, SIN, COS
```

announces that the program containing this statement intends to invoke one or more subprograms for which SQRT, SIN, and COS are to be actual arguments. The rules regarding the use of this statement are straightforward:

1. Built-in (intrinsic) functions, when used as arguments, must appear in an INTRINSIC statement in the invoking program. Such a name can appear in an EXTERNAL statement instead, but when it does, it means that a *programmer-designed* subprogram by that name will be used as an argument. As a result, the *intrinsic* function with that name is unavailable to that program or subprogram.
2. The INTRINSIC statement is reserved strictly for built-in function names; a programmer-designed subprogram must not be listed there.
3. The following intrinsic functions *cannot* be used as actual arguments and, therefore, may not appear in an INTRINSIC statement: INT, REAL, DBLE, CMPLX, CHAR, ICHAR, MAX, and MIN.

## 12.4 ARGUMENTS IN NESTED INVOCATIONS

Associations set up between dummy arguments and actual arguments can be used to transmit data values through several levels of invocation. A case of this transmission appears in Example 12.4. In that construction the main program invokes the subroutine COMB with T1 as one of the actual arguments. COMB, in turn, is set up to invoke some function (either LTEMP or HTEMP). In either event, one of the arguments specified by COMB is TA, a member of its dummy argument list. Since TA is associated with T1 when COMB is CALLed, the association extends to COMB's invocation so that, ultimately, T1's value is associated with TIN (if LTEMP is invoked) or TSTR (if HTEMP is invoked), and it is this value that is used in computing the final temperature.

This ability to transmit dummy arguments can be used regardless of the number of levels of invocation involved. However it is used, the programmer must make sure that, in some way, a value has been assigned to that name by the time it is used in an invocation.

## 12.5 SUMMARY

There are three types of information that can be transmitted to an invoked subprogram:

1. An individual data value of any type. This value may be specified as a constant, a single-valued variable, an array element, or a legitimate FORTRAN expression. When the invocation occurs, this value is associated with the name in the corresponding position of the invoked subprogram's dummy argument list. Then, that value is used in the subprogram whenever there is a reference to that dummy argument name.

2. All or part of an array. When a subprogram is invoked with an array argument, the actual array's address is associated with the name of the corresponding dummy array so that, ultimately, FORTRAN develops an association between each element of the dummy array and a particular element of the actual array. The number of elements described by the dummy array cannot exceed the number specified by the actual array argument.

3. An external function or subprogram name. When such a name is given as an actual argument, the invoked subprogram associates that name with a corresponding dummy procedure name. Then, when the subprogram executes, it, in turn, will use that association to invoke another subprogram. Built-in functions to be used as arguments must be listed in an EXTERNAL statement in the invoking program.

## PROBLEMS

1. Here is a program consisting of a main program and a single subroutine:

```
      PROGRAM      C12P1
      IMPLICIT     NONE
      REAL         X,Y,Z,A1,A2,A3,R(3),B(3,2),U,V,W
      INTEGER*     N,S1,S2,T
      CHARACTER    C1*1,C2*4,R1*1
      DATA  R,N,S1,C1,C2/1.5,2.2,2.5,2.1,'8','1202'/
      X = 2.0
      Y = 4.5
      Z = 3.0

   10 statement
   20 statement

      END

      SUBROUTINE   COMP3 (A1,A2,A3,A4,R1,R2,R3)
        REAL       A1,A2,A3,A,B,C,R1,R2,R3
      INTEGER*2    A4

      A = MIN(A1,A2,A3)
      C = MAX(A1,A2,A3)
      IF (A2 .GE. A .AND. A2 .LE. C) THEN
        B = A2
      ELSE IF (A1 .GE. A .AND. A1 .LE. C) THEN
            B = A1
         ELSE
            B = A3
      END IF
      R1 = A**(A4+1)
      R2 = B**A4
      R3 = C**(A4-1)
      RETURN
      END
```

Note that the main program has two statements (labeled 10 and 20) whose descriptions are not given. A number of possible descriptions are shown below. Based only on the FORTRAN statements given above, determine which of the CALL statements have errors in them and what the errors are. For each of the statements that will work, show the output that would be produced by the PRINT statement that follows the CALL:

(a)
```
10 CALL COMP3 (X,Y,Z,N,U,V,W)
20 PRINT *, U,V,W.
```
(b)
```
10 CALL COMP3 (X,Y,N,Z,U,V,W)
20 PRINT *, U,V,W
```
(c)
```
10 CALL COMP3 (X,Y,Z,U,V,W)
20 PRINT *, U,V,W
```
(d)
```
10 CALL COMP3 (Y,X,Z,S1,V,U,W)
20 PRINT *, U,V,W
```
(e)
```
10 CALL COMP3 (X,Y,Z,S1,U,W,V)
20 PRINT *, U,V,W
```
(f)
```
10 CALL COMP3 (R(1),R(2),R(3),N,V,W,U)
20 PRINT *, U,V,W.
```
(g)
```
10 CALL COMP3 (X,R(1),Y,N,V,W,U)
20 PRINT *, U,V,W
```
(h)
```
10 CALL COMP3 (R,S1,U,V,W)
20 PRINT *, U,V,W
```
(i)
```
10 CALL COMP3 (X,Z,R(1),N,Y,V,W)
20 PRINT *, V,W,Y
```

(j)
```
10 CALL COMP3 (X, Z, R(1), N, U, B(1, 2), W1)
20 PRINT *, U, B(1, 2), W)
```
(k)
```
10 CALL COMP3 (X, Y, C1, S1, U, Z, W)
20 PRINT *, U, Z, W
```
(l)
```
10 CALL COMP3 (X, Y, 4.0, 0, R(1), R(2), R(3))
20 PRINT *, R
```
(m)
```
10 CALL COMP3 (X, Y, 4, 0, B(1, 1), B(2, 1), B(3, 1))
20 PRINT *, B(1, 1), B(2, 1), B(3, 1)
```
(n)
```
10 CALL COMP3 (X, -1.0, -2, 0, N, Y, Z, W)
20 PRINT *, Y, Z, W
```
(o)
```
10 CALL COMP3 (X, Y, Z, N, U, 3.0, V)
20 PRINT *, U, V
```
(p)
```
10 CALL COMP3 (Y, Z, -1.0, S1, N, V, W)
20 PRINT *, N, V, W
```
(q)
```
10 CALL COMP3 (X, Y, Z, -1, C1, V, W)
20 PRINT *, C1, V, W
```
(r)
```
10 CALL COMP3 (R, N, B(1, 1), B(2, 1), B(3, 1)
20 PRINT *, B(1, 1), B(2, 1), B(3, 1)
```
(s)
```
10 CALL COMP3 (2.0, 3.0, 4.0, 1, X, Y, Z)
20 PRINT *, X, Y, Z
```
(t)
```
10 CALL COMP3 (B(1, 1), B(2, 1), B(3, 1), N, B(1, 2), B(2, 2), B(3, 2))
20 PRINT *, B(1, 2), B(2, 2), B(3, 2)
```
(u)
```
10 CALL COMP3 (2.0+X, 6.0, 4.0-Y, 1, U, V, W)
20 PRINT *, U, V, W
```
(v)
```
10 CALL COMP3 (3.0-X, 2.0, 4.0-Y, N, U, V, X)
20 PRINT *, X, U, V
```
(w)
```
10 CALL COMP3 (SQRT(8.0*X), 3, Z+X, N/2, V, W, U)
20 PRINT *, U, V, W
```
(x)
```
10 CALL COMP3 (R(3), R(2), X, 1, 3.0*U, V-1.0, W)
20 PRINT *, U, V, W
```

2. Section 12.1.1 shows a subroutine named CHEMVA1 and its invocation by a main program under different sets of circumstances. Using the same variables as in the example, set up a sequence of FORTRAN statements that will result in the following behavior for CHEMVAL:

   When MW is less than 200, OUTCOME = COEF*VISC** (ENRG+REFIN) −REDUC
   When MW is >= 200and < 400, OUTCOME = VISC** (ENRG+REFIN) −REDUC
   When MW is > 399 and < 600, OUTCOME = VISC**ENRG − REDUC
   When MW is >= 600, OUTCOME = VISC − REDUC

   The computation of MIXVAL is not affected by MW. In fact, the subroutine CHEMVAL is not to be changed at all.

3. Write a subroutine named REMOVE that removes the vowels from the first argument and stores them in the second argument, starting in its leftmost position. Unused positions in the second argument are to be filled with blanks. Note that there is no guarantee that the invoking program always will supply a second argument that is long enough. Consequently, REMOVE must be prepared to recognize and deal with this possibility. One way to do this is to fill the second argument with a special character (e.g., +) to indicate that the invocation was not successful.

   Write a simple main program to test REMOVE. Use a variety of arguments having different lengths, including some that will not work.

4. Write a function named SUMVAL that returns the sum of all the elements in a 24-element one-dimensional array of real numbers.

5. Generalize the function in Problem 4 so that it will operate on a one-dimensional real array of any size.

6. Write a subroutine named ARYPRC that processes a one-dimensional array of real numbers whose size is not defined until the subroutine is invoked. ARYPRC produces four results:

   (1) The sum of all elements each of whose values is below a value specified by the invoking program;

(2) The number of elements used in preparing the sum in (1);
(3) The sum of all elements not used in preparing the sum in (1);
(4) The number of elements used in preparing the sum in (3).

Test ARYPRC with a suitable main program.

7. Design and implement ARYPRC so that it will operate on a two-dimensional array.

8. Write a subroutine named LTRCNT that performs the following processing: when it is invoked, it is given the following arguments:

(1) A one-dimensional character string array;
(2) A character string of some length (let's call that length STRLTH) ;
(3) A one-dimensional integer array where the number of elements is the same as STRLTH.

LTRCNT is to examine the character array and count the number of occurrences of each character in the string mentioned in (2) above. These numbers are to be returned in the integer array. For instance, suppose the character string is 'BJ87H'. Thus, the integer array will have five elements. When the subroutine completes its processing, the first elements will indicate how many B's there are in the character array, the second will show how many J's were found, and so on.

Set up LTRCNT so that it can handle a character array with as many as 50 elements with a maximum length of 20:

(a) If LTRCNT is invoked using a 25-element character array, how long can each element be?
(b) If LTRCNT is invoked using 10-character array elements, how many elements can the character array have?
(c) If LTRCNT is invoked using 12-character array elements, how many elements can the character array have?
(d) If LTRCNT is invoked using a 30-element character array, how long can each element be?
(e) What is the largest allowable value for STRLTH?

Test LTRCNT with a suitable main program.

9. Take a look at the following subroutine:

```
SUBROUTINE      ADJ (A, N)
IMPLICIT        NONE
REAL            A (N) , REF
INTEGER*2       N, I
REF = A(1)
DO I=2, N
  IF (A(I) .GT. REF) THEN
    REF = A(I)
  ELSE
  END IF
END DO
REF = 1.0/REF
DO I=1, N
  A(I) = REF * A(I)
END DO
RETURN
END
```

(a) Using no more than two sentences, describe what ADJ does. Now here is part of a main program:

```
      PROGRAM    C12P9
      REAL       RDNGS (24)
      INTEGER*2  SOMANY, J
      DATA       RDNGS/-3.4, 8.0, -1.0, 2.2, 3*6.1, 4*2.8, 6.6,
     1                 4*-2.7, 8.1, 7.4, 5*5.5, -5.5/
```

Given below are several sequences of statements, each of which includes a CALL to ADJ. Some of these will not work. Indicate the ones that will not work and explain what is wrong with them. For those that will work, show what the output will be. Assume that each of these sequences would be placed immediately after the DATA statement. Treat each of these sequences independently.

(b)
```
CALL ADJ (RDNGS, 24)
DO J=1, 24, 4
   PRINT *, RDNGS (J) , RDNGS (J+1) , RDNGS (J+2) , RDNGS (J+3)
END DO
```

(c)
```
CALL ADJ (RDNGS (6) , 10)
DO J=1, 24, 4
   PRINT *, RDNGS (J) , RDNGS (J+1) , RDNGS (J+2) , RDNGS (J+3)
END DO
```

(d)
```
SOMANY = 8
CALL ADJ (RDNGS (18) , SOMANY)
DO J=1, 24, 4
   PRINT *, RDNGS (J) , RDNGS (J+1) , RDNGS (J+2) , RDNGS (J+3)
END DO
```

(e)
```
SOMANY = 8
CALL ADJ (RDNGS (3) , 3*SOMANY)
DO J=1, 24, 4
   PRINT *, RDNGS (J) , RDNGS (J+1) , RDNGS (J+2) , RDNGS (J+3)
END DO
```

(f)
```
SOMANY = 8
CALL ADJ (RDNGS (SOMANY) , SOMANY)
DO J=1, 24, 4
   PRINT *, RDNGS (J) , RDNGS (J+1) , RDNGS (J+2) , RDNGS (J+3)
END DO
```

Now, here is part of a new main program:

```
      PROGRAM      C12P9A
      REAL         BLD (-6: 7)
      INTEGER*2    QTY, J
      DATA         BLD/4.4, 3.3, -5.5, 2.2, 1.1, 8.8, -3.3, -7.7,
     1                        0.0, -2.2, 7.7, -8.8, -9.9, -10.8/
```

Each of the following sequences of statements is to be considered as part of this main program. As before, we shall invoke ADJ and some of the sequences will not work. Explain the errors in those cases, and show the output resulting from the ones that will work:

(g)
```
CALL ADJ (BLD, 13)
DO J=1, 10, 4
PRINT *, BLD (J) . BLD (J+1) , BLD (J+2) , BLD (J+3)
END DO
```

(h)
```
QTY = 7
CALL ADJ (BLD (-4) , QTY)
DO J=-6, 11, 4
PRINT *, BLD (J) , BLD (J+1) , BLD (J+2) , BLD (J+3)
END DO
PRINT *, BLD (12) , BLD (13)
```

(i)
```
QTY = -3
CALL ADJ (BLD (0) , QTY+10)
DO J=1, 12, 3
PRINT *, BLD (J-7) , BLD (J-6) , BLD (J-5)
END DO
PRINT *, BLD (5) , BLD (6) , BLD (7)
```

(j)
```
QTY = 3
CALL ADJ (BLD (0) , QTY+10)
DO J=1, 12, 3
PRINT *, BLD (J-7) , BLD (J-6) , BLD (J-5)
END DO
PRINT *, BLD (5) , BLD (6) , BLD (7)
```

10. Here is another subroutine:

```
SUBROUTINE      QUANTM (MTRX, R, C, PEG)
INTEGER*2       R, C, MTRX (R, C) , PEG (4) , I, J
DO I=1, R
  DO J=1, C
    IF (MTRX (I, J) .LT. PEG (1)) THEN
      MTRX (I, J) = PEG (1)
    ELSE IF (MTRX (I, J) .LT. PEG (2)) THEN
          MTRX (I, J) = PEG (2)
        ELSE IF (MTRX (I, J) .LT. PEG (3)) THEN
              MTRX (I, J) = PEG (3)
            ELSE
              MTRX (I, J) = PEG (4)
            END IF
        END IF
  END DO
END DO
RETURN
END
```

The following is part of a main program that we shall use to invoke QUANTM:

```
  PROGRAM       C12P10
  INTEGER*2     OBS (4, 4) , CRNR (4) , ROWS, COLS, KR, KC
  DATA          CRNR/4, 8, 12, 16/,
 1              OBS/5, -8, 19, 22, 3, 0, 4, -4,
 2                   -17, 21, 3, -6, 31, 11, 5, -21/
```

(a) What does QUANTM do? As was requested in Problem 9, give a brief explanation. Do not paraphrase the individual statements.

Some of the following sequences have errors in them. Indicate which ones they are and why they are wrong. For those that are right, show what the printout will produce:

(b)
```
CALL QUANTM (OBS, 4, 4, CRNR)
DO KR=1, 4
  PRINT *, OBS (KR, 1) , OBS (KR, 2) , OBS (KR, 3) , OBS (KR, 4)
END DO
```

(c)
```
CALL QUANTM (OBS, 4, 4, CRNR (1) , CRNR (1) , CRNR)
DO KR=1, 4
  PRINT *, OBS (1, KR) , OBS (2, KR) , OBS (3, KR) , OBS (4, KR)
END DO
```

(d)
```
CALL QUANTM (OBS (1, 1) , 3, 3, CRNR)
DO KR=1, 4
  PRINT *, OBS (KR, 1) , OBS (KR, 2) , OBS (KR, 3) , OBS (KR, 4)
END DO
```

(e)
```
CALL QUANTM (OBS (1, 1) , 3, 4, CRNR)
DO KR=1, 4
  PRINT *, OBS (KR, 1) , OBS (KR, 2) , OBS (KR, 3) , OBS (KR, 4)
END DO
```

(f)
```
CALL QUANTM (OBS (1, 2) , 4, 4, CRNR)
DO KR=1, 4
  PRINT *, OBS (KR, KR)
END DO
```

(g)
```
CALL QUANTM (OBS (2, 2) , 3, 3, CRNR)
DO KR=1, 4
  PRINT *, OBS (KR, 1) , OBS (KR, 2) , OBS (KR, 3) , OBS (KR, 4)
END DO
```

(g)
```
CALL QUANTM (OBS (2, 2) , 3, 3, CRNR)
DO KR=1, 4
   PRINT *, OBS (KR, 1) , OBS (KR, 2) , OBS (KR, 3) , OBS (KR, 4)
END DO
```

(h)
```
ROWS = 2
COLS = 3
CALL QUANTM (OBS (ROWS, COLS) , ROWS, COLS, CRNR)
DO KR=1, 4
   PRINT *, OBS (1, KR) , OBS (2, KR) , OBS (3, KR) , OBS (4, KR)
END DO
```

11. In an earlier chapter, we examined Horner's method as a relatively efficient way to compute the value of a polynomial expression. Recall that for a third order polynomial, the value of the polynomial at some particular X (which we shall call P3X) can be expressed as follows:

    P3X = C0 + X*(C1 + X*g) c2 + X*(C3) ) )

    Similarly, we can define the polynomial value P4X for a fourth order polynomial as

    P4X = C0 + X*(C1 + X*(C2 + X*(C3 + X*(C4) ) ) )

    In general, then, for an nth order polynomial,

    PNX = C0 + X*(C1 + X*(C2 + X*(C3 + ···+ X*(CN) ) . . . ) )

    With a slight amount of manipulation, this method lends itself nicely to implementation as a function involving a simple loop. The arguments to such a function consist of an array of coefficients (the C's in the formula used earlier), the degree (order) of the polynomial (we called it N before), and the point at which the polynomial is to be evaluated (X in the previous formula). Note that if N is the degree of the polynomial, then there are N+1 coefficients. Write such a function (name it POLY) and test it with a suitable main program. Note that the processing inside the function is simplified to some extent if the array of coefficients starts with C(0) rather than C(1). This makes it easier to relate the first coefficient (C(0)) to C0 in the formula, and so on through C(N), which corresponds to CN in the earlier formulation. The process is simplified even further if Horner's method is set up to proceed in the direction opposite to that shown in the formula. That is, if we start out by initializing our computed value to C(N), we can multiply it by X and add C(N–1) ). The resulting value, then, can be multiplied by X and C(N–2) added, and so on through the array of C's. Eventually, this brings us to a point where the only value needed to complete the computation is C(0), and that value simply is added in.

12. Write a program that processes an Nth order polynomial to find a root close to zero. The root of a polynomial is a value of X such that

    PNX = 0

    Each input set to this program consists of a value for N followed by N+1 coefficients (i.e., C), C1, C2, . . . , CN). The program is to process any number of input sets, using an N of zero to stop the run. A maximum N of 6 is to be allowed. For each set processed, the program is to print an echo of the input followed by the value of the root.

    The basic approach in designing an algorithm to perform the required processing is to try various values of X until the polynomial value comes out to be zero. Since it is impractical to *compute* a value of zero in a machine (we can *set* a value of zero but it is unlikely that we can *arrive* consistently at exactly zero), we shall have to define a range around zero that we agree will be "close enough." For this problem, we shall consider 1.0E–4 (i.e., 0.0004) to be acceptably close. Thus, instead of testing to see whether some value (say POLY(X) ) is equal to zero, a more practical test would be one in which we compare ABS (POLY (X) ) to some parameter (let us say its name is EPSLON) which we have set to 1.0E–4 by means of a PARAMETER statement.

    Having identified the major test, we can look at the trial and error process. Starting with X=0.0, we can compute the polynomial value at that point so that we have a reference against which to compare later trials. (Of course, if PNX at X=0 is acceptably close to zero (i.e., it is no larger than EPSLON, the process is complete.) If it is larger (as it is likely to be), we try at another value for X, say, 0.5 larger than the previous X. If the polynomial value at that point is smaller (i.e., closer to EPSLON) than the one computed for the previous trial value of X, we are encouraged to believe that we are on the right track, adn we make X larger again by the same amount we used before. This process repeats as long as POLY (X) keeps getting smaller and smaller. As soon as POLY (X) begins moving in the other direction (i.e., away from EPSLON),

we have gone too far. So no, we make X smaller, by an amount half as large as we used to make it bigger for earlier trials. This process continues as described: we change X in a certain direction as long as ABS (POLY (X) ) keeps getting closer and closer to EPSLON. As soon as we have to change direction, that change is a signal for us to halve the amount by which we have been changing X. If INCR is the amount of the change (and EPSLON and POLY have their previous meanings), we can summarize this activity in terms of the pseudocode given below:

```
"Initialize INCR to 0.5, X to 0.0."
"Compute ABS (POLY (X) )."
while ABS (POLY (X) ) > EPSLON:
  "Add INCR to X."
  "Compute ABS (POLY (X) )."
  if
    new ABS (POLY (X) ) > previous ABS (POLY (X) )
  then
    "INCR = -0.5 * previous increment."
  else
  endif
endwhile
"Print successful output values."
```

13. Write a subroutine named MX1 that operates on an N × N array (N no greater than 5) to produce the sum of all the elements, the sums of the elements in each column, and the sum of the elements in each row. A second subroutine, MX2, is to compute the relative sums. That is, the sum of the elements, say, in a given column is to be expressed as the ratio of that sum to the overall sum. The main program in which these routines operate is to print the array "as received" followed by the overall sum, row sums, relative row sums, column sums, and relative column sums. Each matrix consists of real values and the main program is to process any number of such matrices. Since N (the number of rows/columns in a matrix) can vary, use an N of zero to terminate the run.

14. Write a program that processes any number of N × N arrays of real numbers (with N not to exceed 5) so that, for each input array, the program produces the following output:

    (1) The original array "as received."
    (2) The array with its elements rearranged so that each column is sorted in ascending order. For example, if we are looking at column J in array SQRARY, then the smallest value in that column will be in SQRARY (1, J), the next smallest in SQRARY (2, J), and so on.
    (3) The array with its elements rearranged so that each row is sorted in ascending order. NOTE: since this processing is to be independent of that producing the sorted columns, each one of these activities must start with the array's elements in their original positions.
    (4) The array with all of its elements sorted in ascending order, row by row. That is, if we are talking about a 3 × 3 array named MTR3X3, then the smallest value will end up in MTR3X3 (1, 1), the next smallest in MTR3X3 (1, 2), the next in MTR3X3 (2, 1), and so on.

15. Write a program that processes any number of N × N arrays of real numbers (with N not to exceed 5) to produce the following output:

    (1) The original array, one row per line.
    (2) The array (one row per line) with the values in each row divided by the element in that row that is in the array's major diagonal. (The major diagonal of a N × N array consists of those elements having the same row number and column number.) The element that serves as the divisor for a given row is not to be divided (if it were, it would be divided by itself); it is to be printed with its original value.

16. Write a single program that produces all of the output specified for the previous three problems.

17. An operation that is used frequently when information is being processed (whether a compute is involved or not) consists of searching a collection of items in order to find a particular piece of information or to certify that the information is not there. There are numerous ways to do this with a computer. The most straightforward method involves a *linear search*. We simply start at the beginning of a collection and look through the items until we find what we want (in which case we report its position in the collection) or until we have searched the entire collection without success (in which case we report a position of 0, or we use some other signal to indicate that the desired item is not in the collection).

We shall make use of this technique with the following problem: The Recall Automobile Company produces 14 models, each with its own distinctive name and price:

| *MODEL* | *PRICE* | *MODEL* | *PRICE* |
|---|---|---|---|
| ELF | $4844.25 | TROLL | $ 4972.64 |
| CUTIE | $5122.78 | IMP | $ 5454.54 |
| KOZYKAT | $5663.38 | IGUANA | $ 5834.88 |
| OSCAR | $6161.16 | BETTA | $ 6350.00 |
| LIONESS | $6622.44 | CALIGULA | $ 7189.98 |
| HEROIC | $7741.63 | SHOWBOAT | $ 8356.65 |
| CONSUMO | $9774.21 | BRONTOSAUR | $11208.73 |

Each time Recall sells an automobile, a set of input is prepared with the month, day, and year the sale was made, he salesman's number (a 5-digit integer), and the model name. Then, every week that week's data are collected and processed to produce a summary sales report. the program required in this problem develops hat report. After reading and storing the table of model names and prices, the program prints a line for each sale showing the date, salesman's number, the model of the car he or she sold, and its price. Then, after all the data have been processed, the program skips two lines and prints a summary consisting of a line for each model showing the model name, price, number of automobiles sold, and the total amount of revenue. A typical set of input appears as follows:

```
08 14 80     03244     'LIONESS'
```

18. While a linear search is reasonable for small tables (say, 25 entries or less), it becomes inefficient rapidly as the table to be searched gets larger. An improvement over the linear search occurs when the entires in a table are arranged in some order and we apply a *binary search*. This is basically the same type of technique we use when we look for something in the telephone book or dictionary. In each case, the entries are arranged in alphabetical order. Instead of starting at the beginning and working our way through the entires (as we would in a linear search), we pick our starting point by guessing about where in the book the desired entry is most likely to be. Then, we narrow our search down until we have reduced the search area to a small part of the total collection, small enough for us to look through each of the entires until we find what we want or certify that it is not there.

    When this type of search is prepared as an algorithm for computer use, the machine cannot make a "reasonable guess" about where to start looking. As a result, we design the search procedure so that it always starts in the middle of the table (or as close to the middle as we can get). Then, since the table's entries are sorted (let us say, in ascending order, with the lowest value first), the program can compare the value at the middle of the table with the value used for the search. If the value being looked for is higher than the table's middle value, the program "knows" that there is no reason to look at the lower half of the table. Instead, it concentrates on the upper half by splitting it in half and repeating the operation. In this way, the part of the table that still is of interest keeps shrinking until the examination is down to one or two entries, and the search can be concluded.

    We shall illustrate the technique by applying it to a table of 11 entries:

```
3  4  6  8  9  13  18  20  27  31  34
```

    For convenience, we shall refer to these entries as `T(1)` through `T(11)` and we shall say that the number we are seeking is `V`. For our first search, `V` will be `6`:

    (1) The search will start at `INT((1+11)/2)` or at `T(6)`. The value there is `13`. Since this is greater than `V`, this rules out the upper part of the table (i.e., `T(7)` through `T(11)`).
    (2) The second try, then, will be around the middle of the lower half of `T`. This point is computed as `INT((1+6)/2)` or `3`. Thus, the value in `T(3)`, i.e., `6`, is compared with `V`. Since they match, the search is successful, and the program can return a value of 3, indicating that the value we want is in the table at `T(3)`.

    Now, let us try a search for a `V` of `32`:

    (1) Our starting point is the same as before: `INT((1+11)/2)` or `6`. Since the value in `T(6)` (i.e., `13`) is less than the `32` that we want, the lower half of the table is ruled out.
    (2) The new midpoint, then, is `INT((6+11)/2)` or `8`. The value in `T(8)` is `20`. Since the value we want is larger, there is no reason to look anywhere in the table below `T(8)`.

(3) For our third try, then, the midpoint is `INT((8+11)/2)` or `9`. `T(9)` is `27`; still too low.
(4) Our fourth try puts the midpoint at `INT((9+11)/2)` or `10`. `T(10)`'s value of `31` still is too low.
(5) When we compute a new midpoint, we do it the same way as we have been, so that it is `INT((10+11)/2)`. This value comes out to be `10`, the same as it was before. This tells us that there is no reason to look further. The value we want is not in the table, and the procedure would return a position of zero, indicating failure.

Rewrite the appropriate part(s) of the program in the previous problem so that Recall's model/price information is searched using the binary technique described here.

19. Write a generalized subprogram that will conduct a binary search on a table of integer values where the size of the table is supplied as an argument. Test your subprogram with a suitable main program.

20. Write a program that reads a list of 40 words in alphabetical order. The length of each word is ten letters or less. (These are to be stored in an array named `DICT1`). Then, the program is to read a sequence of lines where each line contains a single word. There may be any number of lines and they may be in any order. For each of these lines, the program is to consult `DICT1`. If the word on the input line is found in `DICT1`, it is to be removed from there, and placed, in its proper alphabetical position, in a new array `DICT2`. Also, a line of output is to be printed showing the word and the message `''REMOVED FROM DICT1.''` If the input word is not present in `DICT1`, and never was there to begin with, the program is to print a line of output with the word and the message `''NO SUCH WORD IN DICT1.''` There is no guarantee that every input word will be different. The only guarantee is that each of the original 40 words will be unique. Consequently, if an input word already had been removed from `DICT1` earlier, the program is to print a line showing the word and the message `''REMOVED FROM DICT1 EARLIER.''` At the end of the run (use `'XXXXX'` as an end-of-data signal) leave two blank lines and print the following summary information: the number of input lines processed; the number of words removed from `DICT1`; and the number of input words that never appeared in `DICT1`.

# 13
# *Introduction to Input/Output*

The READ, WRITE, and PRINT statements we have been using thus far have served as simple means for getting information into and out of the processor. This convenience is not free. (If this were a different subject, the student might be asked at this point to list eight useful or desirable items that are truly free.) In exchange for this uncomplicated form, the programmer must give up much of the flexibility that otherwise could be applied to these processes. The programmer must rely instead on a relatively narrow set of controls imposed by the system. The next five chapters explore FORTRAN's wide range of input/output capabilities: what they are, how they work, and how to use them to our advantage. This first chapter sets the stage by defining the general framework within which these powerful features operate.

## 13.1 DATA AND FILES

In bringing input data into the processor or sending results to the outside, we have treated the data as individual items, related to one another only in terms of their meaning within the particular program. Even when a collection of items was organized as an array, that organization did not mean anything outside the context of the program. In fact, until the values were read in by a statement that treated them as array elements, they were just values. Since we have been using only a basic form of FORTRAN's extensive input/output capabilities, we were able to ignore the fact that there is an organizational structure imposed on all information transmitted to or from the central processor. As we begin adding more and more powerful input/output features to our programs, an understanding of this structure will become increasingly useful.

The basis for this input/output organization is the file. While the term certainly is familiar to everyone, it has a very special meaning here. It describes a collection of data to be transmitted by a program. When that transmission occurs from the outside world to the processor, the file is an *input file* and its data are being *read*. When the direction of transmission is from the processor to the outside world, the file is an *output file* and its data are being *written* or *printed*. (Printing is just a special case of writing. The only difference is that the particular type of output device is human compatible, that is, people can read what is written there.) Thus, regardless of what a program does, its operation usually involves the reading of data from one or more input files and the writing of data into one or more output files. In special cases, a particular program may be designed to process information that it generates internally (so that it does not use any input in the sense mentioned above), but it will produce output nonetheless, if it is to be of any use.

### 13.1.1 Organization of Files

Files have certain structural properties which go beyond the individual data values contained in them. FORTRAN is designed to recognize and use files so that versatile programs can be written to handle a wide range of data forms with a variety of physical

devices. In this section we shall determine what these properties are and how they relate to one another.

**13.1.1.1 Records** A file consists of a collection of *records* and each record, in turn, consists of a collection of *data items*. There are no specific rules that define precisely what a record should look like, what it should contain, or how many there should be in a file. These aspects are up to the programmer, and they are determined by the application's particular requirements. For instance, in Example 12.2 of the previous chapter, there is an input file with two kinds of records: One type contains a single data item (e.g., a run number), and the other type, though also on a card, contains several items (e.g., part of a list of words). The file itself consists of a sequence of however many records there happen to be. In the sample run for this program (the one shown in Figure 12.5), the input file had twelve records: a run number (one record); a word list (ten records containing four words each); and a record with the terminating run number of zero.

It is important to understand that a record is an *organizational* concept, not a physical one. Because of the basic nature of certain input/output devices, there may be a relation between a single record and, say, a punched card or a line of data on a terminal display, but that does not have to be the case. For instance, we can write a record on magnetic disk and specify any desired length.

**13.1.1.2 Relations Among Records in a File** It was mentioned earlier that a file consists of a *sequence* of records. That term conveys the idea that the records are in some particular order. Regardless of the physical medium on which a file's data are recorded, each file has a first record and a last record. This is easy enough to see if the file happens to be on a deck of punched cards or its records happen to correspond to lines on a printed page. The same concept holds true even when the records are represented as impulses on a magnetic disk or drum.

The physical order in which a file's records are stored may dictate the order in which such records are available to a program. When this is true, that file is known as a *sequential file*, and the process of obtaining that file's records is known as *sequential access*. A file stored on a reel of magnetic tape is an example. In order to get the information from a record somewhere in the middle of the file, we have no choice but to start with the first record and work our way to the one we want. If we just finished working with a record from somewhere in the middle of the file and we want to work next on a record from some place earlier in the file, we must inch our way back to that record, one record at a time. In other words, when we are at a particular position in a sequential file, we have immediate access only to the record at that position.

With some files it is possible to make records available to a program in any order, regardless of their physical order. Such files are called *direct files* and the process of obtaining records in any order is called *direct access*. Of course, direct access is possible only if the file is represented physically on a device built to allow it. A magnetic disk is an example of such a *direct access device*. If you are not familiar with a disk, think of it as operating somewhat like a phonograph record. Regardless of where the playing arm is positioned, it can be moved to any other position on the record without playing the material in between or going back to the beginning.

Now that the basic difference in accessibility has been established, we can deal with one more issue regarding the interrelation of records in a file: How do we know which record we want? In a sequential file there is not much of a choice: when we read from such a file, as we have been doing in every program so far, we read the next record, i.e., the one that the physical mechanism is positioned to read. In a direct file, where the "next" record can be *any* record in that file, we need to be able to describe the record we want so that the program can select it from all the others. This is done by assigning to each record a unique *record number* based on the record's physical position in the file.

**13.1.1.3 Data Representations on Records** Regardless of its type or meaning, the data on a record can be represented in one of two ways: *formatted* or *unformatted*. In a formatted record the data are represented as characters. That is, each numerical digit, letter of the alphabet, period, comma, blank, dollar sign, or anything else is read or written as an individual character. The processor, by "looking" at a formatted record, cannot "know" anything about the *values* represented there. Information as to how to *interpret* those characters must come from a program. Since we have been using formatted records all along (we just have not found it necessary to apply that name to them), we can look at an example of such a record and see how this concept applies.

Figure 13.1 shows a representation of a line containing some data, along with part of a program—enough to define some variables and read them in. We shall treat the line as input to the program. Using the terminology just introduced, we can say now that the line is a formatted record in a sequential input file. Note that *we* can say that, but the computer cannot without some instructions that tell it how to process the line. We are able to look at the data (even before we see the READ statement) and know that there are five data items there: three numerical values and two character strings. Without the program, of course, we do not know to which variables these values belong; but our knowledge of FORTRAN's rules enables us to recognize the commas as separators (so we can tell how many items there are), and the apostrophes as string delimiters (so we can tell that the last two items are character strings). However, as far as the computer is concerned, none of this is apparent. What the computer "sees" is a string of 80 characters (i.e., one line's worth). All but 28 of them happen to be blanks, and the rest are just nonblank characters: numerical digits, letters, apostrophes, commas, and a period here and there. It is not until the computer "sees" the instructions generated by the READ *, statement that it "knows" which rules to use for proper interpretation of the input line (e.g., to look for commas and treat them as separators; to handle the 408 as a number; and to handle the '408' as a character string). This type of input is known as *list-directed formatted* input. There is another type of formatted input/output known as *edit-directed formatted* input/output in which FORTRAN's interpretive rules are more complex. Edit-directed input/output enables the programmer to control the data formats completely. As we shall see, there are other types of READ and WRITE statements that associate with these other rules for data

```
236.4,-29.67,408,'408','63J'
```

**(a)**

```
PROGRAM          FIMPEH
IMPLICIT         NONE
REAL             TOP, BOTTOM
INTEGER*2        MAGNI
CHARACTER*3      IDN(2)
     .............
     .............
READ *, TOP,BOTTOM,MAGNI,IDN
     .............
     .............
     .............
PRINT *, IDN,TOP,BOTTOM,MAGNI
     .............
     .............
END
```

**(b)**

**FIGURE 13.1** **(a)** List-Directed Input Data Record. **(b)** Program Fragment Illustrating Formatted Input/Output.

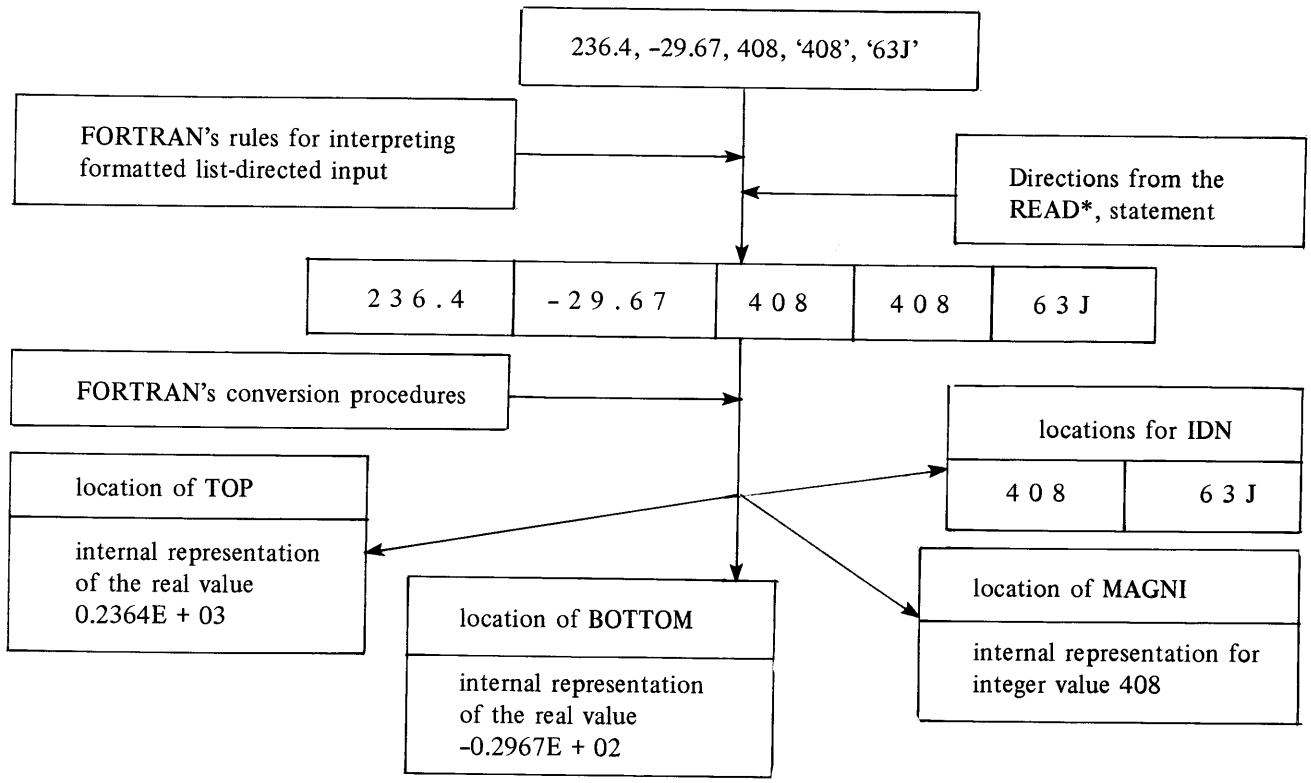



**FIGURE 13.2** Diagram Representing How List-Directed formatted Data is Interpreted and Converted.

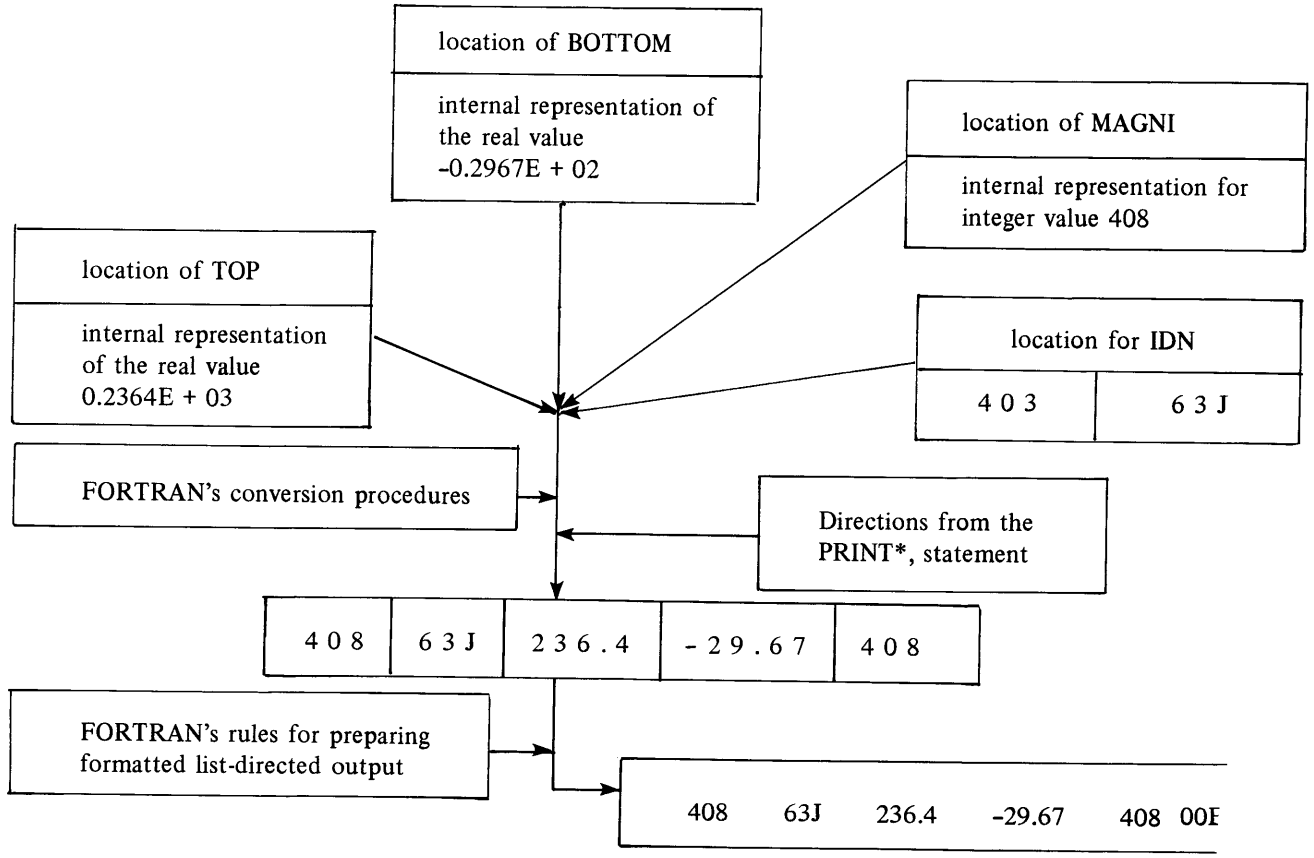





**FIGURE 13.3** Diagram Representing How List-Directed Formatted Output is Produced.

interpretation. Once the proper rules have been identified, the required values can be picked out and, ultimately, converted to the final forms in which they will be stored and used.

This general process of interpreting and converting list-directed formatted data is summarized diagrammatically in Figures 13.2 and 13.3. As Figure 13.3 shows, exactly the same series of events applies for output: the internal values, represented in a form that the processor is designed to "understand," are converted to strings of characters, and those characters are used to build the formatted output record.

In an *unformatted* data record, the values are represented in a form that the computer's electronic circuits are designed to recognize directly. No physical separation, interpretation or conversion is necessary; all of those mechanisms are built into the machinery itself, so that the way the values are interpreted has nothing to do with the type of program or even the language in which it is written.

Of course, the *meaning* of the values still comes from the program (and from the programmer's head). There is no point in showing an example of an unformatted record. Its appearance will depend on the particular type of computing system for which it is prepared. In any case, its contents are unintelligible to anyone except those people familiar with the internal operation of that specific machine. We see, then, that unformatted records are not intended for direct use by humans. Rather, they generally are produced as output by one program with the idea that they will serve as input to another one. Eventually, one or more programs in such a sequence will produce final results that are human-readable.

For example, a program to collect and organize data generated by a day's stock market trades would be designed to produce such data as an unformatted file. That file, then, can be used as input to a variety of subsequent programs, some of which may produce final (i.e., human-readable) output while others may process the collected data to generate other unformatted files for use by still other programs. To carry the point further, let us list just a few examples of the ways in which such data might be processed further:

1. One program (P1) might sort the data alphabetically so that all transactions relating to a given stock are grouped together. The result could be an unformatted file.
2. Another program (P2) might use the sorted file from P1 to produce a nicely laid out (and impressively long) printed report.
3. A third program (P3) might use the same sorted file to produce an unformatted file in which each record represents a summary of the transactions performed for a given stock.
4. A fourth program (P4) might process the file from P3 to produce a printed version of the summary.
5. Yet another program (P5) might start with the original file and perform an hour-by-hour analysis of the day's progress, producing a printed report.

This general process is summarized in Figure 13.4. It is clear that the list of possibilities is endless. The point is that there are many instances in which it is unnecessary for people to see a program's complete results (once they are sure, of course, that the program works properly).

The reason for doing this is that unformatted records are considerably more compact, and they can be transmitted more quickly than formatted ones containing equivalent data. Moreover, if no data conversion is necessary, we can save additional time. Later we shall deal with the types of READ and WRITE statements that are designed for unformatted records.

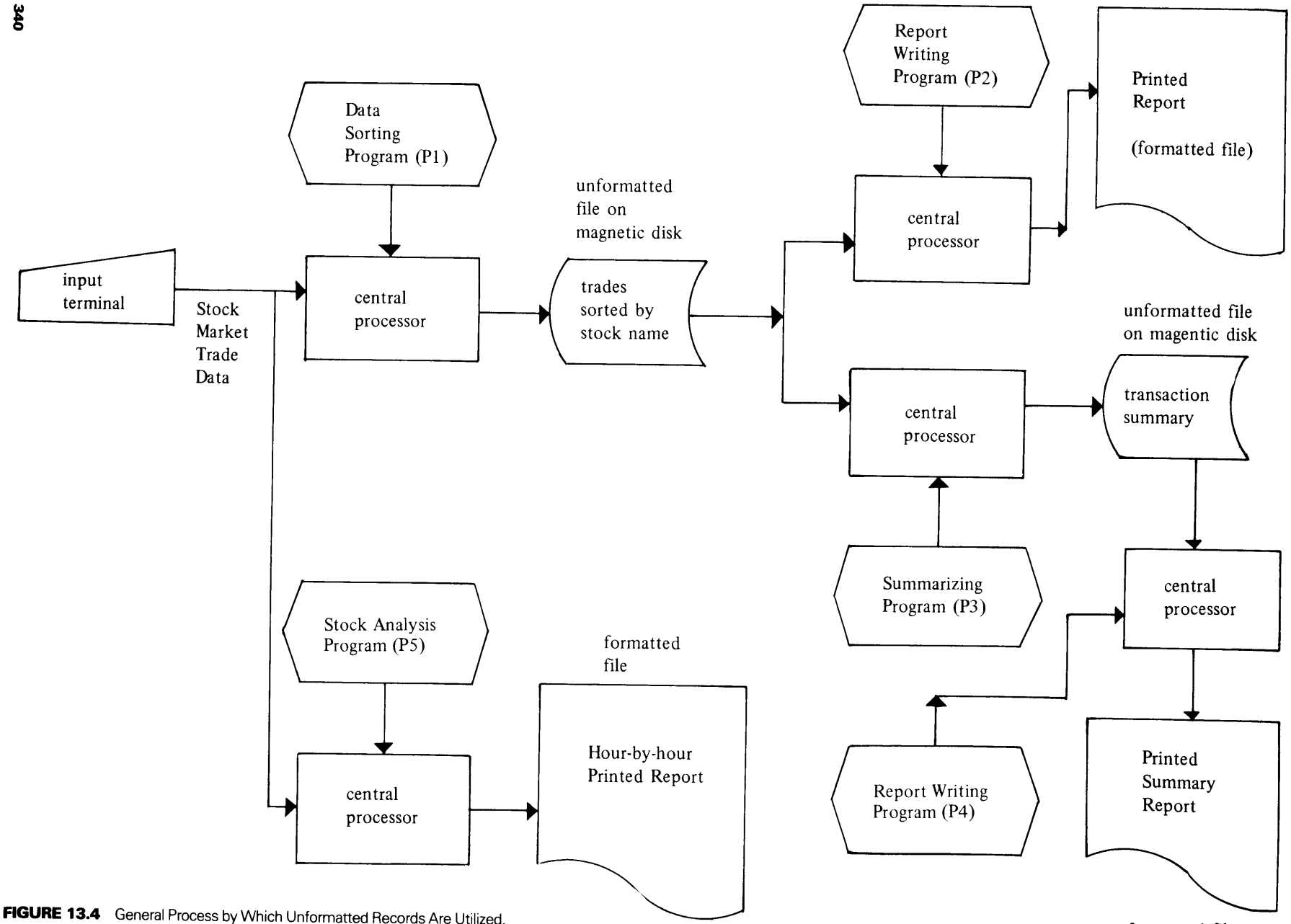



**FIGURE 13.4** General Process by Which Unformatted Records Are Utilized.

**13.1.1.4 External and Internal Files** In examining the concepts of files, records, and data items, we have not emphasized any definite connections with physical devices. The reason is fundamental: A file, like a record, is an *organizational* concept. If we think of a file as a collection of records, distinct from any physical medium on which those records may be represented, the various input/output concepts will fall neatly into place.

One such concept is the difference between internal and external files. An *external* file, to FORTRAN, is a collection of records residing on a physical medium that is separate from the processor. For instance, the list of input words from Example 12.2, when keypunched onto cards, constitutes an external file. The same data, when displayed on the line printer, form another external file.

An *internal* file, on the other hand, is a collection of records right in the main memory of the processor. Since a record consists, ultimately, of data items, it would appear that an internal record is the same as a group of ordinary variables. After all, a variable is a place reserved in main storage for a data item, and this sounds like the same thing. The important difference, however, is that the internal file carries with it all of the organizational bookkeeping that FORTRAN sets up for external files. This means that, even though the records are already in the processor, we are able to pretend that they are on some external device. Then, we can "read" from that (imaginary) "device" just as if it were a terminal keyboard, magnetic tape or disk, card reader, or any other input unit.

The existence of an imaginary input device brings with it FORTRAN's powerful facilities for interpreting a record's contents and converting them in accordance with the directions provided by the program. This facility gives FORTRAN 77 an enormous flexibility which enables us to read in data as a string of characters without committing ourselves to a fixed, specific interpretation of those characters. In fact, we can look at those characters, once they are inside, and use their contents to determine how they will be interpreted and converted. We shall see this versaility in use later on. Right now it is important to establish the fact that FORTRAN 77 provides this opportunity.

There is nothing in FORTRAN that restricts the use of internal files to input. We can use the same mechanism to gather some variables and employ FORTRAN's interpretive and reformatting capabilities to prepare a record for an internal file by "writing" the record onto an imaginary output device.

It follows that *internal files can be only formatted files.* Two more restrictions are that *internal files must consist only of character strings*, and *such files must be sequential.*

### 13.1.2 Files and Units

While the perception of files as disembodied sequences of data records is useful, we cannot ignore the fact that each file must reside on a physical medium if it is to be of any use. This means that, sooner or later, we are going to have to associate the file with a card reader, printer, terminal, magnetic tape, disk, or some other device to or from which the records are going to be transmitted.

The transmission can be understood through the concept of a *unit.* A unit is a piece of bookkeeping. Specifically, it is a link between a file and a particular physical device. It works this way: Each computing system is equipped with some combination of physical input/output devices, and each of these devices is represented by an entry in a table maintained by the computer's software. Each physical device is identified as a *unit* with its own unique unit designation. When a computing center adds a device to its computer system, an additional designation will be created as part of the installation process.

As far as a FORTRAN program is concerned, a file is inaccessible until it is *connected* to a unit. This connection, of course, is an organizational one, i.e., another bookkeeping item. FORTRAN makes the connection between a file and its unit in response to a program statement or as part of its automatic bookkeeping services. In either case, the

result is an entry that says, basically,

> "In this program, unit so-and-so is connected to file such-and-such. Consequently, an input/output statement involving this unit will transmit data to or from that file."

Since all input/output activities involve a file, and since a file, though it may exist in concept, cannot actually be used until it is connected to a unit, all of FORTRAN's data transmission statements require a unit to be specified. When we request a READ operation, our instruction says, in essence,

> "I would like to bring data into the processor from the file connected to the unit I have specified in this statement. Go find out which file that is, and find out which physical device that is."

Similarly, a WRITE or PRINT statement carries the basic message,

> "I want you to place data in the file connected to the unit specified in this statement. This program's bookkeeping will tell you which file that is, and the system's bookkeeping will tell you which physical device is involved."

Here again, FORTRAN 's automatic mechanisms have been at work for us. We have been specifying a standard unit in every READ, WRITE, and PRINT statement. In a READ statement, we have specified the *standard input unit* by default. For most cases, the physical device associated with this unit is the user's terminal. Similarly, the default unit in a WRITE statement usually results in the assignment of the user's terminal also. The absence of an explicit unit number in the PRINT statement causes FORTRAN to assign the *standard system output unit.* This usually is a printer of some kind. The asterisk (as in READ *, or PRINT *, ) tells FORTRAN that the data are to be read (or written) in list-directed form. Thus, it is possible to issue a statement such as

```
READ (5, *) V1, MAXHT, LGTH, DEN
```

in which case FORTRAN is directed to read a list-directed record from the file connected to unit number 5. This statement is fine assuming:

1. The physical device associated with unit 5 is capable of delivering input. (For example, this statement, though legal in FORTRAN, would be impossible to execute if unit 5 happened to be associated with a printer).
2. There is a file connected to unit 5.

We did not have any problems with file-unit connections in our earlier programs because the standard input and output units are connected automatically to files that are created as part of the system's support activities. Such units are called *preconnected* units and are made available to any program. Different HP computers have different preconnected units. For instance, the HP 1000 systems operating in the RTE environment use unit 1 (the user's terminal) as the standard input device, and the standard output device is the line printer, designated as unit 6. For HP 9000 computers operating in the HP-UX environment, unit 5 is the preconnected unit to the user's terminal and is designated as the standard input device. The standard output device in that environment, designated as unit 6, also is preconnected to the user's terminal. Consequently, it is advisable to check the appropriate reference manual for the exact unit designations in force for that HP system.

While the concept of a unit is relatively easy to understand, you might ask, "Why go through the involvement of specifying a unit and then finding out which file corresponds to that unit? Why not just specify a physical device directly?"

The answer brings to light one of the powerful benefits of the file concept: A file may be associated (at different times) with each of several different physical representations

without changing its conceptual structure. In terms of the program, this means that it is possible, for instance, to write records for an output file stored, say, on a magnetic tape and then, on some other occasion, write records for the same file, this time on a magnetic disk. For both instances, the program is identical. Even the unit designation need not be changed. The only change that is required is in the association between units and physical devices, and HP systems are designed to handle such changes simply and conveniently.

### 13.1.3 Operations with Files

The programmer working in FORTRAN 77 makes use of several powerful file operations. Some of these relate directly to corresponding FORTRAN statements while others are less apparent in that they are submerged within larger activities. We shall discuss these operations briefly in the next few sections.

**13.1.3.1 Creating a File** Since a file is an organizational idea, it has its own existence, separate from any program or unit. Because of this we can talk about a file that has no records in it, is not connected to any unit, is not known to our program, and yet still exists. The point is that the bookkeeping information which describes the file is enough to establish that file's existence.

*File creation* is the process that produces such bookkeeping, thereby defining the existence of a file to the computing system. Some files are created by individual programs while others are created automatically as a standard system activity. The input and output files we have been using in all of our programs are examples of this automatic file creation.

A file can be created by one program and used by another. The reason this can happen is that the information produced by the file creation process is "registered" with the system's file manager, which then can make it available to other programs. Such files can be identified by giving them explicit names (making them *named files*), or they can be created as *unnamed files*. Manipulation of the latter type is a little less flexible but still convenient for many applications.

FORTRAN 77 has no explicit statement for creating a file. Instead, certain types of statements cause a file to be created as part of some other activity. Mechanisms for explicitly creating files outside of HP FORTRAN 77 are described for HP 1000 computers in the RTE-6/VM Reference Manual. Corresponding information for HP 9000 computers is given in the HP-UX Reference Manual.

**13.1.3.2 Deleting a File** As the name implies, this process dismantles the organizational information associated with a file. Consequently, since the deleted file no longer exists as far as the system is concerned, its records no longer exist either and, therefore, cannot be obtained. The fact that the file handling capability is a system-wide resource means that it is possible for a file to be created and deleted by two different programs. In actual practice, this often turns out to be the case. For many applications it is convenient to have a program whose specific purpose is to create the required files and to initialize them for subsequent use by other, related programs. Later in the processing cycle, when a particular file has served its purpose, its data have been put to their intended use, and there is no reason to retain the data, another program will delete them. This situation is seen typically in the case of temporary files designed to hold intermediate values whose usefulness vanishes once the final results are produced.

As is the case with file creation, there is no specific FORTRAN statement for deleting a file. Instead, a file is deleted as part of the activities triggered by some of the other input/output statements. An explicit system command is available for file deletion. These system commands are described in the RTE-6/VM Terminal User's Reference Manual for the HP-1000 and the HP-UX Reference Manual for the HP-9000.

**13.1.3.3 Opening a File** In Section 13.1.2 we established the idea that a file (and its records) cannot be used unless that file is organizationally connected to some unit in the system. This connection is made by *opening* a file, an operation that makes a file available to a particular program. Since we know now that we have been using files, and we have not explicitly opened them, we can conclude that this also is part of FORTRAN's automatic file-handling mechanism. In addition, FORTRAN provides the OPEN statement when the programmer wants to connect a file explicitly. It also turns out that the OPEN statement provides a way for a FORTRAN 77 program to create files.

In this connection it is important to reemphasize the following point: The fact that a file exists (i.e., has been created) does not automatically make it available to every program in that system. Many (probably most) systems have special hardware and software facilities designed explicitly to *limit* the availability of certain files. For example, most students writing programs on a university computing system cannot open the university's administrative files (such as the grade file or tuition file), even though the files exist in the same system.

**13.1.3.4 Closing a File** Closing a file has the opposite effect of the "open" operation: It makes a particular file unavailable to that program, thereby signalling the system that the program is through with it (at least for the time being), but the file is not to be destroyed. The same file may be reopened later in that program. Also, the unit form which that file had been disconnected becomes available to be reconnected by another OPEN statement. Although that file must be opened and closed in the same program, it does not have to be opened and closed within the same *subprogram*.

Closing a file can be done explicitly by means of the CLOSE statement, in which case the programmer can reconnect that file later in the program, or it can be done automatically as part of the bookkeeping that is triggered when a program finishes. By the same token, this process also disconnects the unit from its file. That means that if we close a file (with a CLOSE statement) before the end of the program, the unit can be associated with some other file (or even reassociated with the same file) later in the program. In any event, the programmer must make sure that, at any time, *only one unit is assigned to a given file and vice versa.*

**13.1.3.5 Reading and Writing File Records** These operations perform the actual transmission of data between the central processor and a peripheral (input or output) component. Input is handled by FORTRAN's READ statement. In terms of our ideas about the underlying structure, we can say that this statement brings in a record from the file connected to a designated unit and stores that record's contents in specified locations.

Output is handled by one of two statement types. The WRITE statement asks for a general output operation in which the data from specified storage locations are used to construct a record. That record is sent to the file connected to the unit designated in the statement. When the physical output device associated with a particular unit number is a printer of some kind, the output operation (as we know) can be specified by the PRINT statement.

**13.1.3.6 Writing an ENDFILE Record** This operation creates a special kind of record (called an ENDFILE record) and writes that record as the next record on the file. The ENDFILE record contains no data. As the name implies, it is designed to serve as the file's last record. We shall see that when we read records from such a file, we can test for the ENDFILE record and make decisions based on its appearance.

As we stated earlier, it is possible to have a file with no data in it. In effect, such a file has the framework without the substance. One example of this structure is a file with nothing in it but an ENDFILE record.

An ENDFILE record is written by issuing an ENDFILE statement in which the programmer specifies the unit. As will be seen later, the programmer must make sure that

unit specification makes sense. For instance, it is impossible to write an ENDFILE record on the unit associated with a printer (even though it obviously is possible to write other records). Similarly, it makes no sense to write an ENDFILE record (or anything else, for that matter) on a unit associated with a card reader.

**13.1.3.7 Backspacing a File** This operation positions a file to the previous record, thereby making it possible for a program to work its way through a sequential file's records in reverse order. The action is requested explicitly by means of a BACKSPACE statement. Here again, the specified unit must be consistent with the physical capabilities of the associated device. A favorite joke back there in the Old West of computing was for one of the Grizzled Old Hands to ask a Fuzzyfaced Kid to write a program that backspaced the card reader (heh heh).

**13.1.3.8 Rewinding a File** The rewind operation positions a sequential file at its beginning (i.e., its first record). The name implies that the sequential file resides on a reel of magnetic tape; in fact, its origin dates back to a time when tape was the only extended storage medium available. Now, with many sequential files stored on magnetic disks and drums, the name has a broader application. While actual rewinding does not occur with these devices, the result is still the same as if the tape had been rewound.

FORTRAN's REWIND statement performs this operation on the file connected to the specified unit. The file must be sequential.

**13.1.3.9 File Inquiry** This operation enables a program to check the status of a file and base subsequent action on the results of its findings. FORTRAN 77's INQUIRE statement is used for this purpose.

As the previous sections indicate, many of the file operations can be triggered by statements designed specifically for that purpose, or they can be implied (and, therefore, carried out automatically) as part of the activities prompted by some other statements. We shall be able to continue taking advantage of this latter convenience, since much of our input/output will use standard files. However, we shall develop a firm grip on these file operations by using them in a variety of circumstances requiring files of our own design.

The next section presents a brief outline of FORTRAN's input/output statements. Some initial details are provided for each type, so that we can relate them a little more closely to the operations discussed before. The complete story will appear in the next few chapters.

## 13.2 INTRODUCTION TO FORTRAN 77'S INPUT/OUTPUT STATEMENTS

All of FORTRAN's input/output facilities are expressed through nine types of statements. These organize conveniently into three categories related to the kinds of operations being performed:

1. Data transfer.
2. File positioning.
3. Auxiliary operations.

We shall discuss each of these in turn, illustrating its basic form and usage.

### 13.2.1 Data Transfer Statements

This category consists of the READ, WRITE, and PRINT statements. Each of these takes on various forms depending on the type of file (external or internal), its access method (sequential or direct), and the structure of its records (formatted or unformatted).

**13.2.1.1. The READ Statement** To start, we shall build on the list-directed READ statement, since that form (in its basic version) already is familiar. One extension was introduced earlier in this chapter, when a unit number was specified as part of the statement:

```
READ (5, *) V1, MAXHT, LENGTH, DENSITY
```

The *unit specifier* (5) and the *format specifier* (*) are but two of a number of items (as many as six) that may be specified within the parentheses. The other specifiers will be touched on briefly in preparation for a complete discussion later on:

1. As we learned earlier, the END specification, used with a sequential input file, defines a statement to which the program will transfer when it encounters the file's ENDFILE record:

```
READ (5, *, END=199) V1, MAXHT, LENGTH, DENSITY
```

says, "Read the next record from the file connected to unit 5 using list-directed format and store the data in V1, MAXHT, LENGTH, and DENSITY. If you encounter the ENDFILE record during this attempt to read, go to statement 199." Thus, the use of the END specification eliminates the need for a special signal record after the last set of data, along with the explicit test for that signal.

2. The ERR specification, used with any input file, defines a statement to which the program will transfer when an error is encountered during an attempt to read. For example, if our READ statement is set up to bring in numerical data, and one of the items on a record contains a letter, an error signal is produced and the program branches automatically to the indicated statement. An example of such usage is seen by expanding the previous input statement further:

```
READ (5, *, END=199, ERR=177) V1, MAXHT, LENGTH, DENSITY
```

Now it says, "Read the next record from the file connected to unit 5 using list-directed format and store the data in V1, MAXHT, LENGTH, and DENSITY. If an ENDFILE record is encountered during this attempt to read, go to statement 199 and continue from there. If, on the other hand, there is an error during this attempt to read, go to statement 177 and continue from there." The END and ERR specifiers have nothing to do with the particular input unit. Thus, the same READ statement, when rewritten as follows,

```
READ (*, *, ERR=177, END=199) V1, MAXHT, LENGTH, DENSITY
```

still does the same thing. The only difference is that the input unit now is the standard system input unit. Note that the unit specifier (when it is included) must be the first one in the list, and the format specifier must be next. As seen in the above examples, the order of the other specifiers does not matter because they are named explicitly.

3. The REC specification, used with direct files (see 13.1.1.2), defines the particular record to be brought in by the READ statement. (With a sequential file, of course, this specification is irrelevant since the next record is determined automatically by the position of the file.) The form is

```
REC = recordnumber
```

where *recordnumber* is an integer indicating which record is to be read. Since direct access files cannot be list-directed (they may be formatted, but only with edit-directed formatting), it is illegal to include the REC = specification when the format is * or FMT = *.

4. The IOSTAT specifier provides the programmer with an opportunity to receive and examine a signal that reports the result of the READ operation. (We shall see that the same type of specifier can be used to report the outcome of other input/out operations.)

Usage is as follows:

IOSTAT = *statusindicator*

where *statusindicator* is an integer variable in which the program will deliver a value depending on the result. That value is zero if the READ occurred without error, positive (the exact value depending on the particular implementation) if an error is encountered, and negative (the value, again, depending on the implementation) if an ENDFILE record is encountered.

It also is possible to use named specifiers for unit number and format. When this form is used, our example (for unit 5) might look like this:

```
READ (UNIT=5,FMT=*,END=199,ERR=177) V1,MAXHT,LENGTH,DENSITY
```

Note that there is a particular restriction on the combined use of UNIT = and FMT = specifiers:

1. If the UNIT = form is used in a formatted READ statement, then the FMT = form also must be used.
2. If the unit in a formatted READ statement is specified without the UNIT = form, it must appear first. When this happens, the format may be specified either way, but it must be second.
3. If both UNIT = and FMT = are used, their order is unimportant.

When every specifier includes its name, we can write them in any order. However, it is a good idea to select a particular order and use it consistently. The one shown above is easily remembered. Many programmers, however, prefer the form used earlier, in which neither UNIT = nor FMT = is used (there is no choice for the other specifiers), and since this, too, is easily remembered, we shall use that form, as in the example

```
READ (5, *,END = 199,ERR = 177) V1, MAXHT, LGTH, DEN
```

for most purposes.

When the file is formatted but not list-directed (i.e., it is edit-directed), the format specifier no longer is an asterisk. Instead, it tells the program where to find the format description. This is done in one of several ways, the most common of which is by giving a statement number in which the programmer has recorded such a description. For instance, the statement

```
READ (5,15,END = 199,ERR = 177) V1,MAXHT,LENGTH,DENSITY
```

says, "Read the next record from the file connected to unit 5 using edit-directed format and store the data in V1, MAXHT, LENGTH, and DENSITY. A detailed format description is to be found in statement 15. If an ENDFILE record is encountered, go to statement 199 and continue from there. If, on the other hand, this input attempt produces an error, go to statement 177 and continue from there." The rules for the various specifiers, given above, apply here as well.

Additional types of specifiers, details on how to describe edit-directed formats, and other forms of READ statements, will be discussed in subsequent chapters.

**13.2.1.2 The WRITE Statement** The WRITE statement is constructed like the READ statement discussed in the previous section. The unit number (with or without the UNIT= form), format description (with or without the FMT= form), and the ERR= specifier are used in accordance with the same rules discussed for the READ statement. However, the END= specifier makes no sense with a WRITE statement and its use in such a statement is illegal. The other specifiers described for the READ statement apply here as well, with the same rules governing their use.

We shall now show an example statement similar to the one used in the previous section. If we specify

```
WRITE (6, 16, ERR = 277) V1, VFINAL, MAXHT, LENGTH, DENSITY
```

we are saying, "Use the data in V1, VFINAL, MAXHT, LGTH and DEN to prepare an edit-directed formatted record and write it as the next record on the output file connected to unit 6. An exact format description will be found in statement 16. If an error is produced in the process of performing this output operation, go to statement 277 and continue from there."

**13.2.1.3 The PRINT Statement** Since this output statement is intended for use with a particular type of device (i.e., a printer or another device "pretending" to be a printer), its form is necessarily restricted. Specifically, the statement contains only a format specifier and a list of items to be "printed." For instance, the statement

```
PRINT 26, VI, VFINAL, MAXHT, LENGTH, DENSITY
```

says, "Prepare an edit-directed formatted record using V1, FINAL, MAXHT, LENGTH, and DENSITY and write that as the next record on the file connected to the standard system output device. An exact description of the format will be found in statement number 26." There is no unit specifier because FORTRAN automatically assigns the unit number for the standard system output device. The parentheses are not used either since there is nothing else that can be specified. Note that this device, while usually a printer, need not actually be a printer. Instead, it can be some other device (like a magnetic tape or disk) from which we intend to print the records eventually. Similarly, there is nothing to prevent us from using a WRITE statement in which the unit number happens to be associated with a printer.

Use of the PRINT statement for list-directed output is not shown here; we are quite familiar with that form already.

Before we leave this second brief look at the PRINT statement, we shall use it to introduce one simple aspect of edit-directed output. This will give us an opportunity to become acquainted with the basic form, and it will provide us with a useful formatting feature.

When a FORTRAN program specifies an output operation by means of a PRINT statement, the information in that statement must include a signal that helps guide the physical operation of the printing device. Specifically, the device must be instructed whether to start a new line of print, skip a line, begin a new page, or start printing at the beginning of the same line that was printed by the last PRINT statement. This type of instruction is known as *carriage control.*

When the PRINT statement is used with list-directed output, as we have been doing all along, the FORTRAN compiler takes care of carriage control automatically by supplying a signal that moves the printer to the next line with each PRINT statement. If the programmer requires more complete control over the printer's operation, he or she needs to use edit-directed output. With this form, FORTRAN expects the first output character in each PRINT statement to be a carriage control signal, and not part of the data. We shall introduce three such signals at this point so that the general idea is established:

1. A blank character signals the printer to start printing at the beginning of the next line. The blank itself is not printed and, therefore, does not count as one of the output characters.
2. A character of '0' signals the printer to skip the next line and start printing at the beginning of the line after that.

3. A character of '1' signals the printer to start printing at the first line of the next page. The '1' itself is not printed and, therefore, does not count as one of the output characters.

It is quite legal to issue an edit-directed PRINT statement without an output list. This is a convenient way to manipulate the printer, as the following pair of statements illustrate:

```
      PRINT 16
   16 FORMAT ('1')
```

The 16 in the PRINT statement indicates that the edit-directed format is described in statement number 16 and, sure enough, statement 16 is a FORMAT statement whose first and *only* specification is a literal '1', a signal to start a new page. Since there is no output listed in the PRINT statement, all it will do is force the printer to the first line of the next page.

### 13.2.2 File Positioning Statements

FORTRAN provides three statements (BACKSPACE, REWIND, and ENDFILE) that do not actually cause the transmission of data. Instead, they are used to position a file for manipulation by other statements.

#### 13.2.2.1 The BACKSPACE Statement

When a sequential file is opened, one part of the resulting activity is to position the file at its first record. Then, when that file is read, its first record is brought into the processor and the file is set at the second record, and so on. The BACKSPACE statement repositions the file by moving it back one record. Thus, if a file happens to be set to have its fourth record read, a BACKSPACE statement will set it so that the next READ statement will read the third record. Note that the BACKSPACE skips over the record without reading it.

The form for this statement is

BACKSPACE *unitno*

where *unitno* is the number of the unit to which the file is connected. It also can be written as

BACKSPACE (UNIT = *unitno*)

When the second form is used, the ERR= and IOSTAT= specifiers may be included in the parentheses, along with one other specifier (IOSTAT) which will be discussed later. The unit number, of course, always must be specified regardless of which form is used.

A few simple rules define the use of this statement:

1. It can be used only with sequential files.
2. If a file happens to be positioned at its first record when a BACKSPACE is requested, nothing happens.
3. If a file happens to be positioned after its ENDFILE record when a BACKSPACE is requested, the file will be repositioned to a point just before the ENDFILE record.
4. BACKSPACE cannot be used with a file whose records are list-directed.

**13.2.2.2 The ENDFILE Statement** This statement writes an ENDFILE record at the point where a file is positioned. The writing of the ENDFILE record puts a boundary on that file in the sense that the data record immediately ahead of the ENDFILE record now is recognized to be the last record in the file. Even though it might be physically possible to write additional data records after the ENDFILE record, those records may just as well not be there, since it will be impossible to obtain them later on.

Once the ENDFILE is written, the file is positioned at a point just after the ENDFILE record. Consequently, the data records in that file cannot be reached until the file is repositioned either with a BACKSPACE or REWIND.

The ENDFILE statement is written using one of the same two forms shown for the BACKSPACE, i.e.,

```
ENDFILE unitno
```

or

```
ENDFILE (UNIT = unitno)
```

Here again, the ERR= and IOSTAT= specifiers may appear inside the parentheses when the second form is used.

**13.2.2.3 The REWIND Statement** This statement repositions a file to a point just before its initial record. If the file already is there, nothing happens. The statement may be written using either of the two forms described before. When the second form is used, either or both of the ERR= and IOSTAT= specifiers may be included. Thus, the statement

```
REWIND (UNIT = 2,ERR = 399)
```

says, "rewind the file connected to unit 2. If something goes wrong and the rewind cannot be done, go to statement 399 and continue from there." As we saw with the READ/WRITE statements, this also could be written as

```
REWIND (2,ERR = 399)
```

Alternatively, assuming STAT2 were declared as an integer variable, we could write

```
REWIND (2, IOSTAT = STAT2)
```

which says, "Rewind unit 2 and report the result of the operation in the variable named STAT2."

### 13.2.3 Auxiliary Input/Output Statements

Three additional statements (OPEN, CLOSE, and INQUIRE) provide important supportive services for file handling.

**13.2.3.1 The OPEN Statement** As outlined earlier, this statement makes the connection between a unit and a file. If the file to be connected does not exist, the OPEN statement will trigger activities that will create it as well. The general form is

```
OPEN (openlist)
```

The information in parentheses (*openlist*) consists of a list of specifications that help define the organization of the file. As many as nine types of specifiers may be included in this list.

1.The OPEN statement must include a unit specifier using either of the two forms described earlier (see, for example, 13.2.3.1). Thus,

```
OPEN (3)
```

and

```
OPEN (UNIT = 3)
```

produce the same result. Most of the remaining specifications are optional. Exceptions will be pointed out as the discussion proceeds.

2. We may include an error specifier (ERR = *statementnumber*) that will cause an automatic branch to the indicated statement number should FORTRAN find it impossible to open the file. (For example, there is no such unit number as the one specified in the OPEN statement.)

3. An access specifier (ACCESS = *accessmethod*) may be included to indicate how the records will be read or written. the two choices are ACCESS='DIRECT' or ACCESS='SEQUENTIAL'. The latter is assumed if the ACCESS= specifier is omitted.

4. The form of the records also can be specified by including FORM='FORMATTED' or FORM='UNFORMATTED'. If the form is not specified, FORTRAN will assign a form automatically, depending on the access method. For DIRECT files the assumed form is UNFORMATTED; for SEQUENTIAL files the assumed form is FORMATTED.

5. The IOSTAT specifier also is available for use with the OPEN statement.

6. The RECL specifier indicates the record length for the file being opened. This must appear for a direct file and cannot appear for a sequential file. (All the records in a direct file must have the same length.) For example, the statement

```
OPEN (3, IOSTAT=STAT3, ACCESS='DIRECT', FORM='FORMATTED', RECL=120)
```

says, "Connect unit 3 to a direct, formatted file having a record length of 120 characters. Report the result of the operation in an integer variable named STAT3."

7. An additional specifier (BLANK), intended for formatted input files, tells FORTRAN how it should interpret blanks that it finds in formatted numerical input values. The two choices are BLANKS='NULL' and BLANKS='ZERO'. When the former is specified, FORTRAN will ignore all blanks except in the case where an entire numerical field is blank. Then, it will treat that field's value as zero. When BLANKS='ZERO' is specified, blanks other than leading blanks will be treated as zeros. For example, using b to indicate a blank, a numerical formatted input field with the value bbb76bb will be treated as 76 when BLANKS='NULL' is specified and as 7600 when BLANKS='ZERO' is specified. A field containing bbbb will be treated as a value of zero in either case. When the BLANKS specifier is not included, FORTRAN will assume 'NULL'.

8. The FILE specifier is used to connect a unit with a named file. This may be a file created by another program and "known" to the system (but not to this program), or it may be a brand new file that now will be created and named. The form is

FILE = *filename*

where *filename* is a character string whose length and construction are determined by each system's rules for naming files. In many systems, the rules are the same as those for naming FORTRAN variables.

9. The STATUS specifier tells FORTRAN how the file is to be handled. This has nothing to do with the IOSTAT specifier, which reports the outcome of the OPEN operation. There are four alternatives for this specifier:

(1) STATUS = 'NEW'—This creates a new file. When this is specified, the FILE specifier must appear also, giving a name that is new to the system.

(2) STATUS = 'OLD—Here again, FILE must appear when this status is specified, and the name must be recognized by the system.

(3) STATUS = 'SCRATCH'—This indicates that the file is temporary. The FILE specification *cannot* appear when the STATUS is specified as SCRATCH. When this status is given, the file is created and made available for use until a CLOSE statement is issued for this unit. At that time, the unit is not only disconnected, but the file is deleted as well.

(4) STATUS = 'UNKNOWN': —This is the status assumed by FORTRAN when no status is specified by the programmer. The response depends on the particular FORTRAN implementation. In many cases, the system's action will be the same as for a status of 'OLD' if the file is named and the name is recognized; otherwise, it will react as if the status were 'NEW'.

**13.2.3.2 The CLOSE Statement** This statement's basic intent is opposite to that of the OPEN statement: That is, it breaks the connection between a unit and a file. The form is

CLOSE (*closelist*)

Up to four types of specifiers may be included. Of these, the unit specifier (in either form) must appear, and the others are optional:

1. The IOSTAT specifier may be used to report the outcome of the CLOSE operation.
2. As is true with other file processing statements, the ERR specification may be included to provide (and branch to) a designated place in the program should something go wrong.
3. The STATUS specifier is used to define the disposition of the file once it is disconnected. FORTRAN will retain the file if STATUS='KEEP', or it will delete the file if STATUS='DELETE'. However, the programmer must make sure that his or her specifications are consistent. for example, FORTRAN will not accept STATUS = 'KEEP' in a CLOSE statement if that same file was connected earlier with an OPEN statement that said STATUS = 'SCRATCH'.

If a CLOSE statement is issued in which the specified unit has no file connected to it or in which the unit does not exist altogether, FORTRAN will accept the statement but nothing will happen.

Note that the STATUS='KEEP' specification makes it possible to close a file and still have that file exist. Consequently, that file can be reconnected later with another OPEN statement. Of course, the same is true for a unit.

**13.2.3.3 The INQUIRE Statement** All of the input/output statements examined thus far perform some type of operation on a file. In contrast, the INQUIRE statement provides a way of determining the properties of a file or unit without actually doing anything with it. Basically, INQUIRE asks FORTRAN to check on a particular file or unit. Since the purpose is to report the status of that file or unit, the statement must specify a variable in which each finding is to be reported. The general form is

INQUIRE (*inquirylist*)

When we wish to inquire about a particular file, the parenthesized *inquirylist* must include a file name, designated by a FILE = specifier. Thus, for example, the statement

INQUIRE (FILE='PYMSTR', *otherspecs*)

says, "Please report on the properties of the file named PYMSTR. I am interested in those properties listed in *otherspecs*, and I am telling you in which variables to report these properties."

Another way to use the INQUIRE statement is to ask about a certain unit. As you would expect, this form requires a unit specifier instead of a file specifier. For instance, the statement

INQUIRE (3, *otherspecs*)

says, "Please report on the properties of unit 3. I am interested in those properties listed in *otherspecs*." (The UNIT = form also may be used.)

In either case, *otherspecs* may consist of any combination of a wide variety of specifiers. Some of these were discussed with reference to other file-related statements, and they have the same meaning here. Accordingly, we shall just list them. Others are intended specifically for use with the INQUIRE statement, and these will be described briefly.

1. The IOSTAT specifier may be used. A value of zero indicates that the designated file or unit has a consistent set of properties.
2. The ERR specifier may be used as in other statements.
3. A group of specifiers is available to check on some of the fundamental properties.
   (1) The existence of the file or unit is checked by the EXIST specifier. This is done by specifying

   EXIST = *exstat*

   where *exstat* is the name of a variable declared with a LOGICAL statement (Chapter 4). FORTRAN assigns a value of .TRUE. if the specified file or unit exists, .FALSE. if it does not.

   (2) The OPENED specifier works basically the same way. When we say

   OPENED = *opnstat*

   FORTRAN assigns the value .TRUE. to the logical variable specified by *opnstat* if the designated file or unit is connected to something, and it assigns .FALSE. if there is no connection.

   (3) The NUMBER specification may be used when the INQUIRE statement refers to a file. By writing

   NUMBER = *numvar*

   we cause FORTRAN to report (in the INTEGER variable named *numvar*) the number of the unit connected to the file named in the FILE specification. If there is no unit connected to that file, then no one can tell what *numvar*'s value will be.

   (4) The NAMED specifier indicates the name of a logical variable to which FORTRAN assigns .TRUE. if the file specified in the INQUIRE statement has a name, and .FALSE. if it does not.

   (5) The NAME specification enables us to obtain the name of a file (if there is one) connected to a specified unit. Thus, when we say

```
LOGICAL         EXIST, OPTST, NMTST
CHARACTER*10    FILNAM
   .......
   .......
INQUIRE (UNIT=4, EXIST=EXTST, OPENED=
1                                  OPTST, NAMED=NMTST, NAME=FILNAM)
```

   we are making the following request: "I would like to know some things about the status of unit number 4. Specifically, tell me (by reporting in the variable EXTST) whether this system has a unit number 4. Also, let me know (in the variable named OPTST) whether unit 4 has a file connected to it. If that is true, I want to look in variable NMTST to find out whether that file has a name. If it does, I want you to report that name in the character string FILNAM." Once the INQUIRE statement assigns the appropriate values to the specified variables, the program can go on to test for those values and take some action based on what it finds.

4. Another group of related specifiers enables the programmer to get information about the organization of a file.

(1) The ACCESS specifier gives the name of a character variable where FORTRAN is to report the access method (DIRECT or SEQUENTIAL) for which the file or unit is connected. If the file or unit is not connected at the time the INQUIRE statement is executed, then no one can say what the value will be in that variable. As the next two specifiers indicate, it is possible to check on the access in more detail.

(2) The form

SEQUENTIAL = *seqvar*

enables the programmer to determine whether the specified file or unit is available for sequential access. This is different from the ACCESS specifier because it indicates explicitly whether sequential access is allowed. This is reported by automatically assigning a value of YES to the character variable named by *seqvar*. Note that a value of YES does not necessarily rule out the allowance of direct access for that file or unit. If sequential access is not allowed, FORTRAN places the character string NO in that variable. A third possible value, UNKNOWN, is assigned when FORTRAN cannot determine whether sequential access is allowed.

(3) The DIRECT specifier provides the same service with regard to direct access. In response to the specification

DIRECT = *dirvar*

FORTRAN places a value of YES in the character variable named by *dirvar* if the designated file or unit allows direct access, NO if it does not, or UNKNOWN if it cannot find out. As is true with the SEQUENTIAL specification, a value of YES in *dirvar* does not automatically imply that the file or unit does not allow sequential access.

5. Several additional specifiers enable us to produce information about the form of the records and about certain aspects of their contents.

(1) The most general specifier in this category is the FORM specifier. Its meaning here is the same as was described for the OPEN statement. Thus, by specifying

FORM = *formvar*

FORTRAN will report in the character variable named in *formvar* the form (FORMATTED or UNFORMATTED) for the designated file or unit. If the file or unit is not connected, no one can determine the resulting value in *formvar* (that is, the value is undefined).

(2) More detailed information about the file's form can be obtained with the FORMATTED and UNFORMATTED specifiers. These are not to be confused with the *values* developed by FORTRAN in response to a FORM specifier. Each of these is a separate specifier with its own designated character variable. For instance, the specification

FORMATTED = *fmtvar*

causes FORTRAN to deliver a value of YES in the variable named by *fmtvar* if the file or unit allows formatted records, NO if it does not, or UNKNOWN if FORTRAN cannot find out. Similarly, the specification

UNFORMATTED = *unfvar*

causes FORTRAN to deliver a value of YES, NO, or UNKNOWN in the variable named by *unfvar* that describes the status of the unit or file with regard to allowing unformatted records. The fact that a file allows unformatted records does not necessarily mean that it automatically disallows formatted records. The same is true for a file or unit that allows formatted records.

(3) Information about the record length is obtained by using the RECL specifier along with an integer variable in which FORTRAN will report the value. If the file's records are formatted, the record length will be in characters. For unformatted records, the way the value is expressed will depend on the particular type of computer being used. If the file or unit is not connected, or if it does not allow direct access, the value for the record length will be undefined.

(4) The NEXTREC specifier enables the programmer to determine a file's position. By specifying

NEXTREC = *nxtvar*

we request FORTRAN to indicate in the integer variable named by *nxtvar* the position of the next record in the designated file (or in the unnamed file connected to the designated unit). If the file is positioned at its beginning, FORTRAN will report a value of 1. If the file is not connected for direct access (or not connected at all), the value reported in *nxtvar* is undefined.

(5) The BLANK specifier has the same basic use as in the OPEN statement. When we say

BLANK = *blnkvar*

FORTRAN will report (in the character variable named by *blnkvar*) the way blanks are used in the designated file or unit. The values delivered to the variable are NULL or ZERO, and these have the same meanings as described for the OPEN statement. If the file is not connected, or if it is not connected for formatted input/output, *blnkvar*'s value will be undefined.

## 13.3 EXAMPLE PROGRAMS

Having introduced FORTRAN's file handling capabilities, we shall put some of these ideas to use in two example programs. The first of these uses input data to build a file on magnetic disk. The second program uses that file in conjunction with new input data to produce an *updated* version of the file, i.e., a new version that incorporates the changes introduced by the input data.

**Example 13.1** In a burst of adventure and high confidence, three young hotshots have resigned their positions at the Windowledge Brokerage to form their own mutual fund company. Clients would be sought and asked to invest in the Canine Fund, a collection of carefully watched stocks with great promise. Because of their impressive performance at Windowledge, the three hotshots have been able to start their business with an established list of clients.

As one of its services, the Fund will send to each of its clients a monthly statement showing how many shares that client owns. To get this process started, a program is needed to build a file of client data. Information for each subscriber is available initially in list-directed form on a punched card containing the following items:

| | |
|---|---|
| subscriber's name | (25 characters or fewer) |
| subscriber's account number | (a 6-digit integer from 1–999000) |
| dollar amount of initial investment | (given to the nearest cent) |

Information from each of these records is to produce two types of output:

1. A disk record in list-directed format consisting of the account number, name, the date (month/day/year) the record was created, the type of transaction (a purchase in this case), and the number of shares (to the nearest thousandth of a share) owned by that subscriber.

```
CANINE FUND - STATUS REPORT

ACCT:  120454          NAME:   MERVYN W. BUMBLE

DATE       TRANS      AMT INV       PRICE        NO. SHARES

11/12/80   PUR    4.E+03  10.237    390.739
```

**FIGURE 13.5** Sample Output for Example 13.1.

2. A written report (on a separate page for each subscriber) containing the subscriber's name and account number, the date the record was created, the type of transaction (a purchase in this case), the dollar amount of the initial investment, the price per share at the time of investment, and the number of shares purchased by that initial amount. An example of such a page is shown in Figure 13.5. Note that the real values appear either in floating point form or conventional form. The HP FORTRAN 77 compiler determines the form for each particular real value based on its magnitude. A few experiments with different values will establish the exact rules for your implementation. (Be sure to include values with large negative exponents as well as those having large positive exponents.)

To keep our attention focused on the file building process itself, we shall keep things simple by assuming that the cards are in order by ascending account number (lowest account number first, and so on), and there are no errors in the data. We shall assume further that the output file is to be attached to unit 2. Units 5 and 6 are assumed to be the system's input and output units, respectively.

The implementation is quite simple. In addition to the actual file processing, the program will compute and print the number of subscribers (SUBSCRIBERS), the total dollar amount invested (TOTAL_INVESTED), and the total number of shares purchased (TOTAL_SHARES). Accordingly, the following program structure suggests itself:

1. The stage is set by a subroutine (INIT) that initializes SUBSCRIBERS and TOTAL_INVESTED, opens the output file on unit 2, and reads a single line containing today's date (DATE) and price per share (PRICE).

2. The major processing cycle consists of a DO-WHILE construct that prepares an output record for each subscriber's input card and uses a subroutine (PRTSUB) to produce the output page for each subscriber. (Note that the test that begins the DO-WHILE is included (as the END= specifier) in the READ statement.)

3. After all the records have been read and written, a concluding part of the program (a subroutine named SUMUP) will close the output file and print the summary information.

As part of the summary process, SUMUP will produce a dummy output record with an account number of 999999. This record, placed between the last subscriber's record and the ENDFILE record, is a way of safeguarding a sequential file when it is used for subsequent processing. The technique of ending a file with a record having the largest possible sequence number is called *bounding*. By doing this, it becomes much easier to protect programs against input data that are improperly sequenced. Although we said that this is not a problem in this example, the technique is a useful one to know about. The program for this example is shown in Figure 13.6.

**Example 13.2** We shall expand the Canine Fund's services by enabling subscribers to increase or decrease their holdings by sending in money or requesting money. The basic process may be described as follows: Each day, subscribers' requests are prepared by recording one request per line arranged in order by account number. These data are preceded by a line containing that day's date and the day's average price for a share of the Canine Fund. That price is used to convert the dollar amount of each transaction to an equivalent number of shares. Thus, a purchase or withdrawal of so many dollars is expressed as some number of shares, and that number (rounded to the nearest thousandth of a share) is added to or subtracted from the subscriber's total. In either case, a new list-directed file record is produced showing the latest information: date of transaction; type of transaction (PUR for purchase, WTH for withdrawal); amount of the transaction; price per share at which the purchase or withdrawal was made; and the new total number of shares. (Of course, the name and account number are included in the

```
C*********************************************************
C                   EXAMPLE 13.1                          *
C*********************************************************
C   TOTAL_INVESTED:    TOTAL AMOUNT INVESTED              *
C   TOTAL_SHARES:    TOTAL SHARES PURCHASED               *
C   PRICE:      TODAY'S PRICE PER SHARE                   *
C   AMTINV:     AMOUNT INVESTED BY A SUBSCRIBER           *
C   SHARES:     SHARES PURCHASED BY A SUBSCRIBER          *
C   SUBSCRIBERS:    NUMBER OF SUBSCRIBERS                 *
C   DATE:       TODAY'S DATE (MM/DD/YY)                   *
C   NAME:       SUBSCRIBER'S NAME                         *
C   ACCTNO:     SUBSCRIBER'S ACCOUNT NUMBER               *
C*********************************************************
      PROGRAM          EX1301
      IMPLICIT         NONE
      REAL             TOTAL_INVESTED, TOTAL_SHARES,
     1                 PRICE, AMTINV, SHARES
      INTEGER*2        ACCTNO, NUMSHR
      CHARACTER        NAME*25, DATE*8
      LOGICAL          WE_HAVE_DATA

      CALL INIT (SUBSCRIBERS, TOTAL_INVSETED, TOTAL_SHARES,
     1           DATE, PRICE)
      DO WHILE (WE_HAVE_DATA)
        READ (5,FMT=*,END=99) ACCTNO, NAME, AMTINV
        SUBSCRIBERS = SUBSCRIBERS+1
        TOTAL_INVESTED = TOTAL_INVESTED + AMTINV

C     NOW WE SHALL COMPUTE THE NO. OF SHARES, ROUNDED TO THE NEAREST
C     THOUSANDTH OF A SHARE. THE TECHNIQUE TO BE USED IS THE ONE
C     DESCRIBED SO BEAUTIFULLY IN SECTION 6.2.5.4.

        SHARES = AMTINV/PRICE
        SHARES = ANINT(1000.0*SHARES)/1000.0
        TOTAL_SHARES = TOTAL_SHARES + SHARES
        WRITE (2,FMT=*) ACCTNO, NAME, DATE, 'PUR', SHARES
        CALL PRTSUB (ACCTNO, NAME, DATE, AMTINV, PRICE, SHARES)
      END DO

   99 CALL SUMUP (DATE, SUBSCRIBERS, TOTAL_INVESTED,
     1            TOTAL_SHARES, ,NAME)
      PRINT *,'     '
      PRINT *,'END OF RUN.'
      STOP

      END
```

**FIGURE 13.6 (a)** The Overall Program for Example 13.1.

record as before.) A one-page statement (similar to the one in Figure 13.5) is printed for each subscriber involved in a transaction. The program also produces and prints an overall summary showing the total number of purchases and withdrawals, along with the dollar amounts and equivalent share amounts for each of the totals.

```
C*******************************************************
C                      INIT                             *
C*******************************************************
C  THIS SUBROUTINE INITIALIZES THE TOTALS, COUNTER,     *
C  OPENS THE OUTPUT FILE ON UNIT 2, AND READS THE       *
C  DATE AND SHARE PRICE.                                *
C*******************************************************
C  TOTAL_INVESTED:       TOTAL AMOUNT INVESTED          *
C  TOTAL_SHARES:     TOTAL NUMBER OF SHARES PURCHASED   *
C  SUBSCRIBERS:        NUMBER OF SUBSCRIBERS            *
C  DATE:          TODAY'S DATE, MM/DD/YY                *
C  PRICE:         TODAY'S PRICE $XX.XXX PER SHARE       *
C*******************************************************
      SUBROUTINE          INIT (SUBSCRIBERS, TOTAL_INVESTED,
     1                          TOTAL_SHARES, DATE, PRICE)
      IMPLICIT            NONE
      REAL                TOTAL_INVESTED, TOTAL_SHARES, PRICE
      INTEGER*2           SUBSCRIBERS
      CHARACTER*8         DATE

      SUBSCRIBERS = 0
      TOTAL_INVESTED = 0.0
      TOTAL_SHARES = 0.0
      OPEN (2,ACCESS='SEQUENTIAL',FORM='FORMATTED')
      READ *, DATE, PRICE

      RETURN
      END
```

**(b)**

```
C*******************************************************
C                     PRTSUB                            *
C*******************************************************
      SUBROUTINE       PRTSUB (ACCTNO,NAME,DATE,AMTINV,PRICE,SHARES)
      IMPLICIT         NONE
      REAL             AMTINV,PRICE,SHARES
      INTEGER*2        ACCTNO
      CHARACTER        NAME*25,DATE*8,BLANKS*6
      PARAMETER        (BLANKS='      ')

C     AN EDIT-DIRECTED WRITE STATEMENT WILL START A NEW PAGE.

      WRITE (6,16)
   16 FORMAT ('1')

      PRINT *,BLANKS
      PRINT *,BLANKS
      PRINT *,BLANKS
      PRINT *,'CANINE FUND - STATUS REPORT'
      PRINT *,'ACCT: ',ACCTNO,'NAME:  ',NAME
      PRINT *,BLANKS
      PRINT *,BLANKS
      PRINT *,BLANKS
      PRINT *,'DATE   TRANS   AMT INV   PRICE   # SHARES'
      PRINT *,BLANKS
      PRINT *,DATE,'PUR',AMTINV,PRICE,SHARES
      RETURN

      END
```

**(c)**



**FIGURE 13.6 (b)** INIT Subroutine for Example 13.1. **(c)** PRTSUB Subroutine for Example 13.1.

```
C*******************************************************
C                       SUMUP                          *
C*******************************************************
      SUBROUTINE        SUMUP (DATE,SUBSCRIBERS, TOTAL_INVESTED,
     1                         TOTAL_SHARES, NAME)
      IMPLICIT          NONE
      REAL              TOTAL_INVESTED, TOTAL_SHARES, ZEROS
      INTEGER*2         SUBSCRIBERS,
      INTEGER*4         BOUND
      CHARACTER         DATE*8, NAME*25, SPACES*5
      PARAMETER         (SPACES='     ',BOUND=999999,ZEROS=0.0)

      NAME = SPACES
      WRITE (2,FMT=*) BOUND, NAME, DATE, ZEROS
      WRITE (6,26)
   26 FORMAT ('1')
      PRINT *,SPACES
      PRINT *,SPACES
      PRINT *,SPACES
      PRINT *,'CANINE FUND - SUMMARY REPORT'
      PRINT *,'DATE: ',DATE
      PRINT *,SPACES
      PRINT *,'NO. OF SUBSCRIBERS: ', SUBSCRIBERS
      PRINT *,'NO. OF SHARES PURCHASED: ', TOTAL_SHARES
      PRINT *,'TOTAL AMOUNT INVESTED: ', TOTAL_INVESTED
      ENDFILE (2)
      CLOSE (2)
      RETURN
      END
```

**FIGURE13.6 (d)** SUMUP Subroutine for Example 13.1.

To maintain our concentration on the file-related processes, we shall make some simplifying assumptions as we did in the previous example:

1. The input data are guaranteed to be in proper sequence.
2. All input transactions will be consistent and "legal." For example, there will be no attempt to withdraw more money than a subscriber has in his or her account, nor will there be an attempt to conduct a transaction on a nonexistent account number.

Now that the requirements have been defined, we can turn our attention to converting these requirements into a well-organized program. Note that the file described above contains the same type of information developed by the file creating program in Example 13.1. As a result, we can meet these requirements by designing our program so that it uses that file as input to produce a brand new output file containing all the subscribers' records. Those for whom a purchase or withdrawal was made will have updated information; the other records merely will be copied exactly as they were on the input file. thus, a file produced as output from one run can serve as input for the next, and so on. This overall flow of events is show in Figure 13.7. Though somewhat simplified, it exemplifies the type of updating process that finds frequent use in a wide variety of applications. Since each updating run produces a new copy of the data, the previous copy (i.e., the input for that run) provides a built-in safety feature. It still is available should something go wrong and the run has to be repeated. For this reason, many installations make it a standard practice to save such data for several runs back before they are considered old enough so that it is safe to put the reel or tape or the space on the disk to some other use.

Based on these considerations, we see that the program requires two input files: the subscribers' records (which we shall connect to unit 4), and the records containing the day's transactions (which automatically will be preconnected to the system input unit). Since each of these files will be arranged in

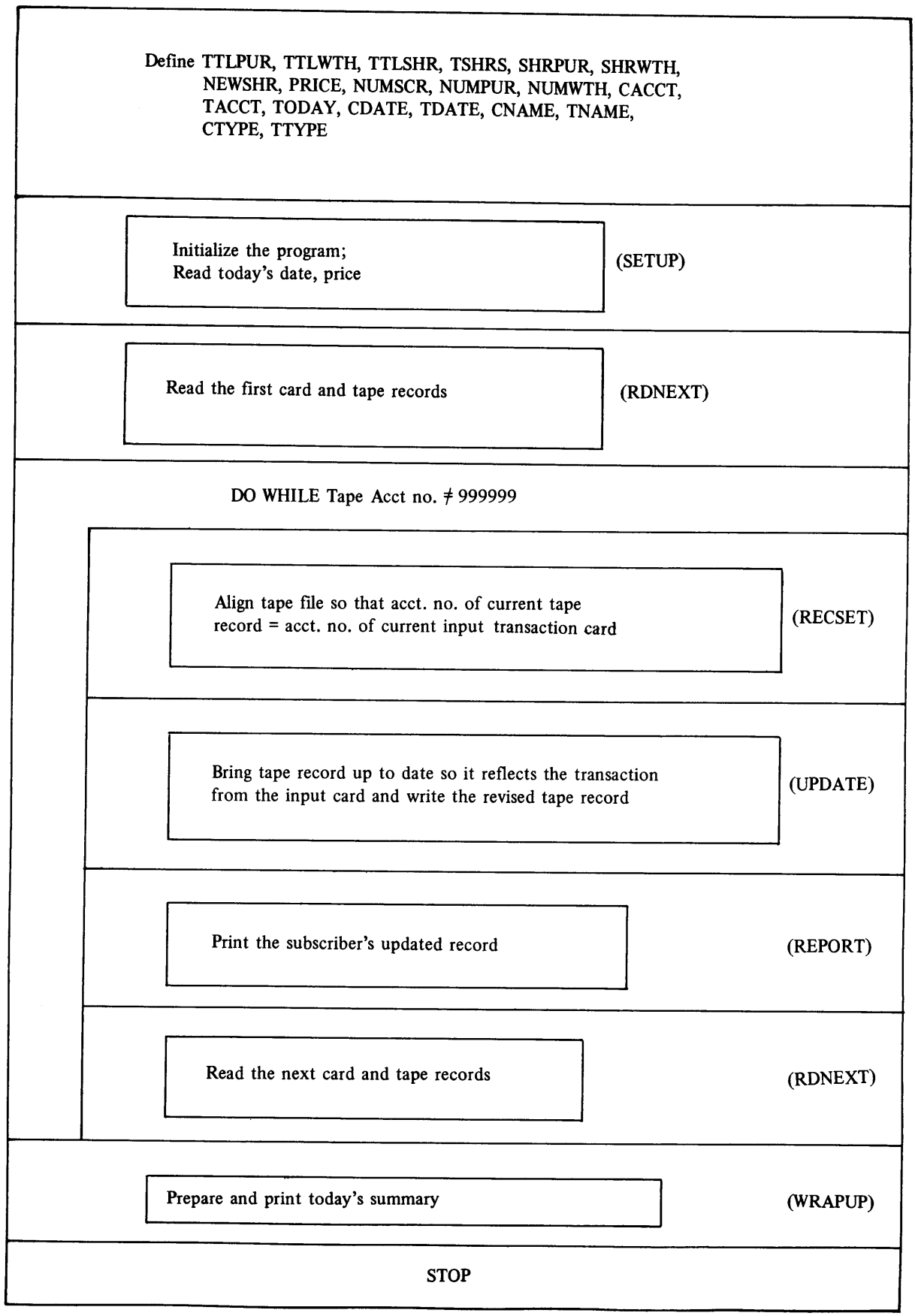
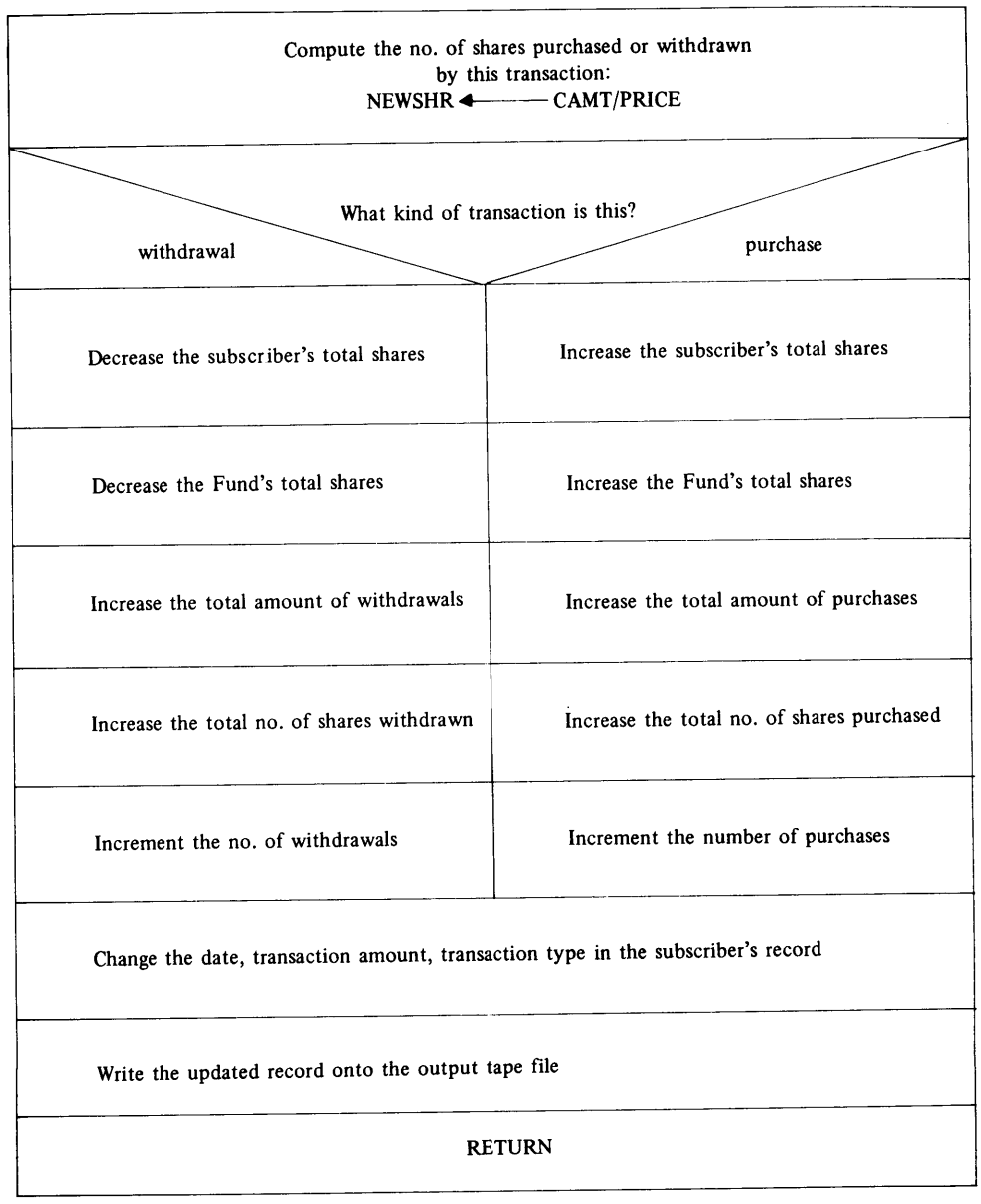





**FIGURE 13.7 (a)** Information Flow for Example 13.2.

```
"Define TOTAL_PURCHASED, TOTAL_WITHDRAWN, TOTAL_SHARES,
        SHARES_OWNED, SHARES_PURCHASED, SHARES_WITHDRAWN,
        SHARES_PROCESSED, PRICE, SUBSCRIBERS,
        NUM_PURCHASES, NUM_WITHDRAWLS, INPUT_ACCT,
        FILE_ACCT, TODAY, INPUT_DATE, FILE_DATE,
        NAME_, NAME_ONFILE, INPUT_TRANS, FILE_TRANS. ' '
"Initialize the program."
"Read today's date and price."
"Read the first transaction and disk records."
while disk account no. is not 999999:
    "Align disk and transaction records so account numbers match."
    "Update disk record to reflect the most recent transaction;
     write the revised disk record."
    "Print the updated subscriber's record."
    "Read the next transaction and disk records."
    endwhile
    "Prepare and print today's summary."
    "Stop."
```

**(b)**




**(c)**

**FIGURE 13.7** **(b)** Pseudocode for Example 13.2. **(c)** N-S Diagram for Example 13.2.

```
"Compute the no. of shares involved in this transaction:
    NEWSHR = CAMT/PRICE."
if
    transaction is a purchase
then
    "Increase subscriber's total shares."
    "Increase the Fund's total shares issued."
    "Increase the total purchase amount."
    "Increase the total no. of shares purchased."
    "Increment the total number of purchases."
else
    "Decrease the subscriber's total shares."
    "Decrease the Fund's total shares."
    "Increase the total withdrawal amount."
    "Increase the total no. of shares withdrawn."
    "Increment the total number of withdrawals."
endif
"Change the date, transaction amount, transaction type in
  the subscriber's record."
"Write the updated record onto the output file."
"Return."
```

**FIGURE 13.7** **(d)** Pseudocode for UPDATE Subroutine for Example 13.2.

the same order, we can anticipate the need for some processing that will make sure that the transaction for a given account is matched against the master record for the same account. Other organizational characteristics are not very different from those of the previous example, so we can summarize the program's construction as follows:

1. A subroutine named SETUP will be used at the beginning of each run to initialize the variables where total values and counters will be accumulated. In addition, the routine will read the day's date and price, as well as the first subscriber's record from each input file.

2. A separate subroutine named RECSET will handle the processing to make sure that the input records from the two units presented to the next part of the program each belong to the same account. To do this, it will read and copy as many subscribers' records as necessary, comparing each one with the current transaction record in turn, until the match is found. Since we assumed that the transactions are in proper order, this process is fairly simple. However, we can imagine that it could become quite intricate when we have to watch for and take care of records that are out of sequence, as well as other possible error conditions. (You will be challenged to do some of this in the problems at the end of the chapter.)

3. Once a matching pair of input records has been provided, another subroutine (named UPDATE) will use the data from the transaction to prepare a modified version of the subscriber's record.

4. Output of the updated record, as well as the preparation of the printed report for that subscriber, will be handled by a subroutine named REPORT.

5. Finally, a simple routine (named RDNEXT) will read the next record from each of the input files, so that the program now will be ready for another cycle. Note that no checking of account numbers is done at this point, since the RECSET subroutine already is designed to do this.

The overall program design is shown in Figure 13.7 and the program itself is given in Figure 13.8. It should be pointed out that in an actual situation, it is more likely that the subscribers' master records would be unformatted. However, this small departure from reality will not detract from the file processing itself.

## 13.4 SUMMARY

When FORTRAN transmits data to or from the central processor, the individual data values are grouped into *records* and the records are grouped further into *files*. These files may be organized in different ways, thereby determining how the values are represented, in what form the records are constructed, and in what order the records may be obtained by or sent from the processor. A summary of these organizational characteristics is given in Table 13.1.

```
C*************************************************************
C             EXAMPLE 13.2 - THE MAIN PROGRAM                 *
C*************************************************************
C  THIS PROGRAM PROCESSES SUBSCRIBERS' TRANSACTIONS AGAINST *
C  THEIR RECORDS IN THE CANINE FUND. EACH TRANSACTION       *
C  SPECIFIES THE AMOUNT OF MONEY TO BE ADDED TO ('PUR') OR  *
C  WITHDRAWN FROM ('WTH') A SUBSCRIBER'S HOLDINGS. THE PRO- *
C  GRAM PRODUCES A NEW DISK RECORD FOR EACH SUBSCRIBER      *
C  WHICH IS A COPY OF THE PREVIOUS VERSION (IF THERE WAS NO *
C  TRANSACTION) OR AN UPDATED VERSION SHOWING THE RESULTS OF*
C  THE PURCHASE OR WITHDRAWAL. VARIABLE NAMES ARE:          *
C   SUBSCRIBERS:   TOTAL NUMBER OF SUBSCRIBERS              *
C   NUM_PURCHASES, NUM_WITHDRAWLS: NO. MAKING               *
C                PURCHASES, WITHDRAWALS                     *
C   TOTAL_PURCHASED, TOTAL_WITHDRWAN: TOTAL AMTS OF         *
C             PURCHASE, WITHDRAWAL                          *
C   TOTAL_SHARES:        TOTAL NO. OF SHARES IN CANINE      *
C   SHARES_PURCHASED, SHARES_WITHDRAWN: NO. OF SHARES       *
C           PURCHASED, WITHDRAWN                            *
C   SHARES_PROCESSED:        NO. OF SHARES IN A TRANSACTION *
C   TODAY:          TODAY'S DATE (MM/DD/YY)                 *
C   INPUT_DATE, FILE_DATE:   DATE ON AN INPUT LINE,         *
C                  DISK RECORD                              *
C   INPUT_ACCT, FILE_ACCT:   ACCOUNT NO. ON AN INPUT LINE,  *
C                              DISK RECORD                  *
C   NAME_IN, NAME_ONFILE:   NAME ON AN INPUT LINE,          *
C                             DISK RECORD                   *
C   INPUT_TRANS, FILE_TRANS:   TRANSACTION TYPE             *
C                               ('PUR' OR 'WTH') ON A RECORD*
C   INPUT_AMT:      TRANACTION AMOUNT SPECIFIED ON A LINE   *
C   FILE_AMT:       TRANSACTION AMOUNT SPECIFIED ON DISK    *
C   SHARES_OWNED  NO. OF SHARES A SUBSCRIBER OWNS           *
C   PRICE:          TODAY'S PRICE PER SHARE                 *
C*************************************************************

      $FILES (0,2)
      PROGRAM         EX1302
      IMPLICIT        NONE
      REAL            TOTAL_PURCHASED, TOTAL_WITHDRAWN, TOTAL_SHARES,
     1                SHARES_PURCHASED, SHARES_WITHDRAWN,
     2                SHARES_PROCESSED, INPUT_AMT, FILE_AMT,
     3                SHARES_OWNED, PRICE
      INTEGER*4       INPUT_ACCT, FILE_ACCT
      INTEGER*2       SUBSCRIBERS, NUM_PURCHASES, NUM_WITHDRAWLS
      CHARACTER       TODAY*8, INPUT_DATE*8, FILE_DATE*8,
     1                NAME_IN*25, NAME_ONFILE*25,
     2                INPUT_TRANS*3, FILE_TRANS*3

      CALL SETUP (TOTAL_PURCHASED, TOTALWITHDRAWN, TOTAL_SHARES,
     1            SHARES_PURCHASED, SHARES_WITHDRAWN, SUBSCRIBERS,
     2            NUM_PURCHASES, NUM_WITHDRAWLS, TODAY, PRICE)
      CALL RDNEXT (SHARES_OWNED, FILE_AMT, INPUT_AMT,
     1             FILE_ACCT, INPUT_ACCT, FILE_DATE, INPUT_DATE,
     2             NAME_ONFILE, NAME_IN, FILE_TRANS, INPUT_TRANS)
      DO WHILE (FILE_ACCT .NE. 999999)
        CALL RECSET (TOTAL_SHARES, SHARES_OWNED, FILE_AMT,
     1               SUBSCRIBERS, FILE_ACCT, INPUT_ACCT,
     2              FILE_DATE, NAME_ONFILE, FILE_TRANS)
        CALL UPDATE (TOTAL_PURCHASED, TOTAL_WITHDRAWN, TOTAL_SHARES,
     1               SHARES_PURCHASED, SHARES_WITHDRAWN,
     2               SHARES_PROCESSED, SHARES_OWNED, FILE_AMT,
     3               INPUT_AMT, PRICE, SUBSCRIBERS, NUM_PURCHASES,
     4               NUM_WITHDRAWLS, FILE_ACCT, TODAY, FILE_DATE,
     5               NAME_ONFILE, FILE_TRANS, INPUT_TRANS)
        CALL REPORT (SHARES_OWNED, SHARES_PROCESSED, FILE_AMT, PRICE,
     1               FILE_ACCT, NAME_ONFILE, FILE_DATE, FILE_TRANS)
        CALL RDNEXT (SHARES_OWNED, FILE_AMT, INPUT_AMT,
     1               FILE_ACCT, INPUT_ACCT, FILE_DATE, INPUT_DATE,
     2              NAME_ONFILE, NAME_IN, FILE_TRANS, INPUT_TRANS)
      END DO

   99 CALL WRAPUP (TOTAL_PURCHASED, TOTAL_WITHDRAWN, TOTAL_SHARES,
     1             SHARES_PURCHASED, SHARES_WITHDRAWN, PRICE,
     2             SUBSCRIBERS, NUM_PURCHASES, NUM_WITHDRAWLS, TODAY)
      STOP
      END
```

**FIGURE 13.8 (a)** Main Program for Example 13.2.

```
C*************************************************************
C                          SETUP                              *
C*************************************************************
C THIS SUBROUTINE INITIALIZES THE PROGRAM BY SETTING THE     *
C COUNTERS AND TOTALS TO ZERO AND OPENING THE TWO DISK FILES*
C ON UNITS 4 AND 2.                                          *
C*************************************************************
      SUBROUTINE    SETUP (TOTAL_PURCHASED, TOTAL_WITHDRAWN, TOTAL_SHARES,
     1                     SHARES_PURCHASED, SHARES_WITHDRAWN,
     2                     SUBSCRIBERS, NUM_PURCHASES, NUM_WITHDRAWLS,
     3                     TODAY, PRICE)
      IMPLICIT  NONE
      REAL      TOTAL_PURCHASED, TOTAL_WITHDRAWN, TOTAL_SHARES,
     1          SHARES_PURCHASED, SHARES_WITHDRAWN, PRICE
      INTEGER*2 SUBSCRIBERS, NUM_PURCHASES, NUM_WITHDRAWLS
      CHARACTER TODAY*8

      TOTAL_PURCHASED = 0.0
      TOTAL_WITHDRAWN = 0.0
      TOTAL_SHARES = 0.0
      SHARES_PURCHASED = 0.0
      SHARES_WITHDRAWN = 0.0
      SUBSCRIBERS = 0
      NUM_PURCHASES = 0
      NUM_WITHDRAWLS = 0
      READ *, TODAY, PRICE
      OPEN (4,FMT='FORMATTED',ACCESS='SEQUENTIAL')
      OPEN (2,FMT='FORMATTED',ACCESS='SEQUENTIAL')

      RETURN
      END
```

**(b)**

```
C*************************************************************
C                          RDNEXT                             *
C*************************************************************
C  THIS SUBROUTINE MERELY READS THE NEXT LINE AND THE NEXT   *
C  RECORD FROM THE INPUT FILE (UNIT 4). IF THERE ARE NO MORE*
C  TRANSACTIONS, INPUT_ACCT IS SET TO 999999 AND RETURNS.    *
C*************************************************************

      SUBROUTINE     RDNEXT (SHARES_OWNED, FILE_AMT, INPUT_AMT,
     1                       FILE_ACCT, INPUT_ACCT, FILE_DATE,
     2                       INPUT_DATE, NAME_ONFILE, NAME_IN,
     3                       FILE_TRANS, INPUT_TRANS)
      IMPLICIT       NONE
      REAL           SHARES_OWNED, FILE_AMT, INPUT_AMT
      INTEGER*4      FILE_ACCT, INPUT_ACCT
      CHARACTER*8    (FILE_DATE, INPUT_DATE)*8,
      CHARACTER*7    (NAME_ONFILE, NAME_IN)*25,
      CHARACTER*7    (FILE_TRANS, INPUT_TRANS)*3

      READ (4,FMT=*) FILE_ACCT, NAME_ONFILE, FILE_DATE,
     1               FILE_TRANS,FILE_AMT, SHARES_OWNED
      READ (1,FMT=*,END=19) INPUT_ACCT, NAME_IN, INPUT_TRANS,
     1                      INPUT_AMT
      RETURN
   19 INPUT_ACCT = 999999
      RETURN
      END
```

**(c)**

**FIGURE 13.8** **(b)** SETUP Subroutine for Example 13.2. **(c)** RDNEXT Subroutine for Example 13.2.

```
C*************************************************************
C                          RECSET                             *
C*************************************************************
C  THIS SUBROUTINE SYNCHRONIZES THE DISK AND LINE INPUT     *
C  RECORDS SO THAT THE ONES CURRENTLY IN THE PROCESSOR       *
C  BELONG TO THE SAME ACCOUNT. IT DOES THIS BY READING AND   *
C  COPYING AS MANY DISK RECORDS AS NECESSARY (UPDATING THE   *
C  OVERALL TOTALS AS IT GOES) UNTIL THE ACCOUNT NUMBERS      *
C  MATCH. IF THERE ARE NO MORE INPUT LINES AGAINST WHICH TO  *
C  COMPARE, RECSET WILL "KNOW" THIS BECAUSE THE RDNEXT SUB-  *
C  ROUTINE WILL HAVE SET INPUT_ACCT TO 999999. AS LONG AS    *
C  INPUT_ACCT IS NOT 999999, THE ACCOUNT NUMBERS EITHER WILL *
C  MATCH, OR RECSET WILL CONTINUE TO WRITE AND READ DISK     *
C  RECORDS UNTIL SUCH A MATCH OCCURS. EVEN IF THE MATCH      *
C  OCCURS AT ACCOUNT NO. 999999, IT IS JUST ANOTHER MATCH AS*
C  FAR AS RECSET IS CONCERNED; ITS SPECIAL NATURE IS NOTED   *
C  AND HANDLED IN THE MAIN PROGRAM, NOT HERE.                *
C*************************************************************
      SUBROUTINE RECSET (TOTAL_SHARES, SHARES_OWNED, FILE_AMT,
     1                   SUBSCRIBERS, FILE_ACCT, INPUT_ACCT, FILE_DATE,
     2                   NAME_ONFILE, FILE_TRANS)
      IMPLICIT    NONE
      REAL        TOTAL_SHARES, SHARES_OWNED, FILE_AMT
      INTEGER*4   FILE_ACCT, INPUT_ACCT
      INTEGER*2  SUBSCRIBERS
      CHARACTER   TDATE*8, NAME_ONFILE*25, FILE_TRANS*3

      DO WHILE (FILE_ACCT .NE. INPUT_ACCT)
        TOTAL_SHARES = TOTAL_SHARES + SHARES_OWNED
        WRITE (2,FMT=*) FILE_ACCT, NAME_ONFILE, FILE_DATE,
     1                  FILE_TRANS, FILE_AMT, SHARES_OWNED
        READ  (4,FMT=*) FILE_ACCT, NAME_ONFILE, FILE_DATE,
     1                  FILE_TRANS, FILE_AMT, SHARES_OWNED
      END DO

      RETURN
      END
```

*(Continued)*

**FIGURE 13.8 (d)** RECSET Subroutine for Example 13.2.

A file is said to *exist* when its organizational properties (and its optional name) are made known to the system on which it will be used. The fact that a file exists does not necessarily mean that it is available to any program that wants it. This availability is established by *connecting* a file to a *unit*, thereby associating that file with a physical device from which the records are to be *read* or onto which the records are to be *written*.

There are nine FORTRAN statements for dealing with files:

1. READ:—transmission of records from a file to the processor.
2. WRITE:—transmission of records from a processor to a file.
3. PRINT:—a special case of WRITE in which the records are to be displayed on a printing device.
4. BACKSPACE:—repositioning of a file to the previous record in its physical sequence.

```
C*************************************************************
C                          UPDATE                             *
C*************************************************************
C  UPDATE USES THE DATA FROM THE CURRENT INPUT LINE TO PRO-  *
C  DUCE NEW FIGURES FOR THAT ACCOUNT. THESE ARE WRITTEN ON   *
C  THE OUTPUT DISK AND ADDED TO THE APPROPRIATE SUMMARY      *
C  VALUES.                                                   *
C*************************************************************
      SUBROUTINE UPDATE (TOTAL_PURCHASED, TOTAL_WITHDRAWN,
     1                   TOTAL_SHARES, SHARES_PURCHASED,
     2                   SHARES_WITHDRAWN, SHARES_PROCESSED,
     3                   SHARES_OWNED, FILE_AMT, INPUT_AMT, PRICE,
     4                   SUBSCRIBERS, NUM_PURCHASES, NUM_WITHDRAWLS,
     5                   FILE_ACCT, TODAY, FILE_DATE,
     6                   NAME_ONFILE, FILE_TRANS, INPUT_TRANS)
      IMPLICIT      NONE
      REAL          TOTAL_PURCHASED, TOTAL_WITHDRAWN, TOTAL_SHARES,
     1              SHARES_PURCHASED,SHARES_WITHDRAWN,
     2              SHARES_PROCESSED, SHARES_OWNED,
     3              FILE_AMT, INPUT_AMT, PRICE
      INTEGER*4     FILE_ACCT, INPUT_ACCT
      INTEGER*2     SUBSCRIBERS, NUM_PURCHASES, NUM_WITHDRAWLS
      CHARACTER     (FILE_DATE,TODAY)*8,NAME_ONFILE*25,
     1              (FILE_TRANS, INPUT_TRANS)*3

      SHARES_PROCESSED = INPUT_AMT/PRICE
      SHARES_PROCESSED = ANINT(1000.0*SHARES_PROCESSED)/1000.0
      IF (INPUT_TRANS .EQ. 'PUR') THEN
        SHARES_OWNED = SHARES_OWNED + SHARES_PROCESSED
        TOTAL_SHARES = TOTAL_SHARES + SHARES_OWNED
        TOTAL_PURCHASED = TOTAL_PURCHASED + INPUT_AMT
        SHARES_PURCHASED = SHARES_PURCHASED + SHARES_PROCESSED
        NUM_PURCHASES = NUM_PURCHASES + 1
      ELSE
        SHARES_OWNED = SHARES_OWNED - SHARES_PROCESSED
        TOTAL_SHARES = TOTAL_SHARES + SHARES_OWNED
        TOTAL_WITHDRAWN = TOTAL_WITHDRAWN + SHARES_PROCESSED
        SHARES_WITHDRAWN = SHARES_WITHDRAWN + SHARES_PROCESSED
        NUM_WITHDRAWLS = NUM_WITHDRAWLS + 1
      END IF
      FILE_DATE = TODAY
      FILE_TRANS = INPUT_TRANS
      FILE_AMT = INPUT_AMT
      WRITE (2,FMT=*) FILE_ACCT, NAME_ONFILE, FILE_DATE,
     1                FILE_TRANS, FILE_AMT, SHARES_OWNED
      SUBSCRIBERS = SUBSCRIBERS + 1

      RETURN
      END
```

*(Continued)*

**FIGURE 13.8 (e)** UPDATE Subroutine for Example 13.2.

```
C*************************************************************
C                          REPORT                             *
C*************************************************************
C  THIS SUBROUTINE PRINTS THE REPORT FOR EACH SUBSCRIBER     *
C  WHOSE ACCOUNT HAS BEEN AFFECTED BY A TRANSACTION.         *
C*************************************************************
      SUBROUTINE REPORT (SHARES_OWNED, SHARES_PROCESSED, FILE_AMT,
     1                   PRICE, FILE_ACCT, NAME_ONFILE,
     2                   FILE_DATE, FILE_TRANS)
      IMPLICIT   NONE
      REAL       SHARES_OWNED, SHARES_PROCESSED, FILE_AMT, PRICE
      INTEGER*4  FILE_ACCT
      CHARACTER  NAME_ONFILE*25, FILE_DATE*8, FILE_TRANS*3, BLANKS*5
      PARAMETER  (BLANKS='     ')

      WRITE (6,26)
   26 FORMAT ('1')
      PRINT *, 'CANINE FUND - STATUS REPORT'
      PRINT *, BLANKS
      PRINT *, 'ACCT NO.: ',FILE_ACCT,'NAME: ',NAME_ONFILE,
     1         'FILE_DATE: ',DATE
      PRINT *, BLANKS
      PRINT *, BLANKS
      PRINT *, BLANKS
      PRINT *, 'TRANS','AMOUNT','PRICE','NO. SHARES','TOTAL OWNED'
      PRINT *, BLANKS
      PRINT *, FILE_TRANS, FILE_AMT, PRICE,
     1         SHARES_PROCESSED, SHARES_OWNED

      RETURN
      END
```

*(Continued)*

**FIGURE 13.8 (f)** REPORT Subroutine for Example 13.2.

5. REWIND:—repositioning of a file to its first record.
6. ENDFILE:—the writing of a special record that concludes a file.
7. OPEN:—the process of connecting a file to a unit.
8. CLOSE:—the process of disconnecting a file from a unit.
9. INQUIRE:—activation of a series of tests that report the status of a file or unit.

These statements may be used with a wide variety of specifiers that enhance their flexibility and the information to be obtained from them. A summary is given in Table 13.2.

```
C*************************************************************
C                          WRAPUP                             *
C*************************************************************
C  THIS SUBROUTINE PRINTS ALL THE SUMMARY FIGURES AND CLOSES*
C  THE FILES. NOTE THAT THE ASSIGNMENT OF 999999 TO THE AC- *
C  COUNT NO. IN THE LAST OUTPUT RECORD IS NOT NECESSARY; IT *
C  IS ALREADY SET TO THAT VALUE WHEN WRAPUP IS CALLED.      *
C*************************************************************
      SUBROUTINE WRAPUP (TOTAL_PURCHASED, TOTAL_WITHDRAWN, TOTAL_SHARES,
     1                   SHARES_PURCHASED, SHARES_WITHDRAWN,
     2                   PRICE, SUBSCRIBERS, NUM_PURCHASES,
     3                   NUM_WITHDRAWLS, TODAY)
      IMPLICIT           NONE
      REAL               TOTAL_PURCHASED, TOTAL_WITHDRAWN, TOTAL_SHARES,
     1                   SHARES_PURCHASED, SHARES_WITHDRAWN, PRICE, FILE_AMT
      INTEGER*2          SUBSCRIBERS, NUM_PURCHASES, NUM_WITHDRAWLS
      INTEGER*4          FILE_ACCT
      CHARACTER          TODAY*8, SPACES*6, FILE_DATE*8, FILE_NAME*6
      PARAMETER          (SPACES = '      ')

      WRITE (6,36)
   36 FORMAT ('1')
      PRINT *, 'CANINE FUND - SUMMARY REPORT FOR ',TODAY
      PRINT *, SPACES
      PRINT *, SPACES
      PRINT *, 'TODAY''S PRICE: ',PRICE,' DOLLARS PER SHARE'
      PRINT *, SPACES
      PRINT *, 'PURCHASES:'
      PRINT *, 'NO. OF SUBSCRIBERS MAKING PURCHASES: ',NUM_PURCHASES
      PRINT *, 'NO. OF SHARES PURCHASED: ',SHARES_PURCHASED
      PRINT *, 'TOTAL AMOUNT OF PURCHASES: ',TOTAL_PURCHASED
      PRINT *, SPACES
      PRINT *, 'WITHDRAWALS:'
      PRINT *, 'NO. OF SUBSCRIBERS MAKING WITHDRAWALS: ',NUM_WITHDRAWLS
      PRINT *, 'NO. OF SHARES WITHDRAWN: ',SHARES_WITHDRAWN
      PRINT *, 'TOTAL AMOUNT WITHDRAWN: ',TOTAL_WITHDRAWN
      PRINT *, SPACES
      PRINT *, 'NO. OF SUBSCRIBERS: ',SUBSCRIBERS
      PRINT *, 'TOTAL NO. OF SHARES CURRENTLY OWNED:',TOTAL_SHARES
      FILE_DATE = TODAY
      FILE_ACCT = 9999
      FILE_AMT = 0.0
      FILE_NAME = SPACES
      WRITE (2,FMT=*) FILE_ACCT, NAME_ONFILE, FILE_DATE,
     1                FILE_TRANS, FILE_AMT, SHARES_OWNED
      ENDFILE 2
      CLOSE (1)
      CLOSE (2)

      RETURN
      END
```

**FIGURE 13.8 (g)** WRAPUP Subroutine for Example 13.2.

**Table 13.1** A Summary of FORTRAN 77 File Characteristics

| | | unformatted | formatted | |
|---|---|---|---|---|
| | | | list-directed | edit-directed |
| external | sequential | ✓ | ✓ | ✓ |
| | direct | ✓ | ✓ | ✓ |
| internal | sequential | × | ✓ | ✓ |
| | direct | × | × | × |

**Table 13.2** Specifiers for Input/Output Operations

| Specifier | Reported As (Specified As) | INPUT/OUTPUT OPERATION | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | READ | WRITE | PRINT | BACKSPACE | REWIND | ENDFILE | OPEN | CLOSE | INQUIRE |
| ACCESS= | character | | | | | | | ✓ | | ✓ |
| BLANK= | character | | | | | | | ✓ | | ✓ |
| DIRECT= | character | | | | | | | | | ✓ |
| END= | integer | ✓ | | | | | | | | |
| ERR= | integer | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EXIST= | logical | | | | | | | | | ✓ |
| FILE= | character | | | | | | | ✓ | | ✓(1) |
| FMT= | integer or * | ✓ | ✓ | ✓ | | | | | | |
| FORM= | character | | | | | | | ✓ | | ✓ |
| FORMATTED= | character | | | | | | | | | ✓ |
| IOSTAT= | integer | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| NAME= | character | | | | | | | | | ✓ |
| NAMED= | logical | | | | | | | | | ✓ |
| NEXTREC= | integer | | | | | | | | | ✓ |
| NUMBER= | integer | | | | | | | | | ✓ |
| OPENED= | logical | | | | | | | | | ✓ |
| REC= | integer | ✓ | ✓ | ✓ | | | | | | |
| RECL= | integer | | | | | | | ✓ | | ✓ |
| SEQUENTIAL= | character | | | | | | | | | ✓ |
| STATUS= | character | | | | | | | ✓ | ✓ | |
| UNFORMATTED= | character | | | | | | | | | ✓ |
| UNIT= | integer | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓(2) |

(1) UNIT specifier cannot be wed when FILE= is used
(2) FILE specifier cannot be used when UNIT= is used

## PROBLEMS

1. Write a brief definition for each of the following terms:

   (a) File
   (b) Record
   (c) Input File
   (d) Output File
   (e) Sequential File
   (f) Direct File
   (g) Unit
   (h) Formatted Data
   (i) Unformatted Data
   (j) List-Directed Data

(k) Edit-Directed Data
(l) System Input Unit
(m) System Output Unit
(n) ENDFILE
(o) External File
(p) Internal File
(q) Preconnected File
(r) Updating

2. Given below is a list in which each file description consists of a combination of characteristics. Some are legal, others are not. Indicate which is which, and explain why the illegal ones are illegal:

(a) Input, Sequential
(b) Direct, Sequential
(c) Output, Formatted
(d) Output, Formatted, Sequential
(e) Input, External
(f) Input, External, direct
(g) Input, Internal, direct
(h) Output, Internal
(i) Output, Internal, Unformatted
(j) Input, Internal, Sequential
(k) Input, Internal, Sequential, Formatted

3. Consider the following program:

```
      PROGRAM          C13P2
      IMPLICIT         NONE
      REAL             V1, V2, TOTLV, PR
      INTEGER*2        NUM, ALLNUM, NDX, UNITS
      CHARACTER*10     NAME
      TOTLV = 0.0
      ALLNUM = 0
      READ (1,*) NUM

      DO NDX=1, NUM
    6   READ (1,*) NAME, V1, V2, UNITS
        PR = SQRT(V1+V2)
        ALLNUM = ALLNUM+UNITS
        TOTLV = TOTLV+PR
        PRINT *, NAME, V1, V2, UNITS, PR
      END DO

      PRINT *, '          '
      PRINT *, NUM
      PRINT *, TOTLV, ALLNUM
      STOP
      END
```

(a) How many files are involved in this program?
(b) How many records does this program read?
(c) How many records does this program print?
(d) We would like this program to process the following input:

```
  NUM:          1
   V1:     -236.4
   V2:      108.86
UNITS:         61
 NAME:    FEFFEL
```

These values were handed to the Chief's brother-in-law after he took the Complete One-Day Computer Workshop, and he was asked to keypunch the values so that they could be processed by the two READ statements in the program. an hour later, he came back with the following two records:

```
     1
61FEFFEL-236.4108.86
```

Explain why this will not work.

(e) Using the values from part (d), prepare the input properly.
(f) Modify Statement 6 so that the program will transfer to Statement 99 if it runs out of data before it is supposed to.
(g) Modify the program so that it will transfer to Statement 88 if something should go wrong when it is trying to read a set of input values for NAME, V1, V2, and UNITS.

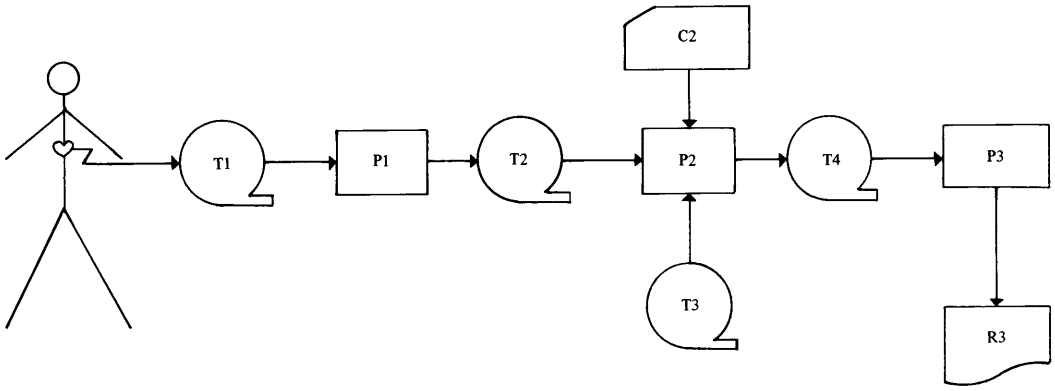



**FIGURE 13.9** Overall Information Flow in Eldorado Heartbeat Process.

4. The El Dorado Medical Center uses a computer to analyze patients' heartbeats. Figure 13.9 shows the general process: a patient is hooked up to a device that records the heartbeats for a specific length of time. The record of those heartbeats (T1) is processed by a program (P1) that "cleans" the data. That is, it reduces or removes disturbances caused by electrical noise, and it adjusts the readings to take care of other known difficulties caused by the behavior of the equipment and/or patient. The result is another record (T2). This record is processed by another program (P2) that also reads a record (C2) with the patient's i.d. and other information. Data for two consecutive heartbeats are extracted from T2, combined with the input record's information, and added to the Center's collection of heartbeat data. This is done by reading the current collection without the new patient (T3) and producing a new version (T4) to which the new patient has been added. At some particular time, all the patients on T4 are processed by another program (P3) that analyzes the heartbeats and produces a report (R3) containing the results for each patient and a summary for the entire collection.

   (a) How many files are involved in the overall process?
   (b) Indicate which files are input files and which files are output files.
   (c) For each file listed in (a), indicate which files are likely to be formatted and which ones are likely to be unformatted. Give the reasons for your choices.
   (d) El Dorado wants a little more information from this system. (Refer to Figure 13.9.) Specifically, P1 is to be expanded so it produces a written report (R1) summarizing what it did. P2 is to be expanded in the same way, producing report R2 in addition to its other duties. Modify Figure 13.9 to show these additional features.
   (e) How many files are added to the system by the changes in (d)? For each of these, indicate whether the file is likely to be formatted or unformatted and explain why.

5. Universal Schmichik uses its computer to process orders for its products. Each mail, phone, or over-the-counter order is prepared on a record. Once a day the accumulated records (for that day and designated C1) are processed by a program (P1). For each record, P1 finds that customer's account on the disk (D1) containing the customer data, reads the information, adds the new order to it, and writes the new version of the account back on the disk, replacing the earlier version. As part of the processing, P1 also finds the inventory data (on the inventory disk D2) for each type of schmichik ordered by that customer. The inventory is reduced by the amount of the order, and the revised information is written onto the disk, replacing the earlier version. (For simplicity, we assume that Universal never runs out of any kind of schmichik.) When an order's processing is complete, P1 prints a copy of the order (R1). When all the orders have been processed, P1 prints a report (R2) summarizing the day's activity, and a report (R3) summarizing the status of the inventory.

   (a) Draw a diagram (such as the one in Figure 13.9) showing these events.
   (b) How many files are involved in the activities described above?

(c) For each file identified in (b), indicate whether it is an input file, an output file, or both, whether it is likely to be sequential or direct, and whether it is likely to be formatted or unformatted. Give reasons for your choices.

6. Describe briefly the difference between:

   (a) Creating a file and opening a file.
   (b) Closing a file and deleting a file.
   (c) Backspacing and rewinding a file.
   (d) File reading and file inquiry.
   (e) Writing and printing a file.

7. Prepare a FORTRAN 77 statement for each of the following activities:

   (a) Read the four items R, W, K, and TR in that order, from the standard system input unit using list directed format.
   (b) Read the two items ST and BR (in that order) from unit 3 using list-directed format. If there is no input, transfer to statement 299.
   (c) Read the items, K, WL, and RT (in that order) from unit 5 using list-directed format. If there is no input, transfer to statement 499. If there is something wrong with the input, transfer to statement 188.
   (d) Read the items K1, NM4, BBL, and DRT (in that order) from unit 2 using edit-directed format. Information about the format is to be found in statement 21. If there is no input, transfer to statement 99. If there is something wrong with the input, transfer to statement 88.
   (e) Write BK, STR, and TRU (in that order) on the system output unit using edit-directed format, with the format described in statement 32. (Do not describe the actual format.) If something is wrong and the output cannot be produced, transfer to statement 77.
   (f) Write WRUP, SKPL, SURT, ALTN, and CRKS (in that order) on unit 7 as five separate records using list-directed format. If something is wrong and the production of output is impossible, transfer to Statement 177.

8. Something is wrong with each of the following statements. Indicate what it is:

   (a) `READ (1, *) END=199, X, Y`
   (b) `READ (26, FMT, END=199) B1, S, TR`
   (c) `WRITE (UNIT=3, *) T, WR`
   (d) `PRINT (UNIT=5, FMT=*) W1, W2`
   (e) `READ (UNIT=5, FMT=*, END=, ERR=17) A1, XT`
   (f) `WRITE (6, 12, FMT=*, ERR=27) YV, YW`
   (g) `WRITE ( (, 12, END=299, ERR=288) TTW, TH`
   (h) `PRINT 26, FMT=*, ERR=18, GR, BG, EST`
   (i) `READ (END=99, ERR=188) AC, DE, AC`
   (j) `PRINT (UNIT=6, FMT=22, ERR=88) F1, F3`
   (k) `READ (UNIT=1, FMT=*, END=399, ERR=388)`

9. Write a FORTRAN 77 statement to perform each of the following activities:

   (a) Open a file on unit 4.
   (b) Open a file named MSTR on unit 2.
   (c) Open a sequential formatted file on unit 1.
   (d) Open a direct formatted file with record length 100 on unit 8.
   (e) Create a sequential formatted file named TRANS on unit 3 so that the outcome of the process will be reported in a variable named OPNHOW.
   (f) Close the file connected to unit 2.
   (g) Close the file connected to unit 1 and get rid of it.
   (h) Close the file connected to unit 3 and keep it.
   (i) Close the file connected to unit 4, keep the file, and report the outcome of the closing process in a variable named CLSHOW. If something goes wrong, transfer to statement 399.

10. Write the appropriate declarations to go along with each of the following statements:

    (a) `INQUIRE (3, EXIST=IYAM)`
    (b) `INQUIRE (FILE=MYFILE, EXIST=IYAM, NUMBER=WHICH, ACCESS=HOW)`
    (c) `INQUIRE (4, NAME=CALLED, OPENED=OPNST, ACCESS=SEQDIR, FORM=FV)`

(d) `INQUIRE (FILE=MF, EXIST=AM, NUMBER=UN, ACCESS=DS, RECL=NBYT)`
(e) `INQUIRE (2, NAME=CL, ACCESS=SD, NEXTREC=FAR, BLANK=INVIS)`

11. Modify the program in Example 13.1 so that it handles records out of sequence. The output file is unchanged in that its records must be in sequence by ascending (increasing) account number. Those input records found to be out of sequence must be prevented from being added to the output file. (Thus, unlike Example 13.1, this problem no longer guarantees that all the input will be in proper sequence.) The modified program, then, is to look for records out of sequence. When it finds such a record, it is to be written onto a separate output file, connected to unit number 3. (This may be a formatted, list-directed file.)

    After printing the summary information (as in the example program), the revised program is to print the number of records found to be out of sequence (i.e., the number of customer account records placed onto the file connected to unit 3). The last account on unit 3's file is to be followed by a dummy record with account number `999999`.

12. Assume that the new output file produced in Problem 11 (the one connected to unit 3) has been processed by a program that sorted its records and produced a new disk with the records in proper order. Write a program that treats this collection of records as an input file connected to unit 4 and uses the other file produced by Problem 11 (the disk produced by the original example program) as a second input source, connected to unit 2. The program is to *merge* the two files (i.e., prepare a single file) so that the new output file (connected to unit 3) contains all of the records from the two input files in proper sequence. A single dummy record with account number `999999` is to be placed after the last actual account record and before the `ENDFILE` record on the new output file. In addition, the program is to print a summary showing the number of records that came from the newly sorted file, the number of records that came from the original file, and the total number of records in the merged file. (Of course, the dummy records do not count.)

# 14
# *List-Directed Input/Output*

Since we have been using list-directed input/output all along, there is no need to start from the beginning. Instead, this chapter will use what we know already to examine the full range of FORTRAN 77's list-directed features.

## 14.1 PREPARATION OF LIST-DIRECTED DATA

Facilities for handling list-directed data give the programmer an opportunity to specify effective input/output operations without having to pay detailed attention to the exact format. Of course, this does not mean that the data are totally without form. As the previous chapter pointed out, we still are dealing with formatted data; however, the rules are quite relaxed, so that FORTRAN 77's interpretive mechanisms accept a certain amount of variation in the way the data appear.

### 14.1.1 Basic Requirements for List-Directed Input

To make the list-directed mechanism work, the input data values must follow four basic rules:

1. The *number* of data items must be consistent with the number indicated by the input list. For example, if the READ statement says there should be six actual data values. We shall see shortly that items may be omitted from the data (even though their names appear in the list), but we have to show that they are absent.

2. The *type* of each data value must be consistent with the type associated with the corresponding name in the input list. Real values must appear with a decimal point, either in conventional or floating-point form. Character values must appear with the apostrophes around them. Referring to the previous READ statement, if NAME were declared as CHARACTER, its value would have to be enclosed in apostrophes even if it contained nothing but numerical characters.

3. The *order* in which the data values are listed must agree with that in which their names appear in the input list. Using our example again, if 18.27 is the value we intend to store in WDTH, then it must appear as the fourth value in the data list.

4. List-direct input values must be physically separated from each other using one of two ways that FORTRAN 77 recognizes:

   (1) Data items may be separated by one or more blanks

   (2) Data items may be separated by commas. Thus, any of the input lists shown below

```
30662  'TWOMBLEY, M. H. '  39.16  18.27  106.4  317.44
30662,'TWOMBLEY, M. H. ',39.16,18.27,106.4,317.44
30662,  'TWOMBLEY, M. H. ',  39.16,  18.27  ,  106.4,317.44
```

will produce the same results. Note that the blanks appearing together with the commas have no effect.

### 14.1.1 List-Directed Data and Records

When preparing list-directed input, remember that we are dealing with *records*. Each READ statement brings in a new record, and when the input file is connected to a terminal, each record corresponds to a line. Consequently, the programmer must be careful about the relationship between the contents of a line and the length of the input list for a READ statement. We can summarize the various situations as follows:

1. When the number of items in an input list matches the number of data values on the line, the READ statement brings in all the items, and the program is ready to read from the next line. This is illustrated in Figure 14.1. The six input values are read by a single statement and entered on a single line.

2. When the input list asks for more values than there are on a record, the program will read the next record, etc., if it needs to, in order to try to meet the requirements of the input list. This is illustrated in Figure 14.2. In this situation the READ statement still requests six items, but the first record has only three values. Accordingly, that first record is read, taking care of the list's first three items. Then, the next record is read, providing values for the fourth and fifth items. Since there are no more values on that (second) record and the operation still has one more item to go, a third record is read and the first value on that record (in this case, the only one) is read and stored in the last variable, WT.

3. When there are more values on a record than an input list requests, the program uses the appropriate number of values, starting with the first one. Note, however, that *the remaining values are not "saved" for the next* READ statement. Since each READ statement starts with a new record, any values left over on the previous record are lost. This is seen in Figure 14.3. Since the first READ statement wants four items, it will bring in and use all of the first record and the 18.27 from the second record. The second READ statement starts with a new record (the third one), so that HT will receive the value 317.44. Gone and lost (alas) is the 106.4.

```
30662    'TWOMBLEY, M. H. '   39.16   18.27   106.4   317.44
READ *,  IDNO, NAME, LTH, WDTH, HT, WT
```

**FIGURE 14.1** Matching READ Statement and List-directed Input Card.

```
          30662    'TWOMBLY, M. H. '   39.16
             18.27    106.4
               317.44
READ *,  IDNO, NAME, LTH, WDTH, HT, WT
```

**FIGURE 14.2** READ Statement/Data Format Requiring Multiple Records.

```
  30662  'TWOMBLY, M. H. '    39.16
      18.27    106.4
        317.44
READ *,  IDNO, NAME, LTH, WDTH
READ * HT
```

**FIGURE 14.3** Mismatch Between READ Statement Lists and Input Data Format.

```
(a) 361, 'GROOBNIK, Y. A. ', 39. 16, 18. 27, 3. 04, 118. 50
(b) 361, 'GROOBNIK, Y. A. ', 39. 16, 3. 04, 118. 50
(c) READ (*, *) IDNO, NAME, LTH, WDTH, HT, WT
```

**FIGURE 14.4** Effects of Blanks in List-Directed Input Data.

### 14.1.3 Treatment of Blanks

When blanks appear in place of a data value, FORTRAN treats them as part of the separators between input values. For instance, consider the six values in Figure 14.4(a), along with the accompanying READ statement in Figure 14.4(c).

Now, suppose we rewrote the line so that it appeared as in Figure 14.4(b). The 18. 27 in the original line is omitted and there are blanks where it used to be. As a result, everything between the 39. 16 and the 3. 04 is treated as a separator. That means that IDNO, NAME, and LTH will receive the first three values, as they did before; WDTH will receive the next value, which now is 3. 04; 118. 50 is read in HT; and that finishes the record. Since there still is another value requested by the READ statement, the program will attempt to read another record and use its first value for WT. If it can, it will; if not, an error message will be issued.

As indicated earlier, a blank in a character string is a blank, as long as it is inside a set of apostrophes. However, when FORTRAN expects a character string and it finds blanks there, without the enclosing apostrophes, it treats that as a separator and attempts to read the next available input value into the character variable. If this is a number, there is an error and the input is unsuccessful. For example, using the same READ statement as in Figure 14.4, the input line shown in Figure 14.5(a) is acceptable, while the one in Figure 14.5(b) is not.

### 14.1.4 Missing Data

For list-directed data, a missing value can be expressed by placing two separators next to each other in the position where the value would have appeared. Thus, if we were to use the READ statement and data as shown in Figure 14.6, the value in the variable NAME would be unchanged from whatever it was before the READ statement. Compare this with the data in Figure 14.5(a). In that case, the value stored in NAME would be a string consisting entirely of blanks.

### 14.1.5 Additional Possibilities

FORTRAN 77 provides two additional features that may be used for list-directed input data preparation. Since these features have rather special uses, they are not emphasized here; they are mentioned primarily for completeness.

```
(a) 361, '', 39. 16, 18. 27, 3. 04, 118. 50
(b) 361, 39. 16, , 18. 27, 3. 04, 118. 50
```

**FIGURE 14.5** Effect of Blanks with List-Directed Character Input.

```
361, , 39. 16, 18. 27, 3. 04, 118. 50

READ *, IDNO, NAME, LTH, WDTH, HT, WT
```

**FIGURE 14.6** Missing Data in List-Directed Input.

**14.1.5.1 Repeated Input Values** If the same input value is to be repeated several times, it can be represented in compressed form by showing it once, preceded by a repetition factor. The general form is

*rf***value*

where *rf* is an unsigned positive integer. Thus, for example, the sequence of list-directed input values shown below

```
32.6,32.6,32.6,3,0,0,0,0,-12.2,0.0,'HEH','HEH'
```

also can be written as

```
3*32.6,3,4*0,2*-12.2,2*0,2*'HEH'
```

**14.1.5.2 The Slash in List-Directed Input** When a slash (/) appears in list-directed input, FORTRAN 77 interprets this as an end-of-input signal, even though it may not correspond to the physical end of the record. For instance, if there are five input values on a line, and a READ statement designed to bring in the five values runs across a slash after the third value, it will use only the three values that it read so far. The remaining two values are treated as if they were missing. Thus, the fourth and fifth variables in the statement's input list will remain unchanged and the READ statement will be considered as being concluded.

### 14.1.6 List-Directed Output Data

The programmer has only limited control over the formatting of list-directed output. He or she can specify the order in which the individual values are placed in the output list. In addition, the programmer can force the beginning of a new line, because each PRINT *, statement automatically starts a new line. Beyond that, however, control rests with FORTRAN. The maximum number of items on a line, their physical appearance, and the type of separation between them are built into the system, and they may vary. Those listed below are typical of HP systems, but they are not necessarily universal.

1. Integer values are printed with a preceding blank. High order zeros are suppressed. If an integer value is negative, its sign appears just prior to the leftmost nonzero digit. INTEGER*4 values are followed by a T. Thus, value of −147 appears as

```
b-147b
```

and 43674 appears as

```
b43674bTb
```

2. A single precision value will appear either in conventional form or in floating-point form, depending on the magnitude of the number. A value below 0.00001 (i.e., 10**-5) is considered too small to show in conventional form. Thus, a value of 0.000008768 would be printed as

```
b8.678E-06b
```

while a value of −0.05437 would be printed as

```
b-0.05437b
```

Similarly, an absolute value above 100000 (i.e., 10*5) would be considered too large for conventional notation. Thus, a value of 6,243,520 would be printed as

```
b6.24352E+06b
```

while a value of 721.446 would appear as

```
b721.446b
```

3. Character strings are printed exactly as they are (no enclosing apostrophes or extra blanks).

There is no fixed number of items per line. Instead, FORTRAN 77, using its predefined list-directed formatting rules, will keep printing until it runs out of room, at which point it will start a new line. Of course, if it turns out that a particular PRINT *, statement requires such a new line at an awkward point, the programmer can break the statement into several individual ones, each with a smaller output list.

## 14.2 LIST-DIRECTED INPUT/OUTPUT STATEMENTS

This section will discuss some additional capabilities for list-directed data transmission.

### 14.2.1 The List-Directed READ Statement

The list-directed READ statement is no stranger to us. Consequently, all that needs to be done here is to summarize the statement's construction and features.

**14.2.1.1 Basic Forms** The list-directed READ statement can be written as

READ *, *inputlist*
or READ (*unitnumber*,*) *inputlist*
or READ (*unitnumber*,FMT=*) *inputlist*

When the input device is the system's input unit, the FORTRAN compiler accepts another list-directed statement form:

READ (*, *) *inputlist*

The first asterisk indicates the system's input unit and the second asterisk (as always) indicates list-directed format.

When one of the parenthesized forms is used, we can include any or all of the specifiers available for the READ statement (i.e., END, ERROR, and IOSTAT). Thus, we can test list-directed input for end-of-file, error, and so on. (Refer to Table 13.2.)

**14.2.1.2 Input with Implied Loops** As part of the discussion about arrays (Section 7.3.2.2) it was noted that the READ statement could be used to bring in a succession of array elements by specifying an implied loop. This is done by enclosing the loop in parentheses and including it as an "item" in the input list. Suppose, for example, that TBL is declared as a 15-element array, I and NUM are declared as single-valued integer variables, and A, LBL, and VST all are declared as single-valued variables of some type. Then, the statement

```
READ *, A, (TBL(I), I=1,5), LBL, VST
```

will read the next eight values from the system's input unit. The first of these will be stored in A, the next five in elements TBL(1) through TBL(5), the seventh value in LBL, and the eighth one in VST. The other ten elements of TBL will remain unchanged. If we were to change our statement so that it said

```
READ *, A, (TBL(I), I=5,9), LBL, VST
```

there would be no change in the length of the input list; the statement still would read the same number of values. However, this time the second, third, fourth, fifth and sixth values would go into elements TBL(5) through TBL(9). As before, TBL's other elements would remain unchanged.

Even though the FORTRAN word DO does not appear in the READ statement, the action of the implied loop is exactly the same as it is in an ordinary DO loop. Accordingly, if we wrote

```
READ *, A, (TBL(I), I=5,13,2), LBL, VST
```

the next eight input values still would be read, but this time it is the elements TBL(5), TBL(7), TBL(9), TBL911), and TBL(13) that would receive new values.

The flexibility of the implied loop is increased further by the fact that the number of array elements to be transmitted does not have to remain constant for every use of the statement. For example, we can specify something like the following:

```
READ *, A, NUM, (TBL(I), I=7,NUM), LBL, VST
```

As a result, the number of items transmitted by this statement will depend on the value read into NUM. For instance, if NUM is 7, only a single element will be read into TBL, specifically TBL(7), and that will be the third of five items read by that statement. On the other hand, if NUM is 15, the same statement then would read 13 items, with nine of them (i.e., the third through eleventh values) going into elements TBL(7) through TBL(15). As is true with many other things in life, this flexibility has a price attached to it. In order to take advantage of it, the programmer must make sure that the variable limiting value "makes sense." For example, based on the way this READ statement is constructed, a NUM value below 7 means that there will be no cycles through the built-in DO loop. When NUM is below 7, the next input value after NUM will be placed in LBL. This is fine as long as the programmer knows it will happen and intends for it to happen that way.

An implied loop can be set up to keep track of more than one sequence of array elements. For example, suppose we declared another array BRK, this one having 24 elements. Now, if we write the statement

```
READ *, NUM, (BRK(I), TBL(I), I=6,NUM)
```

and NUM is read in as 11, the statement will read a total of 13 values: NUM itself, BRK(6), TBL(6), BRK(7), TBL(7), and so on through BRK(11), TBL(11).

Implied loops may be nested basically in the same way as regular DO loops are. Thus, if we declared TWODEE as a $4 \times 6$ array, R and C as integer variables, and we specified the statement

```
READ *, ((TWODEE(R,C), C=2,4), R=2,3)
```

the result will be that the next six input values will be read and stored, respectively, in TWODEE(2,2), TWODEE(2,3), TWODEE(2,4), TWODEE(3,2), TWODEE(3,3), and TWODEE(3,4). Had we reversed the indexes, so that the statement said

```
READ *, ((TWODEE(R,C), R=2,3), C=2,4)
```

it still would read the same six values. However, their respective destinations now would be TWODEE(2,2), TWODEE(3,2), TWODEE(2,3), TWODEE(3,3), TWODEE(2,4), and TWODEE(3,4).

HP FORTRAN 77 allows an additional use of the implied DO loop as a convenient way for expressing repeated input or output. For example, the statement

```
PRINT *, (BFAC, EST, TRAIL, I=1,2)
```

produces the values of the three variables shown twice:

```
BFAC   EST   TRAIL   BFAC   EST   TRAIL
```

### 14.2.2 The List-Directed WRITE Statement

List-directed output is handled by the WRITE statement, whose similarity to the READ statement already is clear. In addition, FORTRAN provides the PRINT statement as a special case of WRITE. This section examines the range of available features for list-directed use.

**14.2.2.1 Construction of the List-Directed WRITE** Acceptable forms for the list-directed WRITE statement can be summarized as follows:

```
WRITE (unitnumber,*) outputlist
```

or

```
WRITE (unitnumber,FMT=*) outputlist
```

Note that, unlike the list-directed READ statement, the WRITE statement must have parentheses, and it always must specify a unit. If the unit is the system's input unit, it can be indicated by an asterisk, as in the form

```
WRITE (*,*) outputlist
```

The form without parentheses is reserved for a separate statement, PRINT.

Output lists follow the same rules of construction that input lists do. In addition, there are extensions which would not make sense in an input list but provide powerful conveniences for preparing and displaying results of computations.

**14.2.2.2 Literals in an Output List** A literal, i.e., a character string constant, can be specified as just another item in an output list. To review, suppose we have a real variable TENSTR whose current value is 12577.83. If we said

```
WRITE (*,*) 'TENSILE STRESS = ',TENSTR,'LBS/SQ. IN. '
```

the program would print the literals exactly as they appear (without the apostrophes), and it would use its default formatting to show the numerical value. Typically, the resulting line would look like this:

```
TENSILE STRESS =  0.1257783E+05   LBS/SQ. IN.
```

There is no particular limit on the length of a literal in list-directed output; FORTRAN simply will use as many output lines as it needs to fulfill the requirements of the entire output list, regardless of its composition. The form

```
WRITE (1,*) literal
```

or

```
WRITE (6,*) literal
```

(depending on the particular HP system) is particularly handy for displaying messages and prompts on the user's terminal.

**14.2.2.3 Expressions in Output Lists** This powerful feature allows the output list to include a variety of forms that go well beyond the simple literal, single-valued variable, array element, or array name. The following also are acceptable:

1. A substring of a character variable.
2. Any valid arithmetic expression. All of FORTRAN's arithmetic facilities are available, including parentheses and function references. However, the functions thus invoked must not contain any READ or WRITE statements.

3. Any valid character string expression as long as the length of each participating character string is known.

Assuming the appropriate context, this means that we could specify something like

```
      WRITE (*,*) IDNO, 'SUBSTANCE '//NAME//''''S VALUE IS ',
     1                                3.27*(VISC + SQRT(DENS/DIFFUS))
```

This feature will not receive further emphasis in this book. While it can be a great convenience, the occasions for its use must be considered carefully. This feature makes it tempting to squeeze down the size of a program by combining computations and output into a small number of statements, or even into a single statement. However, that is not nearly as important an objective as the production of reliable programs whose logic is easy to follow.

### 14.2.3 The PRINT Statement

The basic form for the list-directed PRINT statement is

PRINT *, *outputlist*

Structural possibilities for the output list are exactly the same as those available for the WRITE statement.

## 14.3 SUMMARY

List-directed input/output provides a way of transmitting formatted data in which the programmer supplies a minimum of information about the detailed appearance of the records or their individual data items. Instead, FORTRAN "looks for" blanks or commas between adjacent data values, using these as signals for determining when one value ends and the next one begins. Thus, the following string of characters

```
435.6   'KINCAID, D.H.'   42   -6088.05
```

is recognized as a sequence of four list-directed data values.

On input, these values will be related to the items specified in an input list, and FORTRAN will convert the individual character strings in accordance with the types of the corresponding variables. Using the example list given above, and assuming appropriate declarations, the first string of characters (the 435.6) will be converted to a real number, the second string (already in the appropriate form) will be stored as is (without the apostrophes), the third value (42) will be converted to integer, and the fourth one to a real number.

When list-directed output is called for, FORTRAN takes the values from storage in accordance with the specifications in the output list, converts them to character form, and inserts blank separators between the converted values. The number of blanks used for separation of output values depends on the individual compiler, but it is not controlled by the programmer.

List-directed transmission is denoted by using an asterisk as a format specifier. This form is used most typically with the system's standard input and output units, in which case the basic statements are as follows:

READ *, *inputlist*

and

PRINT *, *outputlist*

**PROBLEMS**

1. Consider the following sequence of statements:

```
REAL              B, A(3)
INTEGER*2         COUNT
CHARACTER         WD*8

B = 17.4
A(1) = -0.8
A(2) = 0.0
A(3) = 231.6
COUNT = 1
WD = 'RELEVANT'

READ *, A, B, WD, COUNT
```

Given below are several sets of input data. Some consist of a single line while others have more than one line. Apply the READ statement given above to each set of input. (Treat each set independently.) In some cases, the input will be successful, while in others it will not work. Show the resulting values for B, A, COUNT, and WORD in each case. When the particular combination of data is illegal for the READ statement, indicate what the problem is:

(a) `43.4  -81.26  313.07  -5.4  'BROADEST'  24`

(b) `207.7  4.0  61.07` (first line)
`87.6  'THUNDERS'  18`

(c) `-60  -22.71  -808.7  14.4  '3247.068'  -16`

(d) `19.1  207.08  -412.63  8.8  '  ',`

(e) `227.83,  ,17.4,,,474`

(f) `61.72,808.0,146.7,'CHARACTERISTIC',26` (first line)
`63,-87,'DEVELOPS',-8`

(g) `208.26,,,,-3,-4,87/`

(h) `64.2,,  ,' ',4,18`

(i) `17.2,4*-14.7,'SA'.617`

(j) `50.8,2*0.0/'BALANCED',-6` (first line)
`24,43/`

(k) `/3*26.2,'  ',74/`

(l) `17.43,-81.42,-7.7,28.3` (first line)
`'ENGRAVINGS'/`
`24`

(m) `32.75,,,,,'  TRUE  ',-8`

(n) 61.81,,,,,49,722

2. Consider the following sequence of statements:

```
REAL              PR(6), MDL(3,2), A
INTEGER*2         TTL(2,3), I, J
A = 1.0
DO I=1,6
  PR(I) = A+1
  DO J=1,3
    IF (I .LT. 4) THEN
      MDL(J,1) = PR(I)**J
      TTL(1,J) = MDL(J,1)**(J-1)
    ELSE
      MDL(J,2) + PR(I-1)**(J-1)
      TTL(2,J) = MDL(J,2)
    END IF
  END DO
END DO
```

The following data are submitted as input:

```
4.0,2.7,7*3.0,2*5.0,2*2.2,      (first line)
  6,-1.0,2*5,8,71,-12,
    36,-57
```

Apply each of the following statements to these input lines and indicate the resulting values in the variables PR, MDL, A, and TTL. In those cases where a sequence of statements is given, assume that nothing happens between each READ statement and the next one in that sequence. In those cases where the input will not work, indicate the reason(s) for the problem.

(a) READ *, PR, A, MDL, TTL
(b) READ (*, *) PR, MDL, TTL
(c) READ (*, *) PR(1), PR(2), A, MDL, PR, TTL
(d) READ *, MDL, PR, A, TTL, TTL(1, 2)
(e) READ (*, *) MDL, PR(6), A, (PR(I), I=1.5), TTL
(f) READ *, MDL, PR(1), A, (PR(J), J=2, 6), (TTL(2, I), I=1, 3)
(g) READ *, A, PR(I), I=1, 3, MDL, PR(J), J=4, 6, TTL
(h) READ (*, *) ((MDL(I, J), I=1, 3), J=1, 1), A, PR
    READ (*, *) TTL
(i) READ (*, *) (PR(I), I=4, 6), MDL, A, (PR(I), I=1, 3), TTL(1, 1)
    READ (*, *) (TTL(J, I), J=1, 3), I=2, 2)
    READ (*, *) TTL(2, 3)
(j) READ (*, FMT=*) A
    READ *, TTL
    READ *, TTL

3. Here is part of a program:

```
PROGRAM        C14P3
IMPLICIT       NONE
REAL           A(4), B(2, 4), VERT
INTEGER*2      WR(6), N, R, C
CHARACTER      OBJ*4, STR*8
PARAMETER      (BLANKS='        ')

DO 8 N=1.6
  WR(N) = 6+I
  IF (N.GT. 2) THEN
    A(N-2) = WR(N-2) + MOD(N, 6)
    B(1, N-2) = A(N-2) + WR(N-2)
  ELSE
    B(2, N) = WR(N) * MOD(2*N, 6)
    B(2, N+2) = 2*WR(N) - MOD(N, 5)
  END IF
8 CONTINUE
  VERT = MAX(A(1), A(4), B(1, 4), B(2, 4))
  OBJ = 'BARK'
  STR = OBJ(1:3)//'E'//OBJ(1:2)//'C'//OBJ(4:)
```

Assuming nothing else happens to any of the variables, show the output produced by each of the following statements or sequences of statements. Assume that unit 6 is the standard system output unit and that the system will print according to the rules given in Section 14.1.6. Note that some of these statements are illegal and will not produce output. Indicate which ones they are and why they will not work:

(a) PRINT *, A
(b) PRINT (6) VERT, OBJ
(c) WRITE (6, *) WR
(d) PRINT *, VERT
    PRINT *, BLANKS
    PRINT *, (B(1, C), C=1, 4), STR
    PRINT *, (B(2, C), C=1, 4), OBJ
(e) WRITE (*, *) OBJ//' '//STR
    PRINT *, BLANKS
    WRITE (6, FMT=*) VERT, (WR(N), N=1, 2)
    PRINT *, (WR(N), N=3, 6)

```
(f) PRINT *, (A(N), N=6, 2)
    PRINT *, VERT, OBJ, STR
(g) PRINT *, (A(N), B((N-1)/4+1, N), N=1, 3, 2)
    PRINT *, (A(N), B((N-1)/4+1, N), N=2, 4, 2)
(h) PRINT *, (A(N), B((N-1)/4+1, N), N=1, 4)
(i)   WRITE(*, 8)
    8 FORMAT('1')
      WRITE (6*) OBJ, VERT, STR(4:7), VERT
      WRITE (6, *) 3*VERT, VERT-1
      WRITE (6, *) (WR(R), R=5, 2, -1)
(j) WRITE (*, *) VERT//STR
    PRINT *, (A(R), C=1, 4)
(k) PRINT *, 'A = ', A
    PRINT *, BLANKS
    PRINT *, 'OBJ IS NOT ', STR
    PRINT *, BLANKS
    PRINT *, 'B = ', B
```

4. Redeye Airlines is among the country's smaller airlines. Perhaps you may not have heard of it, but it is there, offering needed service among only four cities: Aching Kidney, Bilgewater, Craggy Mound, and Double Scoop. (For convenience, we shall refer to them as A, B, C, and D, respectively.) While Redeye does not have as many airplanes as some of the other airlines, it tries to keep up by having as many fares as the Big Ones. Thus, the cost to fly between two given cities depends on the ticket class. There are four classes:

   (1) First Class (F): This really is super deluxe. A licensed government official certifies that the plastic drinking cups collected from Class F customers actually are thrown away after each flight.
   (2) Coach Class (Y): The service in this class is standard. On meal flights, the Head Attendant leads the Y class passengers in a lively game of Guess What That Is Under The Viscous Brown Goo.
   (3) Night Coach Class (N): This is the "no frills" plan, available after 10:00 P.M. under certain conditions. Those N class passengers taking their own water may drink it if they also bring their own cups.
   (4) Excursion Class (E): This is the rock bottom economy plan. Passengers traveling under this plan must buy their tickets at least eleven months before the flight and must pay in full at that time. Flights in this class may be cancelled by passengers up to five months before flight time. Flights in this class may be cancelled by Redeye up to eleven seconds before flight time.

   The following table summarizes the rate schedule. Note that the rates given here are for round trips. Redeye books only round trips. Moreover, Redeye books only two-flight round trips. For instance, A to C to A is a round trip; A to B to C back to A is a trip that Redeye will not book: too complicated. Also, the round trip must be in the same class both ways:

| *BETWEEN* | *F* | *Y* | *N* | *E* |
|---|---|---|---|---|
| A and B | $211 | $194 | $187 | $169 |
| A and C | — | $121 | $111 | $ 92 |
| A and D | $248 | $238 | — | $216 |
| B and C | $189 | $180 | $174 | — |
| B and D | — | $ 89 | — | — |
| C and D | $ 96 | $ 91 | $ 87 | $ 77 |

   Wait. There is more:

   (1) Between A and B, Class N is not available on Tuesdays.
   (2) Between A and B, Class E carries a 5% discount if either leg of the flight (A to B or B to A) is taken on a Saturday.
   (3) Between A and C there is no class N on Wednesdays or Sundays.
   (4) Between B and C there is no class N on Saturdays or Sundays.
   (5) Between C and D there is class N only on Mondays.
   (6) Between C and D there is no class E on Monday, Tuesday, Thursday, or Saturday.

   When a Redeye passenger wants to book a trip, he or she gives only the following information: name,

originating city, destination city, flight class, day of week that the passenger is leaving, and day of week that the passenger is returning. Redeye tells the passenger which flights and what time. (Some airline.)

(a) Assuming that the input for each round trip is to be prepared on a single line, design a list-directed format that provides the information described above. The order of the data items is not crucial. However, it is up to you to determine what the data will look like.
(b) Write a program that will read the input you designed for part (a). For each line that it reads, the program is to print a line of output that shows the passenger's name and flight class. That is followed by a second line showing the originating city and day of week. A third line shows the destination city and the day of week for the return flight. The fourth and final line of output shows the price for the round trip. If it turns out that the input violates one or more of the scheduling rules and the round trip cannot be booked, the fourth line is to show an appropriate message instead of the ticket price. Leave a blank line between output sets and use end of file to stop the run.

5. Pressure from its many passengers has persuaded Redeye Airlines to be more flexible. Recall (from Problem 4) that up to now, when a passenger wanted to go, say, from Aching Kidney to Craggy Mound to Bilgewater and back to Aching Kidney, the passenger drove or went by bus. (Lurching Rutways serves all these cities.) But this is changing. Redeye is going to provide this service. To do so, the airline must adjust its rules for computing the rates. Here is what they decided:

(1) Each *trip* will consist of a combination of *legs* and each leg will be considered a one-way trip.
(2) The rate for a one-way trip will be half of the corresponding round trip rate.
(3) The longest (most extensive) trip consists of no more than six legs and the destination of the last leg must be the same as the origin of the first leg.
(4) The rates to be charged for *all* the legs of a trip will be taken from the most economical plan used for a particular leg of that trip. For instance, if a passenger books from A to C in Class F, C to D in Class Y, and D to A in class N, he or she (under this new system) will be charged at the lowest rate (in this case, the N rate) for each of the four legs, even though only one leg originally was booked in class N, and even though one or more of the other legs originally may not have been available in that class. (Excursion class is not included in this revision; passengers book round trip excursions as before.) Isn't that nice? A Redeye passenger, with some planning, could work things out so that he or she flies, say, a four leg flight with three legs being flown in first class luxury while paying night coach rates for the whole thing.

Revise (or rewrite) the program from the previous problem so that it reflects these changes. The input now consists of several lines for each passenger. The first line contains the passenger's name. This is followed by a line for each leg of the trip showing the origin and destination cities (for that leg), the flight class, and the day of the week for that leg. (For ease of processing, you may want to include a special line that separates different passengers' data.) Output for this version shows the passenger's name on a separate line, followed by a line for each leg of the trip showing the origin and destination cities, the class under which the passenger requested that leg, and the class under which he or she is being charged for that leg. A final output line gives the total ticket price. As before, leave a blank line between output sets and use end of file to stop the run.

6. Happy Melvin's Auto Rental offers six kinds of automobiles for rent, each with its own rental plan:

(1) A Phantasm rents for $25 per day plus 20 cents per mile, gas included.
(2) A LeBonza rents for $22 per day plus 23 cents per mile, gas included.
(3) An Albatross rents for $99 per week or any part thereof. There is no mileage charge and the customer pays for the gas. Mileage is guaranteed to be no worse than 22 miles per gallon. (Happy Melvin refunds the difference if the mileage is worse than that.)
(4) An Achilles rents for $26 per day. The first 75 miles are free. After that, it is 12 cents per mile. The customer pays for all the gas, and mileage is guaranteed not to be worse than 20 miles per gallon.
(5) A Classique rents for $176 per week (1/2 week minimum, full week rate for anything over 1/2 week). There is no mileage charge and the customer pays for the gas. Mileage is guaranteed not to be worse than 17 miles per gallon.
(6) A Flamboyo rents for $34 per day. The first 150 miles are free. After that, there is a charge of 17 cents per mile. The customer pays for all the gas and mileage is guaranteed to be no worse than 15 miles per gallon. (You ought to see this Flamboyo; quite a car.)

Happy Melvin would like to offer an interesting supplementary service. He wants a program that will

accept input consisting of a customer's name, the number of miles he/she expects to drive, and the number of days the automobile will be rented. In response, the program is to produce two lines of output: the first is an echo of the input and the second shows the minimum amount that the customer can expect to pay for this usage, along with the type of auto the customer would be getting. Leave a blank line between output sets and use end of file to stop the run. For computational purposes, use a gasoline cost of $1.50 per gallon.

7. Modify the program in Problem 6 so that the gasoline cost (in dollars per gallon) is an input item submitted for each run.

8. Modify the program in Problem 6 or 7 so that the customer does not have to specify the number of days for his/her car rental. If the specification is given, it is used. However, if it is missing, the program is to use a default value of one day.

9. Using the same rental schedule as in Problem 6, modify the program from Problem 6, 7, or 8 so that the output is expanded to show all of the rental possibilities instead of just the least expensive one. Each choice (the cost and the type of auto) is to appear on a separate line, with the least expensive one being printed first, followed by the next one, and so on, with the most expensive choice being printed last.

10. Now, Happy Melvin wants to provide another service. This time, using the same rental schedule as in Problem 6, he wants the customer to be able to submit his/her name, expected number of miles, number of days (optional, as in Problem 8), and a dollar amount. This program is to print an echo of the input on a single line, followed by an additional line for each type of automobile whose rental rate makes it possible for the customer to use the car within the specified dollar amount. The choices are to be arranged so that they are printed in order by increasing cost (least expensive one first). If the dollar amount given in the input is too small to rent *any* of Happy Melvin's wondrous choices, the program is to print a message to that effect. As an added service, the program is to print a third line showing the least amount of money needed to meet the customer's mileage and time requirements. (Of course, this added service applies only to those cases in which the input dollar amount is not enough to rent anything.) You may use either a fixed gasoline cost (as in Problem 6) or design the program to expect a cost figure as an input value (as in Problem 7).

11. Once Happy Melvin gets started, he is difficult to stop. (It is not difficult to see why he is called Happy.) Now he wants a program that accepts input consisting of a customer's name, number of days (with a default of 1 as in Problem 8), and a dollar amount. This time, the program prints an echo of the input, followed by a line of output showing the *maximum* number of miles that those dollars will provide, along with the type of auto in which they would be driven. If the dollar amount is too low to buy anything, the program is to print a message to that effect. As before, use $1.50 per gallon as the gasoline cost for the computations, or include a cost figure as an input item.

# 15
# *Edit-Directed Input/Output*

This chapter discusses the extensive features that enable the FORTRAN programmer to control every detail of input and output data formats. We shall examine the basic structures around which these features are organized and acquaint ourselves with their use. Once the skills have been developed and we can design just about any format we want to design, the next chapter will discuss some advanced formatting features that make these designs even more versatile.

## 15.1 PROPERTIES OF EDIT-DIRECTED DATA

When a programmer uses edit-directed data transmission, he or she becomes responsible for every character in each input or output record. FORTRAN underscores this responsibility by requiring that every edit-directed READ, WRITE, or PRINT statement consist of two components:

1. The input or output list. This may be accompanied by control specifiers such as END= or ERR=.
2. A detailed description of the input or output format. This replaces the asterisk used in list-directed statements and guides the program in its interpretation of the input or in its construction of the output.

The format description can be presented to the program in a variety of ways. Regardless of the mechanism used, the description itself is composed of a sequence of *format specifications*. Since each type of specification has its own precise meaning, it produces a specific result each time it is used. The collection of format specifications is like a little language, with its own vocabulary and rules of construction. These are quite simple, so that there will be no difficulty in learning and using them.

### 15.1.1 Interpretation of Edit-Directed Input

Since we are going to describe our input format in complete detail, it now becomes necessary to place the data values much more precisely than we did with list-directed format. FORTRAN does not "recognize" anything in edit-directed format. It treats each individual column as directed by the corresponding format specification. Thus, if a particular input column has a blank in it, it may or may not be part of a data value. Another way of saying this is that the individual data items are not separated physically, as they are in list-directed format. Instead, the end of one item and the beginning of another are defined by the format specifications.

The same holds true for the data value itself. Numbers, letters, and other characters still appear, but it is the format specification, and not the data item, that tells the program how to read the data. For example, when we keypunch a real numerical value for list-directed input, we include a decimal point so that the program will "know" which part

```
7604932185
```

| Statement | | Resulting Values PR | EEC | TMP |
|---|---|---|---|---|
| `READ (5, '(F3.0,F2.1,F5.3)')PR,EEC,TMP` | **(a)** | 760. | 4.9 | 32.185 |
| `READ (5, '(F4.1,F2.2,F4.3)')PR,EEC,TMP` | **(b)** | 760.4 | .93 | 2.185 |
| `READ (5, '(F3.1,F2.0,F4.2)')PR,EEC,TMP` | **(c)** | 76.0 | 49.0 | 43.18 |

Note that the READ statement (in **(c)**, for instance), also could appear as

```
READ (5,FMT='(F3.1,F2.0,F4.2)')PR,EEC,TMP
```

**FIGURE 15.1** READ Statements with Built-In Format Specifications.

of the value was an integer and which part was a fraction. The decimal point is unnecessary with edit-directed formatting, since the format specification contains this information.

We shall take a closer look at this concept by examining the input line represented in Figure 15.1. Only the first ten columns are shown: the rest of the line is of no interest here. To show the effect of the format specifications, we shall examine what happens when this line is processed by each of three separate READ statements. These are also shown in Figure 15.1. Note that the input list is the same for all three statements: Each one specifies the transmission of three data items (PR, EEC, and TMP), each of which is assumed to have been declared as a real variable. The data themselves provide no clue as to how its string of numerical characters is to be interpreted. For all the program "knows," the entire string could be one long value.

The result of a particular READ statement, when applied to those data, will depend on the format description. In these example statements, this description appears as a character string constant right after the unit number. This is just one of several ways to define an edit-directed format. We shall use it for now because it is a most direct form; the complete information is built into the statement itself. A little later, other forms will be introduced in which the READ or WRITE statement refers to a format description instead of giving the description itself.

Let us look at the first case (Figure 15.1(a)). Note that the input list still determines how many items will be read (three in this case) and what type they are to be (real). Moreover, since each READ statement starts a new record, we are at column 1 of the line and the format specifications describe each column from that point. In this example there are three specifications. Each one begins with an F, the code that indicates a description of a real number in conventional form:

1. The first specification (F3.0) says that the first three columns are to be treated as a three-digit real number (F3) having zero decimal places (.0). That is, the .0 tells the program that the *assumed* decimal point (note again that it is not punched) is to the right of the third digit. Thus, the program will pick off the first three columns, treat their contents as 760., and store that value in PR.
2. The second specification (F2.1) describes the next two columns, i.e., columns 4–5. The value found there is treated as a real number with one assumed decimal place. Accordingly, the contents of the two columns are treated as 4.9 and stored in EEC.
3. The final specification (F5.3) describes the next five columns, i.e., 6–10, treating the rightmost three of those five columns as decimal places. Consequently, the value stored in TMP is 32.185 (or more precisely, 0.32185E+02).

We shall apply the second READ example (Figure 15.1(b)) to the same data as if they had not been read before, so that we start at column 1. We are reading into the same three variables as before, but the interpretation has changed becuse of the format specifications:

1. Columns 1–4 are taken as the first value and treated as 760.4.
2. The F2 in the second specification tells the program that the next value (the one to be stored in EEC) is a real number taken from the next two columns (in this case, columns 5–6). The .2 says that both columns are decimal places. As a result, the 93 found in these columns is treated as .93.
3. F4.3, the final specification in the statement, describes the next four columns (7–10) and causes them to be interpreted as 2.185 and stored as such in TMP.

The third statement (Figure 15.1(c)) causes yet another interpretation. Starting again with column 1, the same three variables are provided with values, but only the first nine columns are used. (We can see that easily enough by adding together the first parts of the three specifications, i.e., 3+2+4.) As a result of this statement, PR, EEC and TMP will contain 76.0, 49., and 32.18, respectively.

By now it is apparent that the same physical input can be interpreted in many different ways, and we can control that process with appropriate format specifications. As the next section shows, exactly the same type of control is available for the arrangement of edit-directed output.

### 15.1.2 Appearance of Edit-Directed Output

When we discuss edit-directed output, we usually think of data prepared for examination by humans. This means that the data are arranged carefully so that the values are easy to read and understand. To see how these format specifications help prepare such output, we shall assume that the output unit is a printing device in which each line displays 132 characters. Because of the way FORTRAN is constructed, the *description* of an output line must include an additional character (i.e., the first one) for carriage control. Recall (Chapter 14) that this additional character is used to instruct the printer to start a new line, start a new page, stay on the same line, and so on. We have seen already that when the carriage control character is a blank, the printer starts a new line.

Readable, well-organized output is prepared using the same format specifications as those for edit-directed input. Many of these specifications operate in the same way for reading and writing, while others have somewhat different effects when applited to output. An example of this difference is seen in the F-specification. When applied to an output item, the F-specification produces a printed value that *includes a visible decimal point.* In addition, the value will show a sign if it is negative. Since the sign and the decimal point occupy spaces on the print line, those spaces have to be counted when the size of the value is specified.

To take a first look at how this works, let us suppose that the variable PR currently has a value of −462.7. If we wanted that value to appear somewhere on a print line in exactly that form, −462.7, we must provide room for each of the six characters in that value. Thus, the first part of our specification is F6. Then, recognizing that one of these six characters will be to the right of the decimal point, we shall complete the specification so that its final form (F6.9) includes that fact.

Just as an input format specification determines the number of columns read, it is possible to control the number of columns occupied in edit-directed output. For instance, if we specify F8.1 for the value just used, we do not change the *value* itself. That value, already established in storage under its name (PR in this example) is whatever it is. However, we can change its *appearance* by changing the specification that edits it. Thus, if

we were to specify F8.1 for that value, it would appear as

```
bb-462.7
```

Since the extra two columns are not needed to describe the value itself, FORTRAN automatically fills those (leftmost) columns with blanks. If we were to occupy the same number of columns (eight) but change the specification to F8.2, the value now would appear as

```
b-462.70
```

The *appearance* of the output is helped greatly by the X-specification, a format code that skips input columns and fills output columns with blanks. We shall introduce it for output and use it more generally later on. Its use is simple. For instance, when we specify 12X as an output format, FORTRAN fills the next twelve output columns with blanks. Thus, it is a convenient way to provide open spaces between values on a line.

We shall combine the two types of specifications to prepare a line of print. Let us say that variables PR, TMP, and EEC contain values of -462.70, 4.937, and 2148.033, respectively. We would like to print these values, in the order given, on a line having the following appearance:

1. The first five columns of the line are left blank.
2. PR occupies the next eight columns and is shown to two decimal places.
3. The next ten columns are left blank.
4. TMP's value is placed in the next six columns and is shown to three decimal places.
5. The next ten columns are left blank.
6. The value from EEC is shown to three decimal places in the next nine columns.
7. The remainder of the line is left blank.

We can develop the format specifications directly from these requirements:

1. A specification of 5X takes care of the first five columns.
2. F8.2 describes the eight-column field for PR.
3. A specification of 10x fills the next ten columns with blanks.
4. TMP's appearance is described by F6.3.
5. Another specification fills the next ten columns with blanks.
6. F9.3 describes the appearance of the third value, i.e., EEC. Thus, when we combine these specifications, the result is

```
5X,F8.2,10X,F6.3,10X,F9.3
```

FORTRAN automatically fills the rest of the line with blanks so that we do not have to worry about an additional description for that. However, we still have to take care of carriage control. Since this output is supposed to start a new line, a blank character is needed to direct the printer. Thus, the complete specification says

```
'b',5X,F8.2,10X,F6.3,10X,F9.3
```

When we incorporate these specifications into a WRITE statement (assuming unit 6 to be the output unit), the result can be presented as follows:

```
      WRITE (6,41) PR,TMP,EEC
41    FORMAT ('b',5X,F8.2,10X,F6.3,10X,F9.3)
```

This example shows another way of specifying an edit-directed format. Recall (from the previous chapter) that, in this form, the format is described in a separate statement whose label (41 in this example) is given as the second specification in the WRITE statement's

parenthesized list. If we want to use the other form (the one shown in Figure 15.1) for the input, we have to put double apostrophes around the blank carriage control specifier as follows:

```
WRITE (6, '(''b'', 5X, F8.2, 10X, F6.3, 10X, F9.3)') PR, TMP, EEC
```

Using the same data values, Figure 15.2 shows the effects of several different format specifications on the appearance of the resulting output line.

## 15.2 CONSTRUCTION OF EDIT-DIRECTED INPUT FORMATS

This section introduces additional format specifications and shows their use in conjunction with edit-directed READ statements.

### 15.2.1 The X-Specification and Input

When the X-specification is used with input, the program skips a designated number of input columns. For instance, the statement

```
READ (5, '(7X, F6.2)') COST
```

says, in effect, "Read the next line from unit 5, skipping the first seven columns. Then take the next six columns (8–13) and treat their contents as a six-digit real number having two decimal places. Store that value in COST, forget about the rest of the line, and accept my official thanks."

When the program responds to an X-specification for input, it ignores anything that might be recorded in the skipped columns. Consequently, those columns may or may not be blank; it makes no difference to FORTRAN.

### 15.2.2 Numerical Specifications

In addition to the F-specification, which we have used earlier, there are three codes that extend the versatility for reading numerical values.

**15.2.2.1 The I-Specification** Input values to be interpreted as integers are read with the I-specification. The general form is I*w*, where *W* is the number of columns to be associated with that integer value. For instance, suppose NUM and COST are declared as an integer and real variable, respectively, and we write the statement

```
READ (5, '(4X, I3, 2X, F6.2)') NUM, COST
```

The results can be summarized as follows:

1. Columns 1–4 of the input line are skipped.
2. The contents of columns 5–7 are treated as a three-digit integer and stored in NUM. This assumes, of course, that those columns contain numerical characters. Anything else (e.g., letters, commas, etc.) is treated as an error. Blank characters generally are ignored (as explained in Section 13.2.3.1). A sign (if used) occupies a column, of course, and must be included in the column count.
3. Columns 8–9 are skipped.
4. The contents of columns 10–15 are treated as a six-digit real value with two decimal places and stored in COST.
5. The rest of the line is ignored.

**15.2.2.2 Integer Input In Octal Form** HP FORTRAN 77 accepts input data in octal form and converts each value for subsequent storage in INTEGER *variables*. HP 16-bit computer systems accept up to 6 octal digits while the 32-bit systems accommodate as many as

Assumed values: PR = −462.70
TMP = 4.937
EEC = 2148.033

Output statement: `WRITE (6,26) PR,TMP,EEC`

| If Statement 26 Says | Then the Resulting Line is |
|---|---|
| `FORMAT('b',F8.2,F6.3,F9.3)` | b−462.70b4.937b2148.033 |
| `FORMAT('b',F10.2,F8.3,F11.3)` | bbb−462.70bbb4.937bbb2148.033 |
| `FORMAT('b',F10.1,F8.2,F11.2)` | bbbb−462.7bbbb4.93bbbb2148.03 |
| `FORMAT ('b',F7.2,F5.3,F8.3)` | −462.704.9372148.033 |
| `FORMAT('b',F8.4,F6.4,F9.5)` | 462.70004.9370148.03300 |

**FIGURE 15.2** Effect of F-Specifications on Numerical Output.

eleven octal digits. Since the maximum 16-bit value is 32767, the maximum acceptable octal input value is `177777` (octal). The corresponding maximum for the 32-bit systems is `17777777777`.

The format description is @*w*, 0*w*, or K*w*, where *w* is the number of columns occupied by the octal value. *w* may exceed the maximum limit, but HP FORTRAN 77 will use only as many of the rightmost digits as the particular system's capacity will allow (i.e., 6 or 11). Thus, if we are using a 16-bit HP computer system and we describe the input value

```
b-140773
```

as 08, the eight columns will be picked up, but the value actually stored will be `140773` (octal).

**15.2.2.3 Integer Input In Hexadecimal Form** HP FORTRAN 77 also accepts integer input in hexadecimal form, converting each value to a 16-bit or 32-bit integer depending or the particular HP system being used. The format description is Z*w* where *w* is the number of columns occupied by the hexadecimal value. (The maximum expressible value, then, is `FFFF` for 16-bit systems and `FFFFFFFF` for 32-bit systems.)

**15.2.2.4 The E-Specification for Input** Sometimes it is inconvenient to express a real number in conventional form. For example, a value like `0.00000000437` is handled more easily as `0.437E-8` or `437.0E-11` or simply `437E-11`. Values expressed in this form are described conveniently by the form E*w.d* where *w* specifies the number of columns being described and *d* indicates how many of those columns are decimal places. Thus, the above mentioned value, when expressed as `0.437E-8`, would be described as `E8.3`. If we were to express it as `437.0E-11`, the appropriate E-specification would be `E9.1`. Removing the `.0` and recording the value simply as `437E-11` means that we have shortened its expression by two columns and eliminated the decimal place (in the *appearance*, not in the *value*). Accordingly, the description for that form is `E7.0`.

**15.2.2.5 The D-Specification** Input values intended for storage in double precision (`REAL*8`) variables can be expressed in conventional form. When this is inconvenient, scientific notation can be used here, too. For example, the value −23657984500000 can be expressed as `−236579845D5` or `−236579845D+5` or `−235479845D+05`. The corresponding format descriptions are `D12.0`, `D13.0`, and `D14.0`.

**15.2.2.6 Decimal Points in Real Input Data** As is abundantly clear by now, FORTRAN does not need a physical decimal point in edit-directed input. However, some people feel more comfortable when they see decimal points in their data. Consequently, FORTRAN

accepts such forms. The programmer must remember to count the decimal point as one of the input columns. If an input decimal point is present and its position differs from that described by the format specification, FORTRAN uses what it finds in the data. Thus, if a value is recorded (in its proper columns) as `342.5` and its format description says `F5.2`, the value is interpreted as `342.5`.

**15.2.2.7 Versatility versus Confusion** The numerical format specifications are much more flexible than has been described here. For instance, a real numerical value expressed in conventional, `E-` or `D-` form can be described by any of the three real specifications discussed before (i.e., `F-`, `E-`, or `D-` specifications). However, it is worthwhile resisting the temptation to overuse these powerful facilities. All too often, what seems at first to be a convenience turns out to create excessive complication and needless confusion. For this reason, the following practices are suggested for dealing with edit-directed numerical input:

1. Although this is self-evident, it is worth saying explicitly: When each of a series of data records is to contain similar items, use the same set of columns for a given item throughout the entire collection of records. For instance, if each record contains information about a package to be shipped, and one of the items is its weight, then each package's weight should appear in the same columns of that package's record.

2. In determining how many input columns to reserve for a numerical data item, give careful thought to the range of values this item might have:

   (1) What is the largest value this item could have?

   (2) What is the largest number of decimal places this item could have?

   (3) Will this item ever have a negative value?

Enough room must be provided for the worst possible case. Thus, if only one value in a thousand is likely to be negative, a separate column must be provided for a sign, despite the fact that the sign is not needed except for that one value. This is true even if the negative value will be small, so that it (presumably) would not need all the columns reserved to handle the largest values.

3. When enough room is provided to handle the full range of an item's values, there will be cases where not all the columns are required. In such cases, make sure that the value is properly aligned in the columns reserved for it; otherwise, it could be misinterpreted. For instance, assume columns 15–20 are reserved for a signed real value, and three of those columns are reserved for decimal places. A value of `-15.487` will be recorded as `-15487` and read with a specification of `F6.3`. To be consistent, then, a value of `-3.2` should be recorded as `-03200`. Alternatively, it also can be expressed as `-b3200` or even `-b32bb`. Note that the `32` appears in the same columns in all of these versions. This will assure its proper interpretation.

### 15.2.3 Character Input: The A- and R-Specifications

Edit-directed character input in HP FORTRAN 77 is handled with the A- and R-specifications. The general forms A*w* and R*w* indicate that the next *w* columns are to be treated as a character string. That is, they are to be taken exactly as they appear. When the A-specification is used, HP FORTRAN 77 treats the leftmost characters as being more significant; the R-specification makes the rightmost characters more important. As we shall see, this has an effect when *w* differs from the length of the variable into which the input characters are to be read.

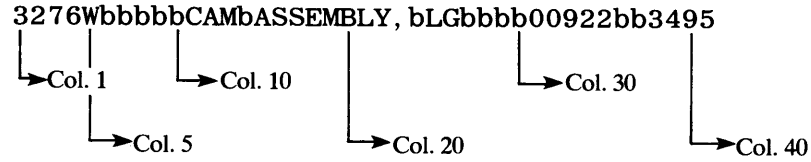
**15.2.3.1 Appearance of Edit-Directed Character Input** Since the A- and R-specifications explicitly direct the program to read the designated columns as characters, there is no need to use apostrophes around the input string. In fact, an apostrophe simply is another character, as is a blank. For instance, consider the following program segment:

```
      REAL          UNITPR
      INTEGER*2     ONHAND
      CHARACTER     IDENT*5, NAME*20
                       .......
                       .......
      READ (5, 12) IDENT, NAME, ONHAND, UNITPR
   12 FORMAT (A5, 5X, A20, I5, 2X, F4.2)
      .......
      .......
```

The description in statement 12 covers the first 41 columns (i.e., 5+5+20+5+2+4) of the input record from unit 5:

| *Columns* | *Interpretation* |
|---|---|
| 1–5 | a 5-character string, stored in IDENT |
| 6–10 | skipped |
| 11–30 | a 20-character string, stored in NAME |
| 31–35 | a 5-digit integer, stored in ONHAND |
| 36–37 | skipped |
| 38–41 | a 4-digit real value with two decimal places, stored in UNITPR |
| 42–80 | ignored |

Thus, when this READ/FORMAT statement combination is applied to the line shown below,




the character variable IDENT will have a value of 3276W, NAME will contain the twenty characters CAMbASSEMBLY, bLGbbbb, ONHAND will contain 922, and UNITPR's value will be 34.95.

Exactly the same results would have been obtained if we had used the R-specification instead of the A-specification. The reason is that, in this particular case, the values for *w* in the specifications matched the respective lengths declared for the character variables (IDENT and NAME).

**15.2.3.2 Reading Parts of Character Strings** If we are going to read input data into a character string variable, it is not necessary for the length of the input string to agree with the declared length of the variable:

1. When the length given by the A-specification is greater than the declared length, the program uses as many of the *leftmost* characters as it can. That is, it fills the variable with input characters and throws away the rest. For instance, if variable WRD is declared as CHARACTER*6 and we try to read eight columns of character data (by specifying A8 in the format description), WRD will end up with the characters taken from the first through sixth columns of the input string. The last two columns will not be stored. These surplus characters fall into a polystyrene Standard Character Bucket (SCB) inside each computer. Every nineteen days, the contents are gathered and sent to a huge Standard

Character Warehouse (SCW) somewhere in Idaho. There, they are cleaned, polished, sorted, and stored away for use later on.

2. When the length given by the A-specification is less than the declared length, the *entire* string in the variable is replaced nonetheless. the program fills the string (starting at the left) until it runs out of data, and then it pads the rest of the string with blanks. Some blanks come from Idaho; others are manufactured on the spot.

We can see the effect of the A- versus R-specification directly by setting up the following declarations:

```
CHARACTER*6 WORD1
CHARACTER*4 WORD2
```

Now, let us assume an input line on unit 5 with the following data, starting with a D in Column 1:

```
DECONSTITUTIONALIZATION
```

and the statement

```
READ (5, '(2X, A3, 5X, A7)') WORD1, WORD2
```

The first A-specification covers a shorter string than WORD1 can hold, and the second one's length exceeds that declared for WORD2. As a result, the program picks up Columns 3, 4, and 5 (after skipping the first two) and stores them in the leftmost three positions of WORD1. The remaining three positions are filled with blanks. Then, after skipping the next five columns (the ones containing STITU), the next seven columns (TIONALI) are picked up, but only the *leftmost* four are used. (WORD2 cannot hold more than four characters.) Thus, we end up with CONbbb in WORD1 and TION in WORD2.

Now we shall use the same input another time, changing only the A-specifications to R-specifications:

```
READ (5, '(2X, R3, 5X, R7)') WORD1, WORD2
```

The same three columns (CON) are picked up as before, but they are stored in the three *rightmost* positions of WORD1. The leftmost three positions are padded with binary zeros, *not* blanks. The R7 gets the same seven characters as before (TIONALI), but the *rightmost* four are used to fill WORD2. Thus, the two variables end up with respective values of zzzCON and NALI. (zzz indicates binary zeros.)

**15.2.3.3 Substrings as Input Items** FORTRAN also enables us to read in part of a string by accepting a substring as a member of an input list. For instance, suppose a variable named TEXT is declared as CHARACTER*25 and currently contains the string

```
BRIGHTbSTARSbAREbSHININGb
```

If we were to read the following line

```
SILVERbMOONS
```

with the statement

```
READ (5, '(A12)') TEXT(1:12)
```

the program would read the first twelve columns and store them in positions 1–12 of TEXT, replacing *only those twelve characters*. Accordingly, the new value in TEXT would be

```
SILVERbMOONSbAREbSHININGb
```

If we had read that same line with the same format description but without the substring

designation in the input list, i.e.,

```
READ (5, '(A12)') TEXT
```

the resulting value in TEXT would have been

```
SILVERbMOONSbbbbbbbbbbbb
```

### 15.2.4 Multi-Record Input Formats: The /-Specification

Since a statement's input list determines the number of items to be read by that statement, the program (guided by the format specifications) will continue to read until it completes the list's requirements. Accordingly, it will read as many lines as it needs.

The programmer can exercise more direct control over such input by using the / specification. When used as part of an input format description, this serves as a signal to skip the rest of the line currently being read and start with column 1 of the next one. To see how this works, let us consider the following program segment:

```
      REAL              V1, V2, STR, BLK
      INTEGER*2         CD1, CD2
      CHARACTER*6       ID1, ID2
        .......
        .......
      READ (5, 18) ID1, CD1, V1, V2, ID2, CD2, STR, BLK
   18 FORMAT (A5, I2, 20X, F7.2, 10X, F7.2/A5, I2, 15X, F6.3, 15X, F6.3)
        .......
```

The input statement reads exactly two lines with the information being taken from the following columns:

| *Columns* | *Interpretation* |
|---|---|
| 1–5 | Stored as a character string in ID1 |
| 6–7 | Stored as an integer in CD1 |
| 8–27 | Skipped |
| 28–34 | Stored in V1 as a real number with two decimal places |
| 35–44 | Skipped |
| 45–51 | Stored in V2 as a real number with two decimal places |
| 52–80 | Skipped (in response to the /) |
| 1–5 (second line) | Stored as a character string in ID2 |
| 6–7 | Stored as an integer in CD2 |
| 8–22 | Skipped |
| 23–28 | Stored in STR as a real number with three decimal places |
| 29–43 | Skipped |
| 44–49 | Stored in BLK as a real number with three decimal places |
| 50–80 | Ignored |

Note that commas are not needed around the slash. We could have done the same thing with different forms. For instance, we could have written

```
      READ (5, 18) ID1, CD1, V1, V2, ID2, CD2, STR, BLK
   18 FORMAT (A5, I2, 20X, F7.2, 10X, F7.2, 29X, A5, I2, 15X, F6.3, 15X, F6.3)
```

or even

```
      READ (5, 18) ID1, CD1, V1, V2
   18 FORMAT (A5, I2, 20X, F7.2, 10X, F7.2)
      READ (5, 19) ID2, CD2, STR, BLK
   19 FORMAT (A5, I2, 15X, F6.3, 15X, F6.3)
```

Use of the / specification certainly is more convenient than either of the two other forms shown above. The two separate READ statements are particularly undesirable since they force the programmer to form two artificially separate input groups when, in fact, he or she may wish to treat the entire list as a single collection.

The slash may be used as many times as desired in a format description. Each time it is used, the program merely skips to the beginning of the next record. Thus, if there are two slashes in succession (//), the program will skip to the end of the current record, skip the entire next record, and be positioned at column 1 of the record after that.

### 15.2.5 Disagreements Between Input Lists and Format Descriptions

In the interest of simplicity and clarity, it is best to make sure that the format specifications are consistent with the input list. For example, if an input list names five items, the corresponding format specifications should describe five items as well. (Of course, this has nothing to do with the number of X-specifications; these are used as needed to indicate which columns are to be skipped.) However, it will be helpful to understand how FORTRAN behaves when there is a mismatch between the length of the list and the number of specifications. Keep in mind that the program will try to read as many items as the input list specifies.

1. When the input list contains fewer items than the format description, the program simply will use as many format specifications as it needs, ignoring the rest. For example, if V1, V2, and V3 are real variables, the statements

```
      READ (5, 15) V1, V2, V3
15    FORMAT (3X, F5.1, 5X, F4.2, 7X, F6.3, 6X, F5.2, 4X, F4.0)
```

will take the first value (the one to be stored in V1) from columns 4–8, the second value from columns 14–17, and the third one from columns 25–30. The rest of the specifications in statement 15 will be ignored. It is the same as if they were not there. Thus, if the READ statement were in a loop, then the next time it is used, it simply will start with a new line, using the same columns as before.

2. When the reverse is true, i.e., the input list is longer than the number of format specifications describing it, the program still will try to read the number of items indicated in the list. To do this, it will use each specification in turn. When the specifications run out, the program goes back and starts at the beginning of the list, continuing this way until the input list has been satisfied. For instance, suppose V1, V2 and V3 are real variables as in the previous example. This time, we attempt to read values for them with the following statements:

```
      READ (5, 25) V1, V2, V3)
25    FORMAT (10X, F5.1, 4X, F4.2)
```

Note that the input list shows three items but the FORMAT statement describes only two. Hmmm. What happens is the following: The program will skip the first ten columns and read columns 11–15 into V1. Then, as expected, it will skip the next four columns (16–19) and read columns 20–23 into V2. So far so good. Then, having run out of descriptions, the program goes back to the beginning of the FORMAT statement, *starts a new line*, skips the first ten columns of that second line, and stores columns 11–15 in V3 using the F5.1 specification given in statement 25 for those columns. The remaining format descriptions are ignored because the program does not need them. The input list has been satisfied and FORTRAN is lying there, digesting quietly. Of course, when the program goes to use this READ statement again, it will start a new line, skipping the first ten columns of that line, and so on.

## 15.3 CONSTRUCTION OF EDIT-DIRECTED OUTPUT FORMATS

FORTRAN provides a wide range of format specifications for arranging output records. This section explains their operation and examines a variety of techniques for their use.

### 15.3.1 Blank Output Columns: The X-Specification

Since we have seen and used the X-specification in a variety of examples, a detailed discussion of its properties is no longer necessary. Accordingly, we just shall note that the specification

*w*X

fills the next *w* columns of the output line with blanks.

### 15.3.2 Carriage Control

We are already well aware of the fact that the format description of a print record must begin with a character that the program uses for carriage control. The various codes for such control are given and illustrated here.

**15.3.2.1 Starting a New Line** As we know, a blank carriage control character causes the program to start a new line. We have been including this blank by specifying the blank as a *literal character*, i.e., `'b'` at the beginning of an output format description. Another way of doing this is to use the X-specification. Since the program uses this code as a signal to insert blanks in the print line, it can serve equally well to indicate the start of a new line. For instance, the statements

```
      WRITE (6, 12) XVAL
12    FORMAT (1X, F8.2)
```

will print the value from XVAL as a real number with two decimal places in columns 1–8 of a new line. Similarly, the statements

```
      WRITE (6, 14) XVAL
14    FORMAT (1X, 9X, F8.2)
```

will print XVAL's value in columns 10–18 of a new line. Note that the 1X is *not* a skipped column; it positions the printer at column 1 of a new line, so that the 9X specification applies to the first nine columns of that line. Hence, the program is ready to use the next eight columns (i.e., 10–17) for XVAL. Incidentally, the same thing would happen with the statement

```
      PRINT 14, XVAL
14    FORMAT (1X, 9X, F8.2)
```

When we want to skip the first few columns before printing an output value, it might seem reasonable to combine the two X-specifications, so that the previous FORMAT statement could be rewritten as

```
14    FORMAT (10X, F8.2)
```

This certainly is legal, but not particularly clear. Even old, grizzled FORTRAN veterans, their leathery faces weatherbeaten from past storms, occasionally forget that the first blank is for carriage control. As a result, they miss on the column count. Of course, this is no great tragedy, but it is just as easy to be clear. Consequently, we shall show the carriage control character as a separate specification.

**15.3.2.2 Other Carriage Control Symbols** There are three other carriage control symbols in common use:

1. The numeric character zero (`'0'`) skips the next line and starts the print process at column 1 of the line after that.

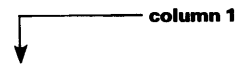
2. The numeric character '1' starts the print process at the top of a new page.
3. The character '+' forces the print process back to the beginning of the same line that was printed by the previous output statement. It is then ready to print again on the same line.

**15.3.2.3 Overprinting** The ability to prevent the printer from moving to the next line enables the programmer to specify *overprinting*, a technique for displaying several characters in the same space on a line. One obvious use for overprinting is to print the same value (or name, or whatever) over and over in the same spot, thereby making it come out much darker, for emphasis. For instance, suppose we wanted to print three values WST, BLR, and TRK on the same line, with the middle value (BLR) appearing much darker. Let us say that some experiments have shown that we can get the emphasis we want with four overprints after the initial printout. Here is one way to set this up:

```
      .......
      .......
   PRINT 71, WST, BLR, TRK
71 FORMAT ('b', 12X, F9.3, 8X, F9.3, 8X, F9.3)
   DO NUMOVR = 1, 4
     PRINT 81, BLR
81   FORMAT ('+', 29X, F9.3)
   END DO
      .......
      .......
```

The first PRINT statement displays the original line, with all three values appearing in the positions determined by the format specifications. That statement is executed once. Then, a second PRINT statement is used to display BLR by itself. Its FORMAT statement (no. 81) uses the carriage control character that forces the process back to column 1 of that same line. By skipping the first 29 columns, we line up with the position where BLR was printed by the previous output statement. More precisely, we are overprinting the entire line, filling all of it with blanks except for columns 30–38, where BLR appears. Finally, by putting that PRINT statement in a DO loop, we execute it four times.

Assuming values of −36.414, 3275.01, and 5987.088 for WST, BLR, and TRK, respectively, the resulting output can be represented as follows:

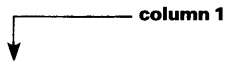



```
bbbbbbbbbbbbbb−36.414bbbbbbbbb3275.010bbbbbbbbb5987.088
```

Another common use for overprinting is to underline selected items in a line of print. For instance, if we wanted to underline WST in addition to printing BLR with great emphasis, we could do the following:

```
      .......
      .......
   PRINT 71, WST, BLR, TRK
71 FORMAT ('b', 12X, F9.3, 8X, F9.3, 8X, F9.3)
   PRINT 91
91 FORMAT ('+', 12X, '_____')
   DO NUMOVR = 1, 4
     PRINT 81, BLR
81   FORMAT ('+', 29X, F9.3)
   END DO
      .......
      .......
```

Note that the second PRINT statement has no output list. This is quite legal, its meaning being that we do not wish to display any values of variables from storage. Instead, everything that we are going to print is contained right in the FORMAT statement. In this particular example, it is the *literal value* shown in the apostrophes. As a result, the program will go back to the beginning of the line it just printed (because of the '+'), put blanks in the first twelve columns, and then print the nine underline characters in the next nine columns. Those also happen to be the same columns in which we printed WST with the previous output statement. Thus, when the DO loop finishes its four cycles, the final result will be



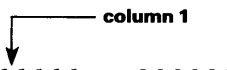

```
bbbbbbbbbbbbb–36.414bbbbbbbbb3275.010bbbbbbbbb5987.088
```

**15.3.2.4 The /-Specification for Carriage Control** The / specification, when used in an output format description, has the same basic effect as it does for input: It causes the program to go immediately to the end of the current record. For printed output, this means that the printer gets itself to the end of the current line. This does *not* imply that it starts the next line. The action taken after the / specification still must be specified by a carriage control character following the slash.

To illustrate, let us assume that N1, N2, N3 and N4 are integer variables containing values of 123, 345, 567, and 789, respectively. If we print these values with the statements

```
      WRITE (6,16) N1, N2, N3, N4
   16 FORMAT (1X,5X,I3,6X,I3/5X,I3,6X,I3)
```

the resulting output will look like this:




```
bbbbb123bbbbbb345
bbbb567bbbbbb789
```

Note that the second line of print starts one column earlier than the first line. The reason is that FORTRAN uses the first of the five blanks (from the 5X specification immediately following the slash) as a carriage control character. That gets the printer to the next line. Thus, only four blanks are inserted in the print record, and the first value on that line starts in column 5 instead of column 6. Once this is understood, the remedy is obvious: We can rewrite the FORMAT statement as

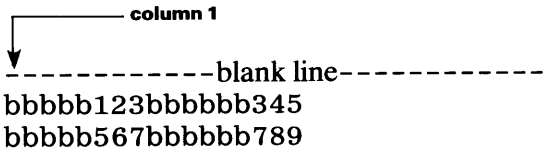
```
   16 FORMAT (1X,5X,I3,6X,I3/1X,5X,I3,6X,I3)
```

and the two output lines will be lined up with each other. The same thing applies when the slash is placed at the beginning of a format description. Thus, suppose we specify the following output:

```
      WRITE (6,26) N1, N2, N3, N4
   26 FORMAT (/'b',5X,I3,6X,I3/'b',5X,I3,6X,I3)
```

In this case, the program will move the printer to the end of the current line (the one it was ready to use before the slash was encountered), and then start a new line in response to the blank carriage control character. The first two values (N1 and N2) are printed on that line. The second slash moved the printer to the end of that line and the blank carriage control character starts the line after that. Thus, statement 26 produces three lines of output: a blank line followed by two lines of print, each containing three integer values.

This is shown below:




### 15.3.3 Formatting of Numerical Output

The I, F, E and D specifications are used for preparing output in the same basic way as they are for interpreting input. There are some slight differences, however, and these will be discussed here.

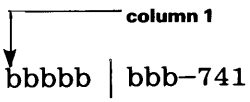
**5.3.3.1 Integer Values** An integer output value is described by the specification

Iw

where *w* indicates the width of the value (that is, the number of columns that the value will occupy on the print line). FORTRAN will use as many of the rightmost columns as it needs, filling the rest with blanks. For instance, if integer variable NUMBLK contains a value of −741 and we write the statements

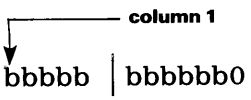
```
    PRINT 17, NUMBLK
17  FORMAT ('b', 5X, I7)
```

the following will be printed on a new line:




The first five blanks will be inserted because of the 5X specification; the other three ahead of the −741 are there because the I7 specification called for seven columns to be filled. (The rest of the line, of course, is blank as well, but is not shown here.)

Notice that FORTRAN filled the unused columns in the integer field with blanks. If the value in an integer variable happens to be zero and we ask for it to be printed with an Iw specification, it would seem that we would get an invisible display of *w* blanks. However, FORTRAN has a special (automatic) test for such a situation. When it occurs, FORTRAN will show one zero to indicate that the value has, indeed, been printed. Thus, if we use the abovementioned statements and NUMBLK happens to have a value of zero, the resulting line will look like this:




The programmer can control FORTRAN's use of blanks to pad an integer output field by using the specification

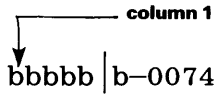
Iw.v

where *w* has the same meaning as before and *v* indicates how many of *w*'s columns are to be visible. When this type of I-specification is given for an output value, FORTRAN still uses a field of *w* columns. However, once it places the value itself in the rightmost part of the field, it inserts additional zeros until a total of *v* columns are filled. Then, if it still has not filled all *w* columns, it completes the rest of the field with blanks. We can see how this works by using the previous value as an example. Keeping the same PRINT statement and

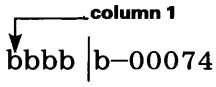
changing the FORMAT statement as follows,

```
      PRINT 17, NUMBLK
   17 FORMAT ('b', 5X, I7.6)
```

our line of print will say




The five blanks at the beginning of the line still are there because of the 5X, and NUMBLK still occupies seven columns because of the I7. However, since our new specification indicates that six of the seven columns are to be visible, FORTRAN replaces two of the blanks with zeros and moves the minus sign so that it is positioned to the left of the leftmost visible digit. The seventh (leftmost) position still remains blank. If the specification had said I7.7 (with the rest of the format remaining the same), the line would show




A specification like I7.8 would be illegal since we would be asking for a number of visible digits greater than the length of the field itself.

The O-, K- and @-specifications can be used to show values from INTEGER*2 variables as octal numbers. If *w* is greater than 6, the rightmost six columns of the field will be used to display the octal digits, and the preceding ones will be padded with blanks. If *w* is not long enough, the *w* positions will be filled with the rightmost octal digits. (NOTE: There is no explicit indicator given with the output to identify the digits as octal digits. Any visible identification must come from the programmer.)

**15.3.3.2 Real Output in Conventional Form** As we have seen already, the F-specification, when used for output, includes a decimal point (and a minus sign for negative values). Consequently, the programmer must make sure that the specification provides enough room to include these symbols. For instance, if TRANS is a real variable whose value (–42.674) we want printed as a five-digit number with three decimal places, the specification will be *no smaller than* F7.3. If *w* (the field width) is any larger, FORTRAN will add blanks. A specification of F9.3, for instance, will produce bb–42.674. If the value happens to be positive, FORTRAN will place a blank in the column where the minus would have gone. Thus, an F9.3 specification applied to a value of 42.674 will print bbb42.674.

**15.3.3.3 Real Output in Single Precision Form** Real numbers can be printed in single-precision notation with the specification

E*w*.*d*

where *d*, as before, indicates the number of decimal places. The field width, *w*, must be large enough to include columns for various symbols that are part of the notation, as well as for the decimal places. For instance, let us use the same value (–42.674) as we did in the previous section. If we wanted to print that value in single precision form showing all of its digits, we would have to provide enough space for FORTRAN to display it as

–0.42674E+02

Thus, in addition to the columns for the five actual digits in the fraction, we need to specify *at least* seven more to take care of the minus sign, the zero in front of the fraction, the decimal point, the "E," the exponent sign, and the two-digit exponent value. That means that the value shown above would be produced by a format specification of E12.5. A

field width of anything less than 12 (for this example) would produce an improper value. If we were to specify more than 12, the additional spaces would be filled with blanks. For instance, a format specification of E14.5 for the same value will print it as bb–0.42674E+02 and a specification of E14.6 will produce b–0.426740E+02. On the other hand, if the specified field width were large enough to display the entire value, but the number of decimal places was insufficient, FORTRAN would not make any adjustments; it follows the format specification exactly. Thus, if we want to display a value of –42.674 in single precision form and we describe its format with a specification of E14.4, the printout will show bbb–0.4267E+02.

When used for output, the E-specification can be extended to define a different number of digits for the exponent. (When we use the basic form E*w*.*d*, FORTRAN automatically provides a two-digit exponent.) If we want to change that for a particular output item, we can describe that by specifying

E*w*.*d*E*e*

In this form, the second E is followed by an integer indicating the number of digits to be used for the exponent. For instance, if we were to print the value –42.674 with the specification E14.5E3, the result would be b–0.42674E+002.

**15.3.3.4 Real Output in Double Precision Form** Representation of output values in double precision form is handled by the D-specification

D*w*.*d*

where *w* and d have the same meanings as in the F and F specifications. This output has the same basic appearance as E-formatted output. The only difference is that the E in front of the signed exponent value is replaced by a D. Thus, if we printed the value –42.674 with the specification D16.8, the resulting output would appear as b–0.42674000D+02.

### 15.3.4 Formatting of Character String Output

FORTRAN enables the programmer to specify the exact placement of output character variables. In addition, there are two methods for including literal character strings in format specifications.

**15.3.4.1 The A- and R-Specifications for Output** The operation of these format descriptions is basically the same as for input (15.2.3). When we specify A*w*, the next *w* output columns are filled with the character string named in the corresponding item of the output list. For instance, suppose WORD has been declared as CHARACTER*7, it has a value in it, and we prepare to print it with the statements

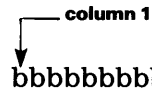
```
      PRINT 14, WORD
   14 FORMAT ('b', 10X, A7)
```

the result will be a new line of print with the first ten columns left blank and columns 11–17 filled with the seven characters from the variable WORD.

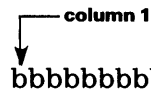
If *w* specifies a length greater than that of the character string named in the output list, FORTRAN prints the string starting at the left and pads the unused positions with blanks. Thus, if WORD contains the string BRIMFUL and we say

```
      PRINT 14, WORD
   14 FORMAT ('b', 10X, A11)
```

FORTRAN will produce the following line:



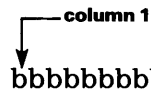

The R-specification, when used under the same conditions, displays the string in the rightmost positions of the field and fills the remaining ones with blanks. So, `R11` instead of `A11` produces



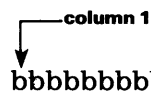

A *w* that is smaller than the string's length will force FORTRAN to print the *w* leftmost characters, with the rest simply not being displayed. Using the same value for `WORD`, the statements

```
      PRINT 14, WORD
14    FORMAT ('b', 10X, A5)
```

will produce



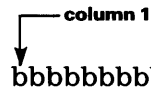

Changing the `A5` to `R5` produces




It also is possible to leave the character counting to FORTRAN. Since FORTRAN has to keep track of string lengths anyway, the programmer can take advantage of that capability by omitting the *w* in the A-specification. Using `WORD` again, if we were to say

```
      PRINT 14, WORD
14    FORMAT ('b', 10X, A)
```

FORTRAN, not finding a length attached to the A- or R-specification, would use the declared length of `WORD`, so that the resulting printout would be



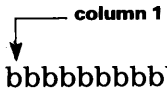

This is a convenient feature if the programmer knows exactly what is going on, or if he/she is not too concerned with the exact position of the output items. In general, it is a good idea to go through the extra work (it is not that much) and include the *w* as part of the specification. That way, whenever anyone looks at the FORTRAN statements, there is a clear, precise, and complete story as to what is going on. (We mention this option so that the student knows it is available; however, we shall not encourage its use. For the sake of completeness, we mention also that the same capability is available for input. That is, if *w* is not included as part of a character specification for an input format, HP FORTRAN 77 will use a number of input columns equal to the declared length of the character variable in which the data are to be stored. Again, this is shaky practice and it is discouraged here.)

**15.3.4.2 Writing Parts of Character Strings** As we have seen repeatedly, once we have identified a portion of a character string (i.e., a substring), it can be treated as a separate

string. Accordingly, it is quite consistent to be able to write a substring as an output item. For instance, if we use WORD (and its value) from the previous section, we can say something like

```
      PRINT 14, WORD(5:7)
   14 FORMAT ('b', 10X, A3)
```

in which case only the last three characters in WORD (i.e., positions 5 through 7) will be shown:

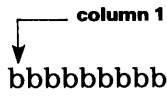



```
bbbbbbbbbbFUL
```

Even if we specify more room in the format description, it is the item in the output list (as always) that determines what is printed. Thus, if we said

```
      PRINT 14, WORD(5:7)
   14 FORMAT ('b', 10X, A9)
```

the resulting printout will be longer, but the same amount of WORD will be shown. Of course, it will not *look* different because FORTRAN pads with blanks anyway. However, we can show what actually happens by making the blanks "visible":

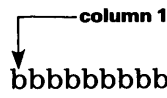



```
bbbbbbbbbbFULbbbbbb
```

**15.3.4.3 Literal Output Strings** Since we have been specifying literal output in list-directed format almost from the beginning, there is nothing new in applying the idea to edit-directed output. Thus, the two statements

```
      WRITE (6,16) 'FINAL RESULTS: '
   16 FORMAT (1X, 10X, A15)
```

produces the output




```
bbbbbbbbbbFINAL RESULTS: bbbbbbbbbb
```

starting in column 11 of a new line. The 15-character literal is treated just like any other character string, its placement and appearance being governed by the same formatting rules described before. (HP FORTRAN 77 also accepts '' as a delimiter for literal output.)

Another possibility, left over from earlier FORTRAN versions, is to place the literal string in the format specification rather than in the output list. (We have used this form earlier, but its specific purpose was carriage control.) The result shown above also can be obtained with the statements

```
      WRITE (6,16)
   16 FORMAT (1X, 10X, 'FINAL RESULTS: ')
```

This avoids the need to specify the length of the literal string. FORTRAN will do the counting automatically. As was pointed out earlier (Section 15.3.4.1) with regard to using the A-specification without the *w*, this may or may not be a convenience, and the programmer must judge whether or not this feature can be used to advantage in a given situation.

There is another form for specifying literal character strings. At one time, it was the only available way, but the newer alternatives just described make it the least convenient.

Here it is, for old times' sake: Output literals may be described with the Hollerith specification. This consists of the length of the string, followed by the letter H, followed by the literal string, without the apostrophes. Thus, the example used earlier can be rewritten as

```
      WRITE (6, 16)
   16 FORMAT (1X, 10X, 15HFINAL RESULTS: )
```

Note that the first character in the literal string follows the H immediately.

## 15.4 ADDITIONAL FORMAT SPECIFICATION TECHNIQUES

At the beginning of this chapter, we referred to FORTRAN's format specifications as a little language. Now that we have seen how several of its important components operate, we shall increase our vocabulary by introducing some additional features for building more extensive sets of specifications.

### 15.4.1 Repeated Specifications

Although each format specification follows a simple structure, combinations of such structures build up quickly. As a result, it is quite possible to develop an input or output process requiring a format description containing several dozen specifications. When this happens, there are features that may help simplify the way in which such formats are represented.

One such feature is the *repetition factor*. When several identical format specifications appear together, they can be replaced by a single specification and a multiplier (the repetition factor) that indicates the number of uses. For instance, the format statement

```
   12 FORMAT (5X, I3, I3, I3, I3)
```

can be rewritten as

```
   12 FORMAT (5X, 4I3)
```

The repetition factor in this case is 4. As another example, suppose that the real variables V1, SPEC, and WRTH are recorded on an input line in columns 7–10, 11–14, and 15–18, respectively. Each value is recorded as four digits and, in each case, two of the four digits are to be treated as decimal places. (The decimal point is not included.) In other words, each variable is described by the specification F4.2. The following statements

```
      READ (5, 15) V1, SPEC, WRTH
   15 FORMAT (6X, 3F4.2)
```

will read and store the three values as required. This produces the same result as if statement 15 were written like this:

```
   15 FORMAT (6X, F4.2, F4.2, F4.2)
```

Repetition factors are used in exactly the same way for output, including literal strings. For example, HP FORTRAN 77 accepts statements such as

```
      WRITE (6, 16)
   16 FORMAT (1X, 3'   NEXT')
```

in which case the specified string is written three times.

### 15.4.2 Repeated Combinations of Format Descriptions

Repetition factors are not limited to single format specifications. When a string of several specifications repeats itself for a given input or output file, that entire string can be

isolated as a pattern, described once, and equipped with a repetition factor. The pattern is identified as such by putting it inside a set of parentheses.

For example, suppose that we have an array VOLS consisting of 25 real values and we want to print VOLS(5) through VOLS(9) on a single line. Each value is to appear in the form vvv.vvvv with an additional position reserved at the left of each value in case a given value turns out to be negative. Accordingly, the format of each value can be described as F9.4 (i.e., four decimal places, plus one position for the decimal point, plus three positions for the integer portion, plus one more position for the sign (blank for a positive value, − for a negative value). Also, the values are to be separated by eight blanks, and the first value is to be printed starting in column 6. When we put all of that together, we end up with the following statements:

```
      PRINT 18, (VOLS(NUM), NUM=5,9)
18    FORMAT(1X,5X,F9.4,8X,F9.4,8X,F9.4,8X,F9.48X,F9.4)
```

Note that the combination F9.4,8X forms a repeating pattern. Thus, we can simplify the FORMAT statement without hiding its meaning:

```
18    FORMAT (1X,5X,4(F9.4,8X),F9.4)
```

FORTRAN will interpret this description by starting a new line in response to the blank carriage control character (as it always does), placing blanks in columns 1–5, and then placing VOLS(5) in columns 6–14, blanks in 15–22, VOLS(6) in columns 23–31, blanks in 32–39, VOLS(7) in columns 40–48, blanks in 49–56, VOLS(8) in columns 57–65, and blanks in 66–73. That takes care of the pattern in parentheses, repeated four times. The remaining specification takes care of VOLS(9), which is placed in columns 74–82 of the print line, thereby completing the output.

**15.4.2.1 Formation of Repeated Patterns** Since an X-specification at the end of an output format has no actual effect, many programmers will add an X-specification in those cases where it will complete a pattern. For our example, we can do this to produce a more concise statement that does the same thing as the previous versions:

```
18    FORMAT (1X,5X,5(F9.4,8X))
```

**15.4.2.2 Mismatches with Repeated Patterns** Earlier, it was established that the input or output list, and not the format description, determines the number of data items to be read or written by a particular statement. Consequently, if there are more data items than format specifiers, FORTRAN will use them over (as long as the type of specifier describes the data properly) until the data list is satisfied. In the case of a simple format list, FORTRAN goes back to the beginning of the list. When the list contains a repeated pattern in parentheses, FORTRAN uses that pattern, rather than the beginning of the format list, as a new starting point.

We can use the format list from the previous section to show how this works. Recall that it said

```
18    FORMAT(1X,5X,5(F9.4,8X))
```

Now, just to illustrate, suppose we try to write seven items using this statement:

```
      WRITE(6,18) (VOLS(I),I=3,9)
```

As a result, FORTRAN will print the first five items (VOLS(3) through VOLS(7)) in the columns described in the previous section, leaving eight blanks between the nine-column fields. Then, since there are two items still to be printed (i.e., VOLS(8) and VOLS(9)), FORTRAN ignores the 1X,5X part of the format description, *ends the output record*, and starts to use the 5(F9.4,8X) pattern again. Since the output record is ended (just as if there had been a slash in the format specifications), the first thing FORTRAN looks for is

a carriage control character. More precisely, FORTRAN tries to use the first data character it finds for carriage control. In this case, it will be the first digit of VOLS(8). If that digit happens to be a 1, FORTRAN will start a new page (because 1, as we know, is the carriage control character for a new page). If it is a 0, FORTRAN will skip a line and start at the beginning of the next line. If it is anything else, FORTRAN just will start a new line. In any case, that first data character is not printed. For instance, if VOLS(8) happens to be −326.0974, the first character (−) will be lost: FORTRAN simply will single space to the next line and print 326.0974. Then, it leaves eight blanks and prints VOLS(9). All of VOLS(9) will appear because FORTRAN still is working on the same specification (the 5(F9.4, 8X)). If it were necessary, FORTRAN would use all five repetitions. Then, if the output list still were not completed, FORTRAN would end the record and start the pattern over, using the first character of the next data item as the carriage control character.

The message is quite clear: Whenever possible, make sure that there is a close match between the items in the data list and the specifiers in the format description. Specifically, it is not a good idea to allow the program to run out of specifiers so that it is forced to double back on the ones that are there. Under certain carefully controlled conditions, this can be done without difficulty, as Section 15.4.4 will show. However, it should not be done indiscriminately.

### 15.4.3 Nested Format Patterns

Occasionally, formats come up in which a repeating pattern is built from other, smaller repeating patterns. Such constructions also can be expressed more concisely by nesting parenthesized patterns inside each other. For example consider the following format description:

```
(10X,F3.1,3X,F3.1,3X,I2,I2,I2,F3.1,3X,F3.1,3X,I2,I2,I2,2X,A4)
```

We see that F3.1, 3X and I2 are descriptions that are repeated. Accordingly, we can replace them with shorthand versions so that our format would look like this:

```
(10X,2(F3.1,3X),3I2,2(F3.1,3X),3I2,2X,A4)
```

Now, we see that the pattern 2(F3.1, 3X), 3I2 appears twice in succession. Therefore, it can be shortened even further by applying a repetition factor to that entire pattern:

```
(10X,2(2(F3.1,3X),3I2),2X,A4)
```

Another aspect of this capability is illustrated by the following format:

```
(10X,I2,3X,I2,3X,A3,F3.2,4X,I2,3X,I2,3X,A3,F3.2,8X,A6)
```

An initial use of the shorthand form reduces the format to

```
(10X,2(I2,3X),A3,F3.2,4X,2(I2,3X),A3,F3.2,8X,A6)
```

Now, we see that we could form a repeating pattern out of the 2(I2, 3X), A3, F3.2 combination, except that the 4X gets in the way. In many instances, it is possible to overcome this difficulty by playing around with the format specifications so that we say the same thing in a different way. In this example, we have managed to be fortunate: The 8X happens to be in the right place, and we can rewrite it as two consecutive 4X specifications. This still provides eight blank spaces, but look what it does to the format description:

```
(10X,2(I2,3X),A3,F3.2,4X,2(I2,3X),A3,F3.2,4X,4X,A6)
```

Now, our pattern is complete, and we can rewrite the format as

```
(10X,2(2(I2,3X),A3,F3.2,4X),4X,A6)
```

Note that we could have done the same kind of thing by breaking the 10X at the front of the format into two pieces:

```
(6X, 2 (4X, 2 (I2, 3X) , A3, F3. 2) , 8X, A6)
```

This type of game-playing may or may not be desirable. Making a format description shorter does not necessarily make it simpler. The main purpose, as it always has been, is to increase simplicity and clarity. If repetition factors, nested factors, and patterns (with or without dazzling footwork) do not help make a program easier to follow and understand, then it is better not to use them. Under those conditions, they are cures looking for a disease.

As is true with simpler format patterns, FORTRAN will use a pattern over and over when the format description covers fewer items than are present in the input or output list. When several repeated patterns are present in a single FORMAT statement, FORTRAN goes back to the *rightmost* pattern and uses that one repeatedly, ending a record each time the complete pattern is used.

### 15.4.4 Automatically Repeating Format Descriptions

FORTRAN attaches special meaning to a parenthesized list of format specifications without a repetition factor. Basically, this form says, "repeat as necessary." For instance, suppose we declared a 100-element one-dimensional real array named TRX. Each set of input data includes a collection of elements which may or may not fill the entire array. To support this flexibility, these input elements are preceded by a line containing three integer values:

1. NUM, the number of elements in that input set.
2. LOWER, the position in which the first input element is to be stored.
3. UPPER, the position in which the last input element is to be stored.

(We assume that the input elements are to be stored in consecutive positions.) These are recorded, respectively, in columns 1–3, 6–8, and 11–13. The elements themselves are recorded four to a line, in columns 3–6, 9–12, 15–18, and 21–24. Each element is to be read with an F4. 1 format. Thus, we have a situation where each input set consists of a single line that helps describe the elements, followed by however many lines containing the elements themselves. Since the program cannot anticipate the number of input elements in any given set, we need a convenient way to specify the general input process so that it will work for the entire range of possibilities.

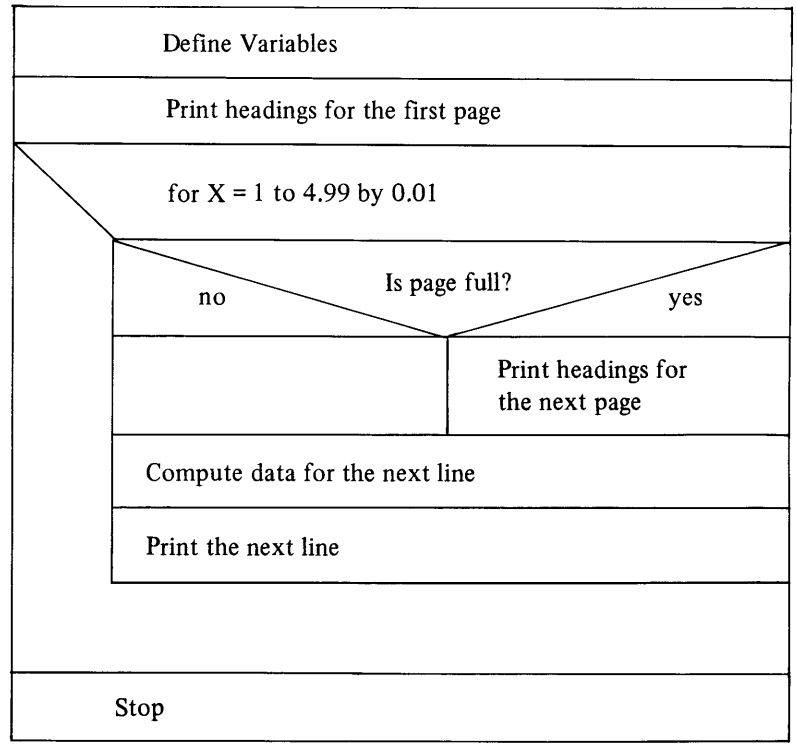
These requirements can be met with the statements

```
    READ (5, 15) NUM, LOWER, UPPER, (TRX (I) , I=LOWER, UPPER)
15  FORMAT (3 (I2, 2X) /4 (2X, F4. 1) )
```

The READ statement is clear enough; it is the FORMAT statement that needs an explanation. The 3 (I2, 2X) takes care of the three integer values in the first input line, and the slash forces FORTRAN to start a new record. The 4 (2X, F4. 1) corresponds, then, to the first four input elements. Now, if the program has to read more elements, the parentheses around the (2X, F4. 1) will force FORTRAN to start a new record and repeat that pattern, taking up to four more elements from the new line. This process will continue until the input list is satisfied.

The same technique can be used for output as long as the programmer remembers to take care of the carriage control requirements. To illustrate, suppose we wanted to print some of TRX's elements from TRX (LOWER) through TRX (UPPER) . This can be done with the statements

```
    PRINT 28, LOWER, UPPER, (TRX (I) , I=LOWER, UPPER)
28  FORMAT (1X, 10X, 2 (I3, 4X) / (1X, 6X, 4 (F4. 1, 2X) )
```




"Define variables."
"Print headings for the first page."
**do** for X = 1 to 4.99 by 0.01:
    **if**
        page is full
    **then**
        "Print headings for next page."
    **else**
    **endif**
    "Compute data for the next line."
    "Print the next line."
**enddo**
"Stop."

**FIGURE 15.3 (a)** Overall Representation of Example 15.1.

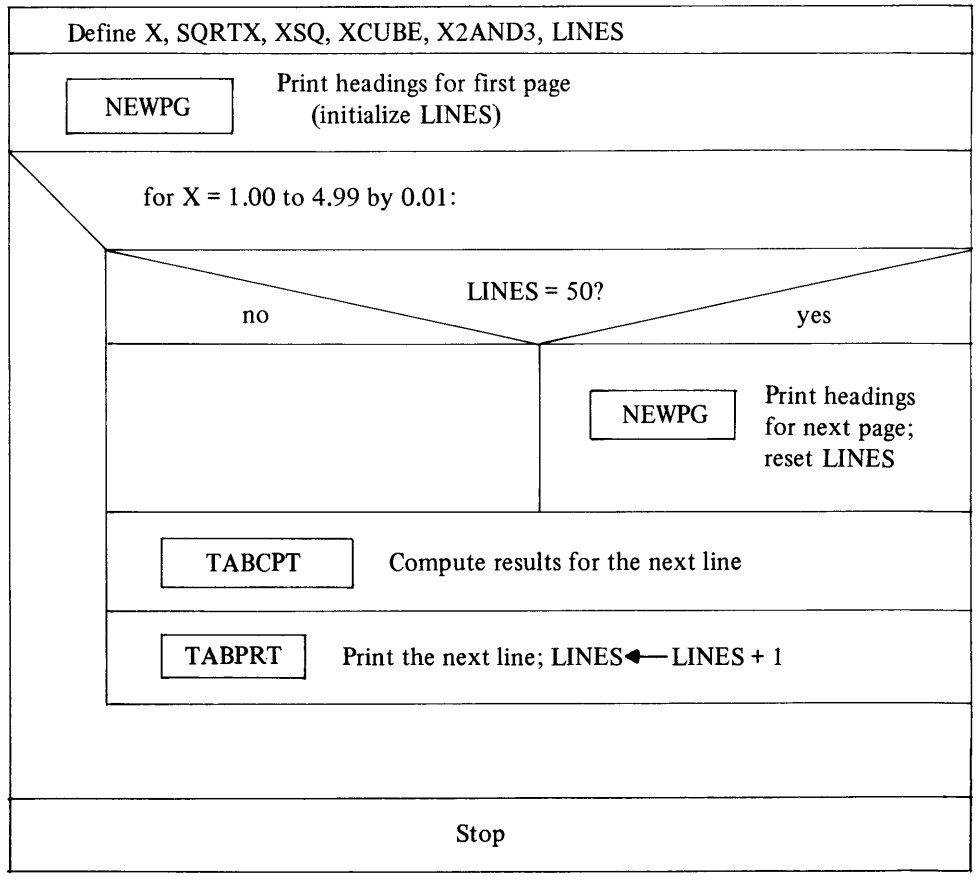



**FIGURE 15.3 (b)** Example 15.1 Representation Showing Breakup into Subprograms.

```
"Define X,SQRTX,XSQ,XCUBE,X2AND3,LINES."
"Use NEWPG to print headings for the first page."
do for X = 1 to 4.99 by 0.01:
    if
        LINES = 50
    then
        "Use NEWPG to print headings for the
                next line."
    else
    endif
    "Use TABCPT to compute results for the next line."
    "Use TABPRT to print the next line of output and
                increment LINES."
enddo
"Stop."
```

**FIGURE 15.3 (b)** *(Continued)*

The following will happen: The two values LOWER and UPPER will be printed on a single line starting in column 11. LOWER will appear in columns 11–13 and UPPER in columns 18–20. Then, FORTRAN will end that record (because of the slash). The new record will start on the next line (because of the 1X), and four elements of TRX will be printed on that new line, starting in column 7. If there are more elements to be printed, FORTRAN will end the record and repeat the pattern. As a result, it will produce however many lines, each containing four elements from TRX starting in column 7. If the output list is completed before the full line of four elements is completed, the program simply prints what it is supposed to print. If that means that the last line has only three elements, fine. Note that if the parentheses were not included around the specifiers after the slash, we would run into the same difficulty described in Section 15.4.2. After printing the first four elements, FORTRAN will use the first character of the next element for carriage control.

## 15.5 EXAMPLES

Now we are ready to apply these formatting features to some more realistic situations. We shall do so by constructing two complete programs in which edit-directed formatting play an important part. This will provide an opportunity to see how we can make these capabilities work for us to produce well-organized input and output data.

**Example 15.1** The production of reliable mathematical tables is a key factor that motivated the development of computers. We shall design a program to prepare such a table as a means of illustrating how proper formatting can be used to great advantage. In this particular case, there is no input requirement. Instead, the program is to generate all of its results based on internal information: All required computations are built around a variable named X. For X of 1.00, 1.01, 1.02, and so on, up to and including 1.49, the program is to print a table in which each line contains X, X squared, the square root of X, X cubed, and the sum of X squared and X cubed. Thus, the actual computing requirements are not terribly severe. Much of our attention will be devoted to the following additional requirements:

1. Except for the X-values themselves, all of the other values are to be computed and displayed rounded to the fifth decimal place.
2. Each page of output is to contain exactly fifty lines of the table.
3. The columns of the table are to be nicely spaced so that they are easily read. Column spacing is to be set up with the assumption that each line has 80 columns available for printout. (Even though the page may be 132 columns wide, we can pretend that the rightmost 52 columns will not be there.)
4. Each column is to be topped by a suitable column heading, and these headings are to appear on every page.

5. The following two labels are to appear at the top of each page:
   (1) USEFUL NUMBERS: -TABLE 808
   (2) X VS. SQRT(X), X**2, X**3, AND X**2+X**3
   Each of these labels is to be centered on an 80-column line.

We shall prepare this program by going through a systematic process in which the necessary programs are identified, developed, and put together to produce the final result.

To begin with, it is not difficult to recognize that the overall process can be treated as a big loop in which each cycle produces a page of output. Each page, in turn, consists of two distinct parts: the headings (the two labels and the column headings), which appear once and then are not needed till the next page, and the computed lines of output. Now each individual line in the table can be considered as a cycle in the loop. Consequently, without having defined any of the details, we already have an overall picture of the program's structure. As indicated by the pseudocode and flow diagram in Figure 15.3(a), the process starts with an initialization activity to set up the system. In this case, initialization consists simply of printing the headlines and labels on the first page, so that everything is ready for the actual computations. The cyclic process, then, is a loop in which each cycle takes care of a line. Inside that loop is an IF-THEN-ELSE component that checks during each cycle to determine whether enough cycles have been performed to fill a page of output.

Now, based on this general organization, we can look at each component in a little more detail. We mentioned initialization before, and we saw that there was nothing mysterious there. However, since we need to prepare a new page (the first one) there and we need to do the same thing for the other pages as part of the cyclic activity, it is clear that we can help keep things simple by constructing the preparation of each new page as a subprogram, so that the main program can treat it as a single "operation." (This subprogram will be named NEWPG.)

Once a page has its labels and column headings, it must receive 50 lines of computed output. Accordingly, we can regulate this counting process simply by counting the lines as they are produced. A line counter (an integer variable named LINES) will be declared and used for this purpose. When a new page is prepared, LINES will be initialized to zero as part of that process. Then, every time a table entry is prepared and printed, LINES will be increased by 1 as part of *that* process.

The activities relating to the production of each entry in the table do not require much discussion. For the sake of illustration, we shall set up a separate subprogram (TBLCPT) to compute the results and another one (TBLPRT) to print them. Since each computed result has to be rounded to five decimal places, we shall use a separate subprogram (ROUND) that can be invoked for each value to be rounded. Now, we have enough additional detail to define the program a little more specifically. This is shown in the diagram and pseudocode of Figure 15.3(b). Note (by comparison with Figure 15.3(a)) that the overall structure is unchanged; the only difference is that there are more details.

Having attended to the organizational matters, we can turn our attention to the appearance of the table itself. First, we shall look at the numerical values themselves.

1. Since X is required to range from 1.00 to 4.99, we know that it will always be positive, and it will not require more than four columns for display. (The specification, in fact, can be defined as F4.2.)

2. The largest values in the table will be those in the last column (i.e., X**2+X**3). For X=5, this value comes to 150, thereby telling us that we need three integer digits. The requirements call for five decimal places, and the values always will be positive. Consequently, we need 5+3+1 or nine positions (including space for the decimal point) to display these values, the format specification being F9.5. To keep things consistent, we shall use the same format for the other three computed values (SQRT(X), X**2 and X**3) even though there will be surplus columns. Of course, this will not affect the results, since FORTRAN will justify to the right and fill in the unused high order positions with blanks. Thus, each type of numerical result is defined and formatted.

3. Now, adding all of this together, we see that the numerical values by themselves account for 4+9+9+9+9 or 40 positions on each output line. We can display these nicely across an 80-column width by leaving 10 positions at the left and leaving 6 positions between each numerical value. That accounts for 10 + 4*6 or 34 more spaces. Thus, we have used 34 + 40 or 74 spaces, leaving another 6 spaces on the right. Of course, this is not the only possible "nice" format, nor is it necessarily the "nicest" (whatever that is). However, it is reasonable and it illustrates the process of preparing such a format.

By combining all of this formatting information that we have developed, we can write the format statement describing a line of the table. (We shall not forget the blank for carriage control):

```
FORMAT (1X, 10X, F4.2, 6X, F9.5, 6X, F9.5, 6X, F9.5, 6X, F9.5)
```

which shortens to

```
FORMAT (1X, 10X, F4.2, 4(6X, F9.5))
```

Each column needs a column heading above it, and we can define such headings now that we have the columns designed and positioned on the page. Since the typical printing device cannot display superscripts, we shall do the next best thing and use FORTRAN-like notation for the headings:

```
X     SQRT(X)     X**2     X**3     X**2+X**3
```

It is a rather simple job to position these over their respective columns since we know exactly where the columns go. The format description works out to be the following:

```
    FORMAT(1X, 12X, 'X', 8X, 'SQRT(X)', 9X, 'X**2', 11X,
$           'X**3', 9X, 'X**2+X**3')
```

That leaves the two labels at the top of the page. The first one (USEFUL NUMBERS: —TABLE 808) is 26 characters long. If we enter that on an 80-position line, we have 80 − 26 or 54 spaces left over, half of which is 27. Thus, our format description for this label is

```
FORMAT (1X, 27X, 'USEFUL NUMBERS: —TABLE 808')
```

The second label (X VS. SQRT(X), X**2, X**3, and X**2+X**3) occupies 41 positions. The left margin, then, is (80 − 41)/2 or 21 (more or less) so that the format description is

```
FORMAT (1X, 21X, 'X VS. SQRT(X), X**2, X**3, X**2+X**3')
```

Now, if we prefer, we can combine all of the labeling into a single glorious format description, skipping a line between the labels and two lines between the labels and column headings. We shall attach a statement number of 38 to it just to give it a number:

```
38 FORMAT  (1X, 27X, 'USEFUL NUMBERS: —TABLE 808'//1X,
  1          41X, 'X VS. SQRT(X), X**2, X**3, X**2+X**3'///
  2          1X, 12X, 'X', 8X, 'SQRT(X)', 9X, 'X**2', 11X, 'X**3',
  3          9X, 'X**2+X**3')
```

Since all the labels and headings are defined as part of the FORMAT statement, the associated output statement is simple:

```
PRINT 38
```

In one sense, this combined format statement is convenient because it places all of the labeling in one concentrated dose. However, if the programmer finds it easier to deal with separate print statements and format statements for each line of output, then that is the preferred approach.

Having completed the formatting details, it is not terribly far-fetched to say that the rest of the program almost writes itself. As we gain more and more experience, you will continue to find that the time spent in looking at the problem, breaking it into manageable components, and preparing the way by means of informative pseudocode or flow diagrams is time well spent. The actual writing of the FORTRAN statements generally will turn out to be the least worrisome part of the entire process. (Don't shake your head; it really will turn out that way.)

The main program, along with the supporting subprograms, is shown in Figure 15.4, and a fragment of the output is shown in Figure 15.5.

**Example 15.2** When somebody buys a house, the usual arrangement is for the purchaser to borrow most of the money. The loan is repaid over an agreed period of time during which the interest is a fixed percent of the amount that is still owed. Thus, when a borrower agrees, say, to a ten percent loan, and a payment is to be made every month, twelve months a year, then the interest charged during a given month is 10%/12 or 0.833 percent of the amount owed at that time.

```
C*************************************************************
C              EXAMPLE 15.1 - THE MAIN PROGRAM                *
C*************************************************************
C  THIS PROGRAM COMPUTES SQRT(X), X**2, X**3 AND X**2+X**3    *
C  FOR VALUES OF X RANGING FROM 1.00 TO 4.99 IN INCREMENTS    *
C  OF 0.01. THE RESULTS ARE ROUNDED AND DISPLAYED TO FIVE     *
C  DECIMAL PLACES (I.E., TO THE NEAREST 1.0E-5).              *
C  THE FOLLOWING SUBROUTINES ARE USED:                        *
C     NEWPG:--PRINTS PAGE AND COLUMN HEADINGS ON A NEW PG     *
C     TABCPT:--COMPUTES THE TABLE VALUES FOR A GIVEN X        *
C     TABPRT:--FORMATS AND PRINTS A LINE OF THE TABLE         *
C     ROUND:--A GENERAL PROCEDURE THAT ROUNDS A GIVEN         *
C             VALUE TO A SPECIFIED NUMBER OF PLACES           *
C*************************************************************
C  LINES IS A VARIABLE USED TO COUNT THE LINES OF COMPUTED    *
C    VALUES ON A GIVEN PAGE. THIS REGULATES USE OF NEWPG      *
C  THE NAMES X, SQRTX, XSQ, XCUBE, X2AND3 (IT IS HOPED) ARE   *
C    SELF-EXPLANATORY                                         *
C*************************************************************
      PROGRAM      EX1501
      IMPLICIT     NONE
      REAL         X,SQRTX,XSQ,XCUBE,X2AND3
      INTEGER*2    LINES

      CALL NEWPG(LINES)

      DO X=1.00,4.99,0.01
        IF (LINES .EQ. 50) CALL NEWPG (LINES)
        CALL TABCPT (X,SQRTX,XSQ,XCUBE,X2AND3)
        CALL TABPRT (X,SQRTX,XSQ,XCUBE,X2AND3,LINES)
      END DO

      STOP
      END
```

(a)

```
C*************************************************************
C                            NEWPG                            *
C*************************************************************
C  THIS SUBROUTINE STARTS A NEW PAGE OF THE TABLE AND PRINTS  *
C  TWO LABELS AND A SET OF COLUMN HEADINGS ON THAT PAGE. IT   *
C  ALSO INITIALIZES THE TABLE PRODUCTION PROCESS FOR THAT     *
C  PAGE BY SETTING (RESETTING) THE LINE COUNTER (LINES) TO    *
C  ZERO.                                                      *
C*************************************************************
      SUBROUTINE   NEWPG (LINCNT)
      IMPLICIT     NONE
      INTEGER*2    LINCNT
      LINCNT = 0
      PRINT 18
   18 FORMAT ('1')
      PRINT 38
   38 FORMAT (1X,27X,'USEFUL NUMBERS:--TABLE 808'//
     1        1X,21X,'X VS. SQRT(X), X**2, X**3, X**2+X**3'
     2        ///1X,12X,'X',8X,'SQRT(X)',9X,'X**2',11X,'X**3',
     3        9X,'X**2+X**3'/)
      RETURN
      END
```

(b)

```
C*************************************************************
C                          TABCPT                             *
C*************************************************************
C  THIS SUBROUTINE PREPARES THE COMPUTED VALUES FOR A        *
C  LINE OF PRINTOUT USING X AS A SINGLE INPUT VALUE          *
C  AFTER COMPUTING EACH VALUE, TABCPT INVOKES ROUND,         *
C  ANOTHER SUBPROGRAM THAT ROUNDS THAT VALUE                 *
C*************************************************************
      SUBROUTINE     TABCPT (V,SQRTV,VSQ,VCUBE,V2AND3)
      IMPLICIT       NONE
      REAL           V,SQRTV,VSQ,VCUBE,V2AND3
      SQRTV = ROUND(SQRT(V),5)
      VSQ = V*V
      VCUBE = V*VSQ
      V2AND3 = ROUND(VSQ+VCUBE,5)
      VSQ = ROUND(VSQ,5)
      VCUBE = ROUND(VCUBE,5)

      RETURN
      END
```

**(c)**

```
C*************************************************************
C                            ROUND                            *
C*************************************************************
C  THIS FUNCTION REQUIRES TWO ARGUMENTS: THE NUMBER TO BE     *
C  ROUNDED, AND THE NUMBER OF PLACES TO WHICH IT IS TO BE     *
C  ROUNDED. (EVEN THOUGH IT ALWAYS IS USED HERE WITH THE      *
C  VALUE FOR THE SECOND ARGUMENT, IT WAS JUST AS EASY TO      *
C  DESIGN IT FOR GENERAL USE, AND THAT USUALLY IS GOOD.       *
C*************************************************************
      FUNCTION ROUND (ARG,NPLACES)
      IMPLICIT  NONE
      REAL      ARG,TEMP
      INTEGER NPLCES
      TEMP = 10.0**NPLCES
      ROUND = NINT(TEMP*ARG)/TEMP
      RETURN
      END
```

**(d)**

```
C*************************************************************
C                          TABPRT                             *
C*************************************************************
C  THIS SUBROUTINE PRINTS A LINE OF TABULAR OUTPUT PRODUCED   *
C  BY TABCPT.                                                 *
C*************************************************************
      SUBROUTINE      TABPRT (Y,SQRTY,YSQ,YCUBE,Y2AND3,LIN)
      IMPLICIT        NONE
      REAL            Y,SQRTY,YSQ,YCUBE,Y2AND3
      INTEGER*2       LIN
      PRINT (6,31) Y,SQRTY,YSQ,YCUBE,Y2AND3
   31 FORMAT (1X,10X,F4.2,4(6X,F9.5))
      LIN = LIN+1
      RETURN
      END
```

**(e)**

**FIGURE 15.4** **(a)** Main Program for Example 15.1. **(b)** NEWPG Subroutine for Example 15.1. **(c)** TABCPT Subroutine for Example 15.1. **(d)** ROUND Function for Example 15.1. **(e)** TABPRT Subroutine for Example 15.1.

USEFUL NUMBERS:--TABLE 808

X VS. SQRT(X), X**2, X**3, X**2+X**3

| X | SQRT(X) | X**2 | X**3 | X**2+X**3 |
|---|---|---|---|---|
| 3.50 | 1.87078 | 12.24873 | 42.86833 | 55.11703 |
| 3.51 | 1.87345 | 12.31882 | 43.23682 | 55.55563 |
| 3.52 | 1.87612 | 12.38912 | 43.60741 | 55.99649 |
| 3.53 | 1.87878 | 12.45961 | 43.98012 | 56.43971 |
| 3.54 | 1.88144 | 12.53030 | 44.35493 | 56.88521 |
| 3.55 | 1.88410 | 12.60119 | 44.73187 | 57.33304 |
| 3.56 | 1.88675 | 12.67228 | 45.11095 | 57.78322 |
| 3.57 | 1.88939 | 12.74357 | 45.49216 | 58.23572 |
| 3.58 | 1.89204 | 12.81506 | 45.87552 | 58.69057 |
| 3.59 | 1.89468 | 12.88676 | 46.26102 | 59.14777 |
| 3.60 | 1.89732 | 12.95865 | 46.64867 | 59.60730 |
| 3.61 | 1.89995 | 13.03074 | 47.03848 | 60.06920 |
| 3.62 | 1.90258 | 13.10303 | 47.43045 | 60.53346 |
| 3.63 | 1.90521 | 13.17552 | 47.82460 | 61.00011 |
| 3.64 | 1.90783 | 13.24821 | 48.22093 | 61.46912 |
| 3.65 | 1.91045 | 13.32110 | 48.61945 | 61.94054 |
| 3.66 | 1.91306 | 13.39419 | 49.02014 | 62.41432 |
| 3.67 | 1.91567 | 13.46748 | 49.42303 | 62.89050 |
| 3.68 | 1.91828 | 13.54097 | 49.82813 | 63.36909 |
| 3.69 | 1.92089 | 13.61466 | 50.23543 | 63.85008 |
| 3.70 | 1.92349 | 13.68855 | 50.64496 | 64.33350 |
| 3.71 | 1.92608 | 13.76264 | 51.05669 | 64.81931 |
| 3.72 | 1.92868 | 13.83694 | 51.47066 | 65.30759 |
| 3.73 | 1.93127 | 13.91143 | 51.88686 | 65.79828 |
| 3.74 | 1.93386 | 13.98612 | 52.30528 | 66.29140 |
| 3.75 | 1.93644 | 14.06101 | 52.72595 | 66.78694 |
| 3.76 | 1.93902 | 14.13610 | 53.14888 | 67.28496 |
| 3.77 | 1.94160 | 14.21139 | 53.57407 | 67.78545 |
| 3.78 | 1.94417 | 14.28688 | 54.00151 | 68.28838 |
| 3.79 | 1.94674 | 14.36257 | 54.43121 | 68.79378 |
| 3.80 | 1.94931 | 14.43846 | 54.86320 | 69.30165 |
| 3.81 | 1.95187 | 14.51455 | 55.29745 | 69.81200 |
| 3.82 | 1.95443 | 14.59084 | 55.73401 | 70.32483 |
| 3.83 | 1.95699 | 14.66733 | 56.17285 | 70.84018 |
| 3.84 | 1.95954 | 14.74402 | 56.61398 | 71.35800 |
| 3.85 | 1.96209 | 14.82091 | 57.05743 | 71.87834 |
| 3.86 | 1.96463 | 14.89800 | 57.50317 | 72.40117 |
| 3.87 | 1.96718 | 14.97529 | 57.95125 | 72.92653 |
| 3.88 | 1.96972 | 15.05278 | 58.40164 | 73.45442 |
| 3.89 | 1.97225 | 15.13047 | 58.85435 | 73.98482 |
| 3.90 | 1.97479 | 15.20836 | 59.30940 | 74.51776 |
| 3.91 | 1.97732 | 15.28645 | 59.76678 | 75.05322 |
| 3.92 | 1.97985 | 15.36474 | 60.22652 | 75.59125 |
| 3.93 | 1.98237 | 15.44323 | 60.68861 | 76.13184 |
| 3.94 | 1.98489 | 15.52192 | 61.15305 | 76.67496 |
| 3.95 | 1.98741 | 15.60081 | 61.61986 | 77.22066 |
| 3.96 | 1.98992 | 15.67990 | 62.08904 | 77.76892 |
| 3.97 | 1.99243 | 15.75919 | 62.56059 | 78.31978 |
| 3.98 | 1.99494 | 15.83868 | 63.03452 | 78.87320 |
| 3.99 | 1.99744 | 15.91837 | 63.51083 | 79.42920 |

**FIGURE 15.5** A Table of Useful Numbers Produced by a Computer.

to arrange for such payments is to agree that the borrower will pay back a certain fixed amount of the loan each month. To that amount will be added the interest due at that point, and the payment for that month will be the sum of those two items. Suppose, for example, that somebody borrows $12000 to be repaid in 24 monthly installments at 12% interest. The agreement is that each month the borrower will pay back $500 of what he owes, i.e., $500 of the principal. Since it is a 12% loan, the interest due in any month is 12%/12 or 1 percent of the unpaid balance. Thus, at the time of the first payment, the interest is 1% of $12000, or $120. That makes the first payment $500 + $120 or $620. The next month, the unpaid balance is $11500

($12000 − the $500 repaid during the first month), so that the interest is 1% of $11500 or $115. Thus, the second payment is $500 + $115, or $615. The rest of the payments are computed in the same way, so that when the 24th and final payment is due, it will be the last $500 plus the interest due on that $500 (i.e., $5), or $505.

For various reasons, most people (lenders as well as borrowers) do not like this type of repayment plan. One obvious reason is that each payment is different, making it easier for everyone to become confused. Thus, instead of setting the principal payment to be a fixed amount each month, the standard practice is to set the *total* payment at a fixed amount each month. Then, both the principal and interest amounts change with each payment, but the two figures always add up to the same total.

What usually happens is that we know the size of the loan (LOAN), the interest rate in terms of percent per year (RATE), and the number of monthly payments (NUMPAY). From RATE we can determine the monthly interest rate as a fraction (MTHINT):

$$\text{MTHINT} = \frac{0.01\text{RATE}}{12}$$

Then, the size of each (equal) monthly payment (PAY) can be computed as follows:

$$\text{PAY} = \text{MTHINT}\,(\text{LOAN}) \left[ \frac{(1+\text{MTHINT})^{\text{NUMPAY}}}{(1+\text{MTHINT})^{\text{NUMPAY}} - 1} \right]$$

As described before, each payment breaks up into two pieces: an amount (PRINC) that goes to repay part of the actual loan, and an amount (INTRST) that covers the interest due. At any given time, the borrower still owes BALANC, so that the interest due on that balance is

```
INTRST = MTHINT * BALANC
```

Having taken care of the interest, we calculate whatever is left of the monthly payment and use it toward repaying he principal:

```
PRINC = PAY - INTRST
```

As a result of this payment, the balance is reduced by PRINC, and the interest for the next payment can be based on that new balance. The entire process starts, of course, with the fact that at the time of the first payment

```
BALANC = LOAN
```

Having said all of that, we can turn our attention to the problem itself. The Cinderblock Building and Loan Company wants to give each of its borrowers a printout showing the payment schedule. (Some of Cinderblock's competitors *sell* such schedules to their clients.) For each payment, the printout is to give the payment number; the balance still owed (not counting that payment); the amount of the payment credited toward that balance; the amount of the payment going for interest; and the new balance resulting from that payment. As a matter of convenience, the printout is to show two payments on each line and 96 payments on each page (unless there are fewer than 96 to be shown). Each page is to include headlines and column labels in keeping with Cinderblock's tradition of dignity and service. The printout is to be based on a page that is 120 columns wide.

Crandall F. (Chip) Block (whose grandfather, Selwyn F. Block, started the company with B. Harlan Cinder) wanted to take a personal hand in designing the printout, and he came up with the layout shown in Figure 15.6. Of course, he left the details to the technical expert (you), but he did supply the following guidelines:

1. Cinderblock will never agree to a loan with more than 500 payments.
2. The size of a loan will not exceed $999999.00, and it always will be in whole dollars.
3. Any loan accepted by Cinderblock must work out to a monthly payment no greater than $9999.99. (Don't ask me why; ask Crandall F. Block.)

Additional information is available with regard to input. Data for each loan is prepared on a single line with the following format:

| *column(s)* | *description* |
|---|---|
| 1–6 | Loan number (LOANID), a six-digit integer |
| 11–30 | Borrower's last name (LASTNM), left justified |

CINDERBLOCK BUILDING AND LOAN CO.
PERSONALIZED MORTGAGE PAYMENT SCHEDULE
PREPARED ESPECIALLY FOR

JACQUES AND LOUELLA PARSNIPS

LOAN NO. 3574 AMOUNT OF LOAN: 30000.00 RATE: 10.50 NO. OF PAYMENTS: 102 PAYMENT: 445.84

| PYMT NO. | AMT DUE | PRINC AMT | INTRST AMT | NEW BAL | PYMT NO. | AMT DUE | PRINC AMT | INTRST AMT | NEW BAL |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 30000.00 | 183.34 | 262.50 | 29816.66 | 2 | 29816.66 | 184.94 | 260.90 | 29631.71 |
| 3 | 29631.71 | 186.56 | 259.28 | 29445.15 | 4 | 29445.15 | 188.20 | 257.64 | 29256.95 |
| 5 | 29256.95 | 189.84 | 256.00 | 29067.11 | 6 | 29067.11 | 191.50 | 254.34 | 28875.61 |
| 7 | 28875.61 | 193.18 | 252.66 | 28682.43 | 8 | 28682.43 | 194.87 | 250.97 | 28487.56 |
| 9 | 28487.56 | 196.57 | 249.27 | 28290.99 | 10 | 28290.99 | 198.29 | 247.55 | 28092.70 |
| 11 | 28092.70 | 200.03 | 245.81 | 27892.66 | 12 | 27892.66 | 201.78 | 244.06 | 27690.88 |
| 13 | 27690.88 | 203.54 | 242.30 | 27487.34 | 14 | 27487.34 | 205.33 | 240.51 | 27282.01 |
| 15 | 27282.01 | 207.12 | 238.72 | 27074.89 | 16 | 27074.89 | 208.93 | 236.91 | 26865.96 |
| 17 | 26865.96 | 210.76 | 235.08 | 26655.20 | 18 | 26655.20 | 212.61 | 233.23 | 26442.58 |
| 19 | 26442.58 | 214.47 | 231.37 | 26228.11 | 20 | 26228.11 | 216.34 | 229.50 | 26011.77 |
| 21 | 26011.77 | 218.24 | 227.60 | 25793.53 | 22 | 25793.53 | 220.15 | 225.69 | 25573.38 |
| 23 | 25573.38 | 222.07 | 223.77 | 25351.30 | 24 | 25351.30 | 224.02 | 221.82 | 25127.28 |
| 25 | 25127.28 | 225.98 | 219.86 | 24901.30 | 26 | 24901.30 | 227.95 | 217.89 | 24673.35 |
| 27 | 24673.35 | 229.95 | 215.89 | 24443.39 | 28 | 24443.39 | 231.96 | 213.88 | 24211.43 |
| 29 | 24211.43 | 233.99 | 211.85 | 23977.44 | 30 | 23977.44 | 236.04 | 209.80 | 23741.40 |
| 31 | 23741.40 | 238.10 | 207.74 | 23503.30 | 32 | 23503.30 | 240.19 | 205.65 | 23263.11 |
| 33 | 23263.11 | 242.29 | 203.55 | 23020.81 | 34 | 23020.81 | 244.41 | 201.43 | 22776.40 |
| 35 | 22776.40 | 246.55 | 199.29 | 22529.85 | 36 | 22529.85 | 248.70 | 197.14 | 22281.15 |
| 37 | 22281.15 | 250.88 | 194.96 | 22030.27 | 38 | 22030.27 | 253.08 | 192.76 | 21777.18 |
| 39 | 21777.18 | 255.29 | 190.55 | 21521.89 | 40 | 21521.89 | 257.52 | 188.32 | 21264.37 |
| 41 | 21264.37 | 259.78 | 186.06 | 21004.59 | 42 | 21004.59 | 262.05 | 183.79 | 20742.54 |
| 43 | 20742.54 | 264.34 | 181.50 | 20478.20 | 44 | 20478.20 | 266.66 | 179.18 | 20211.54 |
| 45 | 20211.54 | 268.99 | 176.85 | 19942.54 | 46 | 19942.54 | 271.34 | 174.50 | 19671.20 |
| 47 | 19671.20 | 273.72 | 172.12 | 19397.48 | 48 | 19397.48 | 276.11 | 169.73 | 19121.37 |
| 49 | 19121.37 | 278.53 | 167.31 | 18842.84 | 50 | 18842.84 | 280.97 | 164.87 | 18561.86 |
| 51 | 18561.86 | 283.42 | 162.42 | 18278.44 | 52 | 18278.44 | 285.90 | 159.94 | 17992.54 |
| 53 | 17992.54 | 288.41 | 157.43 | 17704.13 | 54 | 17704.13 | 290.93 | 154.91 | 17413.20 |
| 55 | 17413.20 | 293.47 | 152.37 | 17119.73 | 56 | 17119.73 | 296.04 | 149.80 | 16823.68 |
| 57 | 16823.68 | 298.53 | 147.21 | 16525.05 | 58 | 16525.05 | 301.25 | 144.59 | 16223.80 |
| 59 | 16223.80 | 303.88 | 141.96 | 15919.92 | 60 | 15919.92 | 306.54 | 139.30 | 15613.38 |
| 61 | 15613.38 | 309.22 | 136.62 | 15304.15 | 62 | 15304.15 | 311.93 | 133.91 | 14992.22 |
| 63 | 14992.22 | 314.66 | 131.18 | 14677.56 | 64 | 14677.56 | 317.41 | 128.43 | 14360.15 |
| 65 | 14360.15 | 320.19 | 125.65 | 14039.96 | 66 | 14039.96 | 322.99 | 122.85 | 13716.97 |
| 67 | 13716.97 | 325.82 | 120.02 | 13391.15 | 68 | 13391.15 | 328.67 | 117.17 | 13062.48 |
| 69 | 13062.48 | 331.54 | 114.30 | 12730.93 | 70 | 12730.93 | 334.44 | 111.40 | 12396.49 |
| 71 | 12396.49 | 337.37 | 108.47 | 12059.12 | 72 | 12059.12 | 340.32 | 105.52 | 11718.80 |
| 73 | 11718.80 | 343.30 | 102.54 | 11375.50 | 74 | 11375.50 | 346.30 | 99.54 | 11029.20 |
| 75 | 11029.20 | 349.33 | 96.51 | 10679.87 | 76 | 10679.87 | 352.39 | 93.45 | 10327.48 |
| 77 | 10327.48 | 355.47 | 90.37 | 9972.00 | 78 | 9972.00 | 358.59 | 87.25 | 9613.41 |
| 79 | 9613.41 | 361.72 | 84.12 | 9251.69 | 80 | 9251.69 | 364.89 | 80.95 | 8886.80 |
| 81 | 8886.80 | 368.08 | 77.76 | 8518.72 | 82 | 8518.72 | 371.30 | 74.54 | 8147.42 |
| 83 | 8147.42 | 374.55 | 71.29 | 7772.87 | 84 | 7772.87 | 377.83 | 68.01 | 7395.04 |
| 85 | 7395.04 | 381.13 | 64.71 | 7013.90 | 86 | 7013.90 | 384.47 | 61.37 | 6629.43 |
| 87 | 6629.43 | 387.33 | 58.01 | 6241.60 | 88 | 6241.60 | 391.23 | 54.61 | 5850.37 |
| 89 | 5850.37 | 394.55 | 51.19 | 5455.71 | 90 | 5455.71 | 398.10 | 47.74 | 5057.61 |
| 91 | 5057.61 | 401.59 | 44.25 | 4656.02 | 92 | 4656.02 | 405.10 | 40.74 | 4250.92 |
| 93 | 4250.92 | 408.64 | 37.20 | 3842.28 | 94 | 3842.28 | 412.22 | 33.62 | 3430.06 |
| 95 | 3430.06 | 415.83 | 30.01 | 3014.23 | 96 | 3014.23 | 419.47 | 26.37 | 2594.76 |

**FIGURE 15.6** An Individual Account Record.

| | |
|---|---|
| 31–40 | Borrower's first name (FNAME1), left justified |
| 41–50 | Borrower's spouse's/partner's/cohort's/first name (FNAME2), left justified |
| 55–60 | Amount of the loan (LOAN) in dollars |
| 61–64 | Duration of the loan in years (61–62) and months (63–64) (YRS, MOS). Thus, a 16½ year loan would be recorded as 1606 in columns 61–64. |
| 68–70 | Interest rate (RATE), to the nearest tenth of a percent. Thus, a 12% loan would be recorded as 120 and a 6½% loan (dreaming, just dreaming) would be recorded as 065. |

Now we can begin to design the program. Using the same general approach we did in Example 15.1, we see that the overall sequence of events can be described as a simple loop, in which each cycle produces a payment schedule for a new borrower. That schedule, in turn, consists of one or more pages. As in the previous example, each page consists of a set of fixed headings followed by a set of computed values. However, the format is more involved, and the last page of a given payment schedule is not necessarily complete. (It may have fewer than 96 entries.) The overall organization is shown in Figure 15.7(a).

Using this organization as a basis, we can begin to divide the program into separate components and describe what each one is to do.

1. It is clear that the printing of page and column headings can be a separate activity. Accordingly, a subprogram (NEWPG) will attend to that.

2. Whenever the program begins processing for a new client, it must initialize (or reinitialize) the system. Without knowing the details, we can see that this involves reading the client's input, setting the names properly in the page headings (so that NEWPG can print them), computing the monthly payment (which needs to be done only once), and initializing the balance due. While such combinations may or may not require several subprograms in different situations, it still is helpful to think of the entire initialization process as a single "operation." This will involve a CALL to a subroutine named READER to bring in the first line, after which the rest of the initialization will be provided by a subprogram named INIT. Just how simple or complicated INIT needs to be will be hidden from the main program. As far as the main program is concerned, it invokes INIT, and when INIT returns, the initialization is done. The input (via the subroutine READER) is kept separate (i.e., it is not included in INIT so that the ENDFILE processing can be handled clearly and directly by the main program. For purposes of illustration, INIT will attend to its work by invoking two other subprograms:

(1) NEWPG will print the headings for the first page.

(2) ROUND, set up as a function, will round the various figures to the nearest cent.

3. The program must divide each payment into a part that goes for interest and the remainder, which helps pay off the principal. The new balance also must be prepared as part of that process. Accordingly, another subprogram (UPDATE) will perform those duties. Since each printed line shows two payments, UPDATE will be designed to prepare results for two payments.

4. Finally, as expected, a separate subprogram (PAYPRT) will be used to print a line in the payment schedule.

Now we can see that the information that we developed when we followed this line of thought leads us to recognize some additional features that the program can include:

1. As was true in the previous example, we can use a line counter to keep track of the printout as it works its way down the page.

2. As we already have determined, UPDATE will prepare results for two payments. That means that we shall find it necessary to keep the data for both payments until the entire line is printed. We can handle this conveniently by making each of the variables INTRST, PRINC and BALANC a two-element array.

3. Since the output format is "doubled up," the set of column headings appears twice. For illustration, then we shall define and store one set of column headings by means of the PARAMETER declaration and merely print the set twice.

Now, there are enough details to develop the pseudocode or N-S diagram further, as shown in Figure 15.7(b). We can see that the diagram (or pseudocode) is not that far removed from the actual program statements that finally must be produced. This relationship is not fictional; it reemphasizes the fact that,

"Define variables."
"Read the first client's input card."
**while**
    There still are clients to process:
    "Reset the program for a new client."
    **do** for every payment:
        **if**
            the current page is full
        **then**
            "Start a new page."
        **else**
        **endif**
        "Prepare a line of output."
        "Print the next output line."
    **enddo**
    "Read the next client's data."
**endwhile**
"Stop."

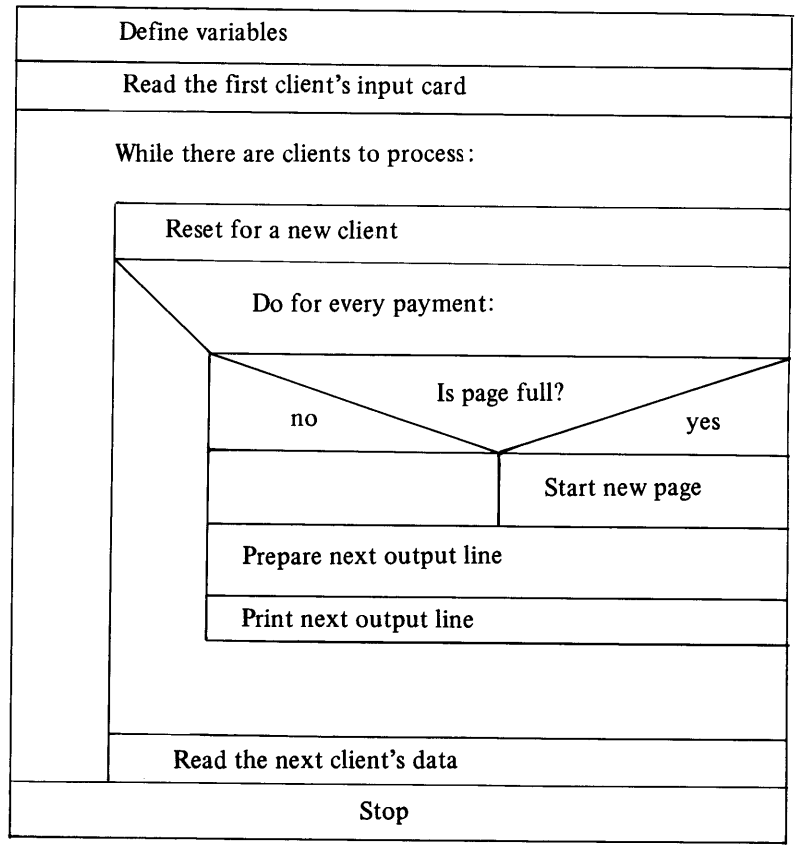



**FIGURE 15.7 (a)** Overall Representation of Example 15.2.

by following this systematic approach to program design, we can work our way to a clear, reliable program in such a manner that the actual writing of the statements is the least troublesome part of the job.

We shall turn our attention to PAYPRT, where much of the format design needs to take place. Using Crandall's information, we can see that the output for each set of payment data requires the following space:

| | |
|---|---|
| payment number: | 3 columns |
| amount still due: | 9 columns (decimal pt. included) |
| principal amount: | 7 columns |
| interest amount: | 7 columns |
| new balance: | 9 columns |

Thus, we have a total of 35 columns, or 70 columns of actual printout for the two sets. Since we are working with a 120-column width, that leaves us with 50 columns to distribute in a nice way. We need four separations between the five columns of data in each payment set, or eight separations for the two sets. If we make each separation four positions wide, that requires 8*4 or 32 columns. A larger separation between the two sets on a line will make the display easier to read, so we shall use eight blank positions for that purpose. This leaves us with ten unused columns, giving us a margin of five positions on either side of the page.

When all of that is put together, the format description takes shape:

```
      FORMAT (1X,5X,I3,4X,F9.2,4X,F7.2,4X,F7.2,4X,F9.2,8X,
     1        I3,4X,F9.2,4X,F7.2,4X,F7.2,4X,F9.2)
```

There is little that can be done to simplify the description:

```
      FORMAT (1X,5X,I3,4X,F9.2,2(4X,F7.2),4X,F9.2,8X,
     1        I3,4X,F9.2,2(4X,F7.2),4X,F9.2)
```

We can play some games with the description, and we shall do so for illustrative purposes. Rewriting the

```
"Define LOANID, LOAN, RATE, MTHINT, INTRST, BALANC, NEWBAL,
        NUMPAY, YR, MO, LASTNM, FNAM1, FNAM2, LINE, SIGNL."
"Initialize SIGNL."
"Use READER to bring in the first client's data."
while
    SIGNL is not zero:
    "Use INIT to reset the program for the new client."
    do for every payment:
        if
            LINE is 48
        then
            "Use NEWPG to start a new page of output."
        else
        endif
        if
            the current payment number is even
        then
            "Use UPDATE to prepare output for a pair of payments."
            "Use PAYPRT to print an output line for two payments."
        else
            if
                this is the final payment
            then
                "Use UPDATE to prepare the final payment's output."
                "Use PAYPRT to print the final line (one payment)."
            else
            endif
        endif
    enddo
    "Use READER to bring in the next client's data."
endwhile
"Print terminating message."
"Stop."
```

**FIGURE 15.7 (b)** Representation of Example 15.2 Showing Breakup into Subprograms.

5X as 1X, 4X, rewriting the 8X as 4X, 4X, and adding another 4X at the end, we convert the description to

```
SP 1
  FORMAT  (1X, 1X, 4X, I3, F9.2, 2(4X, F7.2), 4X, F9.2, 4X,
 1                 4X, I3, F9.2, 2(4X, F7.2), 4X, F9.2, 4X)
```

Now, *this* version can be rewritten as

```
FORMAT  (1X, 1X, 2(4X, I3, F9.2, 2(4X, F7.2), 4X, F9.2, 4X))
```

It is up to you to decide whether this really is a simplification or just a more concise description.

Once the values themselves have been positioned, we can place the column headings over them with little difficulty, thereby giving us a starting point for the layouts needed in NEWPG. To help make this placement easier (and there never is anything wrong in doing that), we shall summarize the data format just developed by showing how the positions of the output line are used:

| *position(s)* | *description* |
|---|---|
| 1–5 | left margin |
| 6–8 | payment number |
| 13–21 | unpaid balance |

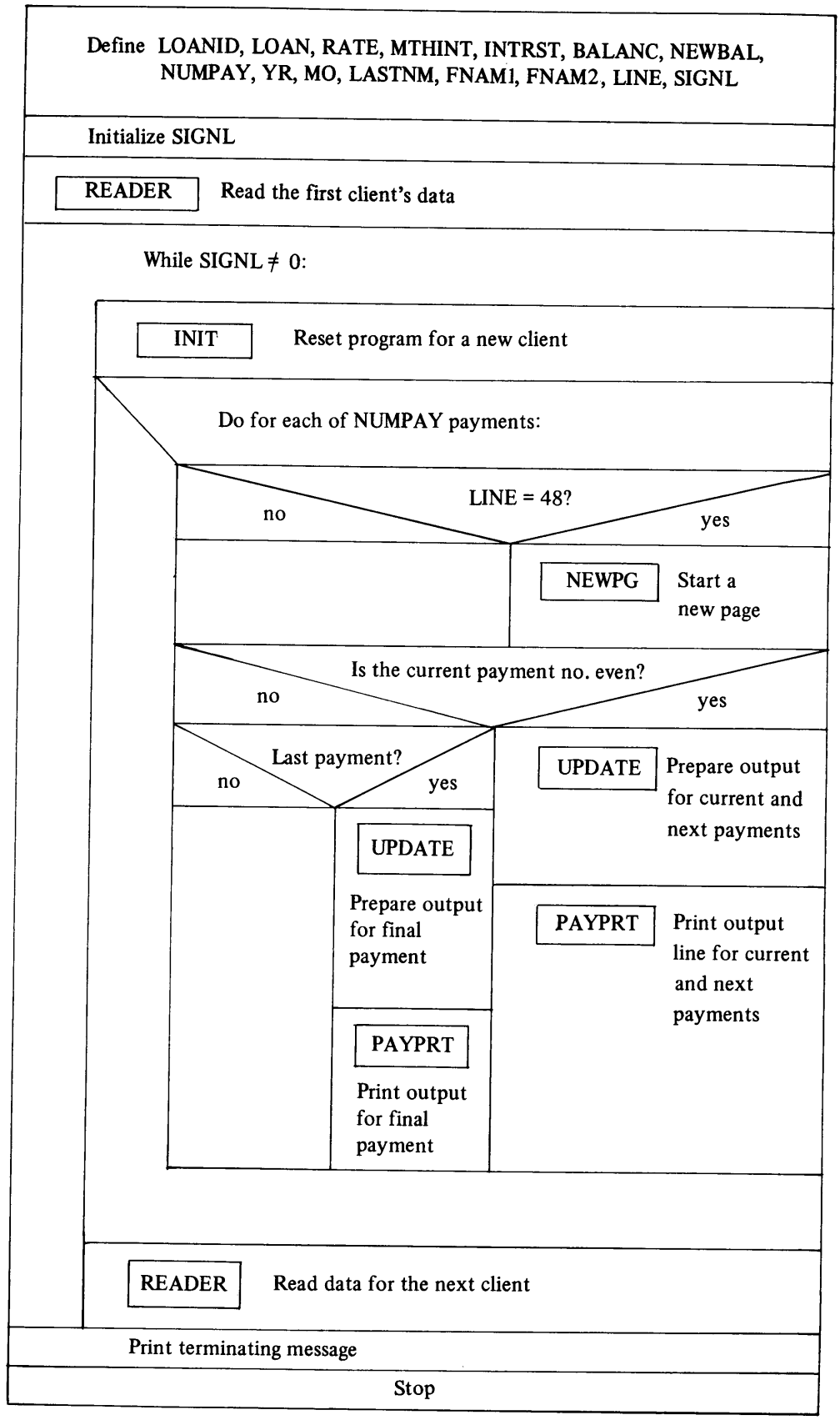



**FIGURE 15.7 (b)** *(Continued)*

| | |
|---|---|
| 26–32 | payment amount credited to principal |
| 37–43 | payment amount credited to interest |
| 48–56 | new balance |
| 65–67 | payment number |
| 72–80 | unpaid balance |
| 85–91 | payment amount credited to principal |
| 96–102 | payment amount credited to interest |
| 107–115 | new balance |
| 116–120 | right margin |

Since we know what the column headings are to look like (Crandall told us), we can determine exactly where these headings should be positioned from the table just shown. See Display 15.1.

**Display 15.1**

*first line of headings:*

| *columns* | *heading* | *columns* | *heading* |
|---|---|---|---|
| 5–8 | PYMT | 64–67 | PYMT |
| 16–18 | BAL | 75–77 | BAL |
| 27–31 | PRINC | 86–90 | PRINC |
| 37–42 | INTRST | 96–101 | INTRST |
| 51–53 | NEW | 110–112 | NEW |

*second line of headings:*

| *columns* | *heading* | *columns* | *heading* |
|---|---|---|---|
| 6–8 | NO. | 65–67 | NO. |
| 16–18 | DUE | 75–77 | DUE |
| 28–30 | AMT | 87–89 | AMT |
| 39–41 | AMT | 98–100 | AMT |
| 51–53 | BAL | 110–112 | BAL |

From this point, it is just a matter of careful counting to make sure the format descriptions match the requirements. Since we decided to store the column headings as character constants (with the PARAMETER declaration), there will be no literal specifications, and our FORMAT statements (one for each line of headings) will look like those shown in Display 15.2.

**Display 15.2**

```
  FORMAT  (1X, 4X, A4, 7X, A3, 8X, A5, 5X, A6, 9X, A3, 10X,      first
 1                 A4, 7X, A3, 8X, A5, 5X, A6, 9X, A3)           line

  FORMAT  (1X, 5X, A3, 7X, A3, 9X, A3, 8X, A3, 9X, A3, 11X,     second
 1                 A3, 7X, A3, 9X, A3, 8X, A3, 9X, A3)           line
```

Note that indentation can be used to make it easier to check the formats.

The other headings to be placed at the top of each page do not involve anything particularly new, if you followed the previous example carefully. Each heading has to be centered (this time using 120 positions), and that process was described earlier. One thing that might be pointed out concerns the heading containing the people's names. Since we know (from the input format) that the last name has 20 characters and each of the two first names has 10 characters, we can add those lengths into the overall length of that heading (remembering to include a blank position on each side of each name). Once that is done, the entire heading can be centered as before.

The rest of the computations are straightforward, so we can proceed with the preparation of the actual statements. In doing so, we become aware of the fact that there may be a situation in which some loan is to be repaid with an odd number of installments. There are many ways to handle this. (Rejection of

```
C*************************************************************
C                EXAMPLE 15.2 - THE MAIN PROGRAM              *
C*************************************************************
C  THIS PROGRAM COMPUTES AND PRINTS LOAN PAYMENT SCHEDULES    *
C  FOR MORTGAGES TO BE REPAID ON A FIXED SIZE PAYMENT BASIS   *
C  WITH THE INTEREST BEING A FIXED PERCENTAGE OF THE UNPAID   *
C  BALANCE.                                                   *
C*************************************************************
C  LOANID:   THE SIX-DIGIT LOAN IDENTIFICATION NUMBER         *
C  LOAN:     THE AMOUNT OF THE LOAN                           *
C  RATE:     THE INTEREST RATE, ANNUAL PERCENT                *
C  MTHINT:   THE INTEREST RATE, MONTHLY FRACTION              *
C  PAY:      THE AMOUNT OF A MONTHLY PAYMENT                  *
C  PRINC:    (1) AND (2); EACH IS AN AMOUNT OF A GIVEN        *
C            PAYMENT CREDITED TOWARD REPAYMENT OF THE LOAN    *
C  INTRST:   (1) AND (2); EACHIS AN AMOUNT OF A GIVEN         *
C            PAYMENT COVERING THE INTEREST DUE THEN           *
C  BALANC:   (1) AND (2); EACH IS THE AMOUNT OF PRINCIPAL     *
C            STILL TO BE REPAID AT THAT POINT                 *
C  NEWBAL:   SAME USAGE AS BALANC                             *
C  NUMPAY:   THE NUMBER OF LOAN PAYMENTS (INSTALLMENTS)       *
C  YR,MO:    THE NUMBER OF YEARS AND MONTHS OVER WHICH        *
C            A PARTICULAR LOAN IS TO BE REPAID                *
C  LASTNM:   BORROWER'S LAST NAME                             *
C  FNAM1,2:  FIRST NAMES                                      *
C  LINE:     A LINE COUNTER THAT REGULATES NEW PAGES          *
C  SIGNL:    A SIGNAL FOR END-OF-FILE PROCESSING              *
C*************************************************************

      PROGRAM          EX1502
      IMPLICIT         NONE
      REAL             LOAN,RATE,MTHINT,PAY,PRINC(2),INTRST(2),
     1                 BALANC(2),NEWBAL
      INTEGER*4        LOANID
      INTEGER*2        YR,MO,NUMPAY,PAYNUM,LINE,SIGNL
      CHARACTER        LASTNM*20,FNAM1*10,FNAM2*10

      SIGNL = 1
      CALL READER (LOANID,LASTNM,FNAM1,FNAM2,LOAN,YR,MO,RATE,SIGNL)

      DO WHILE (SIGNL .EQ. 0)
        CALL INIT (LASTNM,FNAM1,FNAM2,LOANID,LOAN,YR,MO,RATE,MTHINT,
     1           NUMPAY,PAY,NEWBAL,LINE)
        DO PAYNUM = 1, NUMPAY
          IF (LINE .EQ. 48) CALL NEWPG (LASTNM,FNAM1,FNAM2,LOANID,
     1                                  LOAN,RATE,NUMPAY,PAY,LINE)

C       ********************************************************
C       *  HERE WE SPECIFY TWO DIFFERENT COMPUTATION AND PRINTING*
C       *  OPERATIONS DEPENDING ON WHETHER NUMPAY IS ODD OR EVEN *
C       *  SINCE UPDATE AND PAYPRT BOTH ARE SET UP TO WORK ON TWO*
C       *  PAYMENTS, THEY NEED TO BE INVOKED ONLY WHEN PAYNUM,   *
C       *  THE LOOP INDEX, IS AN EVEN NUMBER. IF IT IS AN ODD    *
C       *  NUMBER AND IT HAPPENS TO BE THE LAST CYCLE, THEN WE   *
C       *  KNOW THAT THE LAST OUTPUT LINE WILL HAVE ONLY ONE     *
C       *  PAYMENT ON IT. ACCORDINGLY, THE FIRST ARGUMENTS TO    *
C       *  UPDATE AND PAYPRT ARE USED TO CONTROL THEIR RESPECTIVE*
C       *  OPERATIONS.                                           *
C       ********************************************************

          IF (MOD(PAYNUM,2) .EQ. 0) THEN
            CALL UPDATE (0,PAY,MTHINT,NEWBAL,PRINC,INTRST,BALANC)
            CALL PAYPRT (0,PAYNUM,NEWBAL,PRINC,INTRST,BALANC,LINE)
          ELSE IF (PAYNUM .EQ. NUMPAY) THEN
                 CALL UPDATE (1,PAY,MTHINT,NEWBAL,PRINC,INTRST,BALANC)
                 CALL PAYPRT (1,PAYNUM,NEWBAL,PRINC,INTRST,BALANC,LINE)
               ELSE
          END IF
        END DO
        CALL READER (LOANID,LASTNM,FNAM1,FNAM2,LOAN,YR,MO,RATE,SIGNL)
      END DO

   99 WRITE (6,66)
   66 FORMAT ('1'///1X,10X,'NORMAL TERMINATION. END OF RUN')
      STOP
      END
```

**FIGURE 15.8 (a)** Main Program for Example 15.2.

```
C*************************************************************
C                              READER                         *
C*************************************************************
C  THIS SUBROUTINE READS AN INPUT LINE FOR A LOAN USING THE *
C  FOLLOWING FORMAT:                                         *
C     COLS 1-6:   LOAN I.D.      COLS 55-60:   LOAN AMOUNT   *
C     COLS 11-30: LAST NAME      COLS 61-62:   YEARS TO PAY  *
C     COLS 31-40: FIRST NAME(1)  COLS 63-64:   MNTHS TO PAY  *
C     COLS 41-50: FIRST NAME(2)  COLS 68-70:   INTEREST RATE *
C*************************************************************
      SUBROUTINE         READER (ID,LAST,F1,F2,AMT,Y,M,RT,SGN)
      IMPLICIT           NONE
      REAL               AMT,RT
      INTEGER*4          ID
      INTEGER*2          Y,M,SGN
      CHARACTER          LAST*20,F1*10,F2*10

      SGN = 1
      READ (5,15,END=26) ID,LAST,F1,F2,AMT,Y,M,RT
   15 FORMAT (I6,4X,A20,2A10,4X,F6.0,2I2,3X,F3.1)
      GO TO 77
   26 SGN = 0
   77 RETURN
      END
```

**(b)**

```
C*************************************************************
C                             INIT                            *
C*************************************************************
C  THIS SUBROUTINE SETS (RESETS) THE PROCESS FOR A NEW LOAN.*
C  SPECIFICALLY, IT COMPUTES THE NUMBER OF MONTHLY PAYMENTS,*
C  DETERMINES THE PAYMENT SIZE, INITIALIZES THE AMOUNT OWED,*
C  AND PRINTS THE FIRST SET OF PAGE AND COLUMN HEADINGS.    *
C*************************************************************
      SUBROUTINE INIT (L,F1,F2,ID,AMT,Y,M,INR,MTHRT,
     1                 NP,P,OWED,LNS)
      IMPLICIT           NONE
      REAL               AMT,INR,MTHRT,P,OWED,TX
      INTEGER*4          ID
      INTEGER*2          Y,M,NP,LNS
      CHARACTER          L*20,F1*10,F2*10
C                       ----------NUMBER OF PAYMENTS----------
      NP = 12*Y + M
C                       ----------MONTHLY INTEREST----------
      MTHRT  = 0.833333E-3*INR
C                       ----------PAYMENT AMOUNT----------
      TX = (1.0+MTHRT)**NP
      P = MTHRT * AMT * (TX/(TX-1))
      P = ROUND (P,2)
C                       ----------AMOUNT OWED----------
      OWED = AMT
C                       ----------START PAGE----------
      CALL NEWPG (L,F1,F2,ID,AMT,INR,NP,P,LNS)
      RETURN
      END
```

**(c)**

```
C*************************************************************
C                             ROUND                           *
C*************************************************************
C  THIS IS THE SAME FUNCTION USED IN THE PREVIOUS EXAMPLE.  *
C*************************************************************
      FUNCTION ROUND (ARG,NPL)
      IMPLICIT   NONE
      REAL       ARG,T
      INTEGER*2  NPL
      T = 10.0**NPL
      ROUND = NINT(T*ARG)/T
      RETURN
      END
```

**(d)**

*(Continued)*

**FIGURE 15.8** **(b)** READER Subroutine for Example 15.2. **(c)** INIT Subroutine for Example 15.2. **(d)** ROUND Function for Example 15.2.

```
C*************************************************************
C                          NEWPG                              *
C*************************************************************
C  THIS SUBROUTINE PRINTS PAGE AND COLUMN HEADINGS. EACH     *
C  TIME A NEW PAGE IS NEEDED, NEWPG WILL RESET THE LINE      *
C  COUNTER AS PART OF THE PROCESSING.                        *
C*************************************************************
      SUBROUTINE NEWPG (L,F1,F2,ID,AMT,INTR,NUM,PAYAMT)
      IMPLICIT   NONE
      REAL       AMT,INTR,PAYAMT
      INTEGER*4  ID
      INTEGER*2  NUM
      CHARACTER  L*20,F1*10,F2*10,
     1           CHD11*4,CHD12*3,CHD13*5,CHD14*6,CHD15*3,
     2           CHD21*3,CHD22*3,CHD23*3,CHD24*3,CHD25*3
      PARAMETER (CHD11='PYMT',CHD12='AMT',CHD13='PRINC',
     1           CHD14='INTRST',CHD15='NEW',CHD21='NO.',
     2           CHD22='DUE',CHD23='AMT',CHD24='AMT',CHD25='BAL')

      WRITE (6,11) F1,F2,L
   11 FORMAT ('1'//1X,43X,'CINDERBLOCK BUILDING AND LOAN CO.'/
     1        1X,43X,'PERSONALIZED MORTGAGE PAYMENT SCHEDULE'/
     2        1X,48X,'PREPARED ESPECIALLY FOR'//
     3        1X,37X,A10,' AND ',A10,1X,A20)
      WRITE (6,21) ID,AMT,INTR,NUM,PAYAMT
   21 FORMAT (1X,5X,'LOAN NO. ',I6,5X,'AMOUNT OF LOAN: ',F8.2,
     1        5X,'RATE: ',F5.2,5X,'NO. OF PAYMENTS: ',I3,5X,
     2        'PAYMENT: ',F7.2)
      WRITE (6,31) CHD11,CHD12,CHD13,CHD14,CHD15,CHD11,CHD12,CHD13,
     1             CHD14,CHD15
   31 FORMAT (//1X,4X,A4,7X,A3,8X,A5,5X,A6,9X,A3,10X,
     1           A4,7X,A3,8X,A5,5X,A6,9X,A3)
      WRITE (6,41) CHD21,CHD22,CHD23,CHD24,CHD25,CHD21,CHD22,CHD23,
     1             CHD24,CHD25
   41 FORMAT (1X,5X,A3,7X,A3,9X,A3,8X,A3,10X,A3,11X,
     1           A3,7X,A3,9X,A3,8X,A3,10X,A3)

      RETURN
      END
```

**(e)**

```
C*************************************************************
C                          UPDATE                             *
C*************************************************************
C  THIS SUBROUTINE PREPARES A SET OF ENTRIES FOR A PRINT     *
C  LINE. IN MOST CASES, THAT LINE WILL SHOW DATA FOR TWO     *
C  PAYMENTS. HOWEVER, IF THE TOTAL NO. OF PAYMENTS IS ODD,   *
C  THE LAST LINE WILL HAVE DATA FOR ONLY ONE PAYMENT. THIS   *
C  IS MONITORED BY THE INVOKING PROGRAM, AND UPDATE CHECKS   *
C  THE MODE OF OPERATION BY LOOKING AT THE FIRST ARGUMENT.   *
C*************************************************************
      SUBROUTINE UPDATE (SW,P,MNTH,NEWAMT,PR,INT,BAL)
      IMPLICIT   NONE
      REAL       P,MNTH,NEWAMT,PR(2),INT(2),BAL(2)
      INTEGER*2  SW

      BAL(1) = NEWAMT
      INT(1) = ROUND (MNTH*BAL(1),2)
      PR(1) = P - INT(1)
      BAL(2) = BAL(1) - PR(1)
C                    ----------HALF LINE TEST----------
      IF (SW .EQ. 1) GO TO 77

      INT(2) = ROUND (MNTH*BAL(2),2)
      PR(2) = P - INT(2)
      NEWAMT = BAL(2) - PR(2)
   77 RETURN
      END
```

**(f)**

**FIGURE 15.8** **(e)** NEWPG Subroutine for Example 15.2. **(f)** UPDATE Subroutine for Example 15.2.

```
C*************************************************************
C                          PAYPRT                             *
C*************************************************************
C  THIS SUBROUTINE PRINTS A LINE IN THE PAYMENT TABLE. EACH  *
C  LINE SHOWS DATA FOR TWO PAYMENTS UNLESS THE TOTAL NO. OF  *
C  PAYMENTS IS AN ODD NUMBER, IN WHICH CASE THE LAST LINE    *
C  (AND ONLY THAT LINE) WILL HAVE DATA FOR ONE PAYMENT.      *
C  PAYPRT DETERMINES WHICH OF THESE TWO OPERATING MODES IT   *
C  WILL FOLLOW BY LOOKING AT ITS FIRST ARGUMENT.             *
C*************************************************************
      SUBROUTINE PAYPRT (FLIP,NUM,NEW,PR,IN,BALN,LN)
      IMPLICIT   NONE
      REAL       NEW,PR(2),IN(2),BALN(2)
      INTEGER*2  FLIP,NUM,LN

      IF (FLIP .EQ. 0) THEN
        WRITE (6,48) NUM-1,BALN(1),PR(1),IN(1),BALN(2),
     1                 NUM,BALN(2),PR(2),IN(2),NEW
        LN = LN + 1
      ELSE
        WRITE (6,58) NUM,BALN(1),PR(1),IN(1),BALN(2)
      END IF

   48 FORMAT (1X,5X,I3,F11.2,2(4X,F7.2),4X,F9.2,11X,
     1             I3,F11.2,2(4X,F7.2),4X,F9.2)
   58 FORMAT (1X,5X,I3,4X,F9.2,2(4X,F7.2),4X,F9.2)

      RETURN
      END
```

**FIGURE 15.8 (g)** PAYPRT Subroutine for Example 15.2.

the input is not one of the acceptable ways.) The method selected for this example is relatively simple-minded, but it is clear and effective. The subprograms UPDATE and PAYPRT each can operate in one of two ways. Most of the time, UPDATE will prepare two sets of payment data and PAYPRT will print a line with two sets of payment data. However, if the situation requires it, UPDATE will prepare only one set (and, of course, it will be the final set), and PAYPRT will print a line showing only that single set. (Of course, it will be the last line for that loan schedule.)

The statements, supported by comments, are shown in Figure 15.8.

## 15.6 SUMMARY

Use of edit-directed input or output gives the programmer complete control over the appearance and interpretation of the data brought into the processor or sent from it. To exercise this control, the programmer must describe in exact detail the treatment of each character in the input record (e.g., each column of an input line) or output record (e.g., each space on a line of printout).

These descriptions take the form of *format specifications* constructed by the programmer and used by FORTRAN in conjunction with edit-directed READ or WRITE statements. A typical way of setting up edit-directed input/output is as follows:

READ/WRITE (*unit,fsnum*) *data list*

where *unit* is the number of the unit to or from which the data are being transmitted and *fsnum* is the label attached to another statement in that program or subprogram. That other statement contains the format description, and it has the following form:

*fsnum* FORMAT (*format specification list*)

The *format specification list* consists of a series of individual descriptions, each one of which instructs FORTRAN how to treat a particular data item or what to do at a particular position in a record. When the input or output actually occurs, FORTRAN

matches the format specifications against the items in *data list*, so that each item is processed in accordance with the information found in the corresponding format description.

For instance, assuming the appropriate declarations, the statements given below

```
      READ (5, 17) REALV, INTGR, REALV2, LETRS, INTGR2
17    FORMAT (3X, F6.2, I4, 4X, F6.4, 2X, A10, 5X, I2)
```

instruct FORTRAN to read (from unit 5) a data list consisting of five items. The way the input is to be interpreted is to be found in statement 17. Statement 17 contains the format description, which consists of nine format specifications. Five of these specifications are concerned with the five data items, and the other four guide FORTRAN to the proper positions in the record in order for the input process to be consistent with the way the data happen to be prepared. Specifically, this format will produce the following results:

1. The 3X causes the first three positions in the input record (i.e., the first three columns in the record or terminal input line) to be ignored. Thus, FORTRAN is ready to read starting from the fourth column.
2. The F6.2 tells FORTRAN to treat the next six columns (they happen to be columns 4–9) as a real number with two decimal places. That places FORTRAN at column 10.
3. The I4 causes the next four columns (10–13) to be read as a four-digit integer. Thus, the next input position is column 14.
4. The 4X repositions FORTRAN four columns further in the record. In other words, columns 14–17 are ignored, and FORTRAN is ready to read starting in column 18.
5. The F6.4 brings in columns 18–23 (i.e., the next six columns) and their contents are treated as a real number with four decimal places.
6. The 2X instructs FORTRAN to ignore the next two columns, so that it is ready to read the next item at column 26.
7. The A10 causes the next ten columns to be read as a 10-character string.
8. The 5X tells FORTRAN to skip over the next five columns, so that the next column of interest is column 41.
9. Finally, the I2, which is matched against the fifth (and last) data item (INTGR2) brings in the next two columns (41–42) as a two-digit integer.
10. The rest of the line is ignored because the data list has been satisfied.

FORTRAN provides a variety of format specifications, so that a wide range of data organizations can be described precisely and completely. In addition to the specifications themselves, there are various shorthand forms so that, in many cases, it is possible to express lengthy descriptions concisely.

**PROBLEMS**

1. A program contains the following declarations:

```
REAL      V1, SAV, TAY, BRKT
```

If we prepared a data line that looked like this:

```
64723001478765643210042110232454787452310004232 8E7898965654895210320323002344'50
```

show what the values in the variables will be as a result of each pair of independent statements given below. (Note that some of the statements may be illegal):

(a)
```
      READ (5, 8) V1, SAV
    8 FORMAT (F2.0, F3.0)
```

(b)
```
      READ (5, 9) SAV, V1
    9 FORMAT (F2.0, F3.0)
```

(c)
```
      READ (5,10) V1,TAY,BKRT
   10 FORMAT (F3.1,3X,F4.1,F3.2)
```
(d)
```
      READ 11,V1,SAV,BKRT,TAY
   11 FORMAT (5X,F8.4,3X,F5.1,4X,F5.4)
```
(e)
```
      READ (UNIT=5,FMT=81)V1,SAV,BKRT,TAY
   81 FORMAT (4X,3F6.2,4X,F8.3)
```
(f)
```
      READ (5,31) V1,BKRT,TAY,SAV
   31 FORMAT (2(2X,F5.1,5X,F2.1))
```
(g)
```
      READ (5,51) V1,BKRT
   31 FORMAT (2(2X,F5.1,5X,F2.1))
```
(h)
```
      READ (5,61) TAY,SAV,BKRT
   61 FORMAT (3X,F7.3)
```
(i)
```
      READ (5,71) BKRT,TAY,V1
   71 FORMAT (3X,I3,2(2X,F4.1))
```

2. A program contains the following declarations:

```
REAL            G1,Y
INTEGER         ALT,NUM,HI,LOW
```

Suppose we had a set of data that looked like this:

```
  10365.420144517698.00267543298.85327572310022459865320#1427584302013654175468331

 0098134670.587731021004632632632545578114022257.685775400213541675008990011753l0

437586912004775306214223008563431l5.207040653313038873360233 98.20448731415963027
```

Show the values that will be stored in each variable as a result of the following independent statements or sequences of statements. (Note: some of these may not work.) Indicate which ones will not work and show why:

(a)
```
      READ (5,15) ALT,NUM,HI,G1
   15 FORMAT (I3,2X,2(I3,2X),F1.1)
```
(b)
```
      READ (5,15) NUM,G1,Y,HI,LOW
   15 FORMAT (2(X2,I3,X3,F3.1),I4)
```
(c)
```
      READ (5,16) NUM,ALT,HI,LOW,Y,G1
   16 FORMAT (5X, 2(3X,I2,2X,I3)/3X,F8.3)
```
(d)
```
      READ (5,17) NUM,Y,ALT,HI,G1,LOW
   17 FORMAT (3X,2(I2,3X,F4.1,3X,I2,2X))
```
(e)
```
      READ (5,20) ALT,HI,NUM,LOW,Y,G1
   20 FORMAT (2X,2(I3,2X,I2,3X)/2(3X,F5.2))
```
(f)
```
      READ (5,21) ALT,G1
      READ (5,22) NUM,HI
      READ (5,21) LOW,Y
   21 FORMAT (3X,I4,F5.3)
   22 FORMAT (7X,I4,8X,I3)
```
(g)
```
      READ (5,23) ALT,HI,LOW,ALT,Y
   23 FORMAT (3X,5X/2X,2(2X,3X))
```

3. A program contains the following declarations:

```
REAL            BTS(8),SK(4,3),FAL,T3
INTEGER         AR,CS(5)
CHARACTER       WRD*4,CRE*2,ALT*6
```

These lines are about to be read:

```
  JK832014220788654443012010125789898564G102577J9895433010478787532343325310235469

 4542410012343778787564113265646498753201232345674565 8D578653231G'564565656326566

6546546546546543E10E020201257788654986543213202023565763430234657324324624646<79
```

Show the values that will be stored in each variable as a result of each of the following independent statements or groups of statements. (Note that some of the statements may not work. Indicate the illegal ones and show why they will not work):

(a)
```
      READ (5,8) BTS,SK,AR,ALT
    8 FORMAT (6F2.0,14(2X,F3.1),I2,A6)
```
(b)
```
      READ (5,9) BTS,WRD,AR,CS
    9 FORMAT (8(3X,F2.1)/A,I3,5I2)
```
(c)
```
      READ (5,8) (BTS(AR),CS(AR),AR=2,5),SK
      READ (5,9) WRD,ALT
    8 FORMAT (2(3X,2(4X,F3.1,1X,I2),2X)/2X,
     1          3(1X,2(F2.1,1X,F3.1,1X)))
    9 FORMAT (2(4X,A))
```
(d)
```
      READ (5,8) WRD(1:3),(SK(AR,AR),AR=1,3)
    8 FORMAT (3X,A3/(4X,F4.2))
```

4. The following variables have been declared and given values as shown below:

| *Name* | *Data Type* | *Value(s)* |
|---|---|---|
| POWER | REAL | -14.2 |
| TOTAL | REAL | 546317.0 |
| AVGS(4) | REAL | 8.8,-2.18,613.44,0.8 |
| AMTS(5) | INTEGER | 1,3,70,-22,407 |
| NUM | INTEGER | 4 |
| SPC | INTEGER | -30 |
| NAME | CHARACTER*10 | TWINKLETOE |
| LABELS(6) | CHARACTER*4 | MEAD, MEAT, MOAT, MOOT |

Show the printed page layout resulting from each of the following statements or groups of statements. Use a 'b' to indicate a blank. (Note that some of the statements may not work. Indicate which ones these are (if there are any, and show why they will not work):

(a)
```
      WRITE (6,14) POWER
   14 FORMAT (F5.1)
```

(b)
```
      PRINT 10,TOTAL,AVGS
   10 FORMAT (10X,F8.1,5X,4(F8.2,3X))
```

(c)
```
      WRITE (6,12) (AVGS(NUM),AMTS(NUM),NUM=1,4)
   12 FORMAT (1X,8(5X,F6.2,5X,I4))
```

(d)
```
      WRITE (6,11) LABELS
      WRITE (6,12) AVGS
      WRITE (6,13) AMTS
   13 FORMAT (/5X,5(I6,3X))
   11 FORMAT ('1',1X,10X,6(A4,11X))
   12 FORMAT ('0',1X,10X,4(F12.3,3X)/)
```

(e)
```
      WRITE (6,14)
      WRITE (6,15) POWER,TOTAL
      WRITE (6,16) AMTS
   14 FORMAT ('1',10X,14HLATEST RESULTS)
   15 FORMAT (1X,12X,'NAME=',A9,3X,'POWER=',
     1        F7.2,3X,'TOTAL=',F13.3)
   16 FORMAT (//5(5X,I5))
```

(f)
```
      WRITE (6,18) (AMTS(I),AVGS(I),I=1,4)
   18 FORMAT (//3X,4(3X,I4/5X,F8.4))
```

5. Simplify each of the following FORMAT statements:

(a)
```
FORMAT (1X,I2,2X,I2,5X,F3.1,3X,F3.1,A8)
```

(b)
```
FORMAT (1X,3X,A6,2X,F4.1,4X,I2,5X,A6,2X,F4.1,4X,I2,
1         5X,A6,2X,F4.1,4X,I2,5X,A6)
```

(c)
```
FORMAT (1X,4X,F6.1,F6.1,2X,A5,3X,F6.1,F6.1,F6.1,2X,
1         A5,7X,A2,2X,I2,1X,A2,2X,I2)
```

(d)
```
FORMAT (1X,5X,I2,F4.0,I3,2X,I3,2X,5X,I2,F4.0,I3,2X,
1         I3,2X,5X,I2,F4.0)
```

6. Expand each of the following format specifications:

(a) `FORMAT (2X,2I2,2F3.0)`
(b) `FORMAT (2X,2(I2,2F3.0))`
(c) `FORMAT (3X,3(1X,2(I2,3X,2F3.1,I1)))`
(d) `FORMAT (5X,2(I3,2(I2,2X,I3)),2A6,2(2X,A6))`
(e) `FORMAT (3X,A20,2I4,5(3I2,2F3.2/3(I1,3X)),4X,A4)`

7. Many of the ordinary items that form part of our lives involve data that can be organized easily as edit-directed records. A number of such items are listed below. For each of these, identify the individual pieces of data that go to make up the item and pretend that the data are to be recorded as an edit-directed record. If this record is to be one of many similarly organized records, each data value must be in its assigned set of columns. Accordingly, design an input layout for the data in each item. Write appropriate FORTRAN declaration statements, READ statements, and FORMAT statements consistent with your input designs:

(a) A Social Security Card
(b) A Driver's License (use your driver's license and assume that all records in this collection will be from the same state as yours)
(c) A Student I.D. card (use your card as a model and assume that all records in this collection will be from your school)
(d) A savings or checking account

(e) A medical services card (e.g., Blue Cross or Blue Shield)
(f) An oil company credit card
(g) A department store credit card
(h) The information on the spine of a nonfiction library book
(i) The information in a television listing (like TV Guide) for an individual program
(j) The information describing a particular brand of multiple vitamin pill

8. Write a subroutine named CENTER that places a given character string in the center of another string, as if that second string represented a line to be printed. The length of either string is not established until the subroutine is actually used. The characters on either side of the centered string must be blanks, but they may not be when the subroutine is invoked.

   (a) List each argument required by your subroutine and indicate why each one is needed.
   (b) Indicate what could happen to cause your subroutine to go wrong and show how you would handle each type of situation.

9. Modify the subroutine in Problem 8 so that the string to be centered, when placed in the second string, leaves the characters on either side of it unchanged.

10. The Hi-Tech Restaurant, in keeping with its neo-industrial decor, thought it would be a good idea to print its daily menus on a computer. (Save your comments.) Accordingly, each line on the menu will be recorded on a separate line, starting in column 1. The printed lines, of course, will be of different lengths, but no line will require more than one input record. (Note that Hi-Tech is a exclusive kind of place, so that, on any given day, there are no choices and one price.)

    Since Hi-Tech plans in advance, it would like a program that will process any number of sets of input data (i.e., it should be able to produce several different menus). Each menu is to be printed on a separate page, with each line centered using a width of 100 columns. There is to be a blank line between each line of print. The first input line for each menu gives the day of the week in columns 1–3 (using SUN, MON, TUE, WED, THU, FRI and SAT) and the date (mm/dd/yy) in columns 11–18. However, Hi-Tech wants the first line to show the day of the week written out in full, along with the month. For example, if an initial input line says

    ```
    THUbbbbbbb08/14/80
    ```

    the corresponding output line (centered, of course) should say

    ```
    HI-TECH MENU FOR THURSDAY, AUGUST 14, 1980
    ```

    The second line always contains a price in columns 1–5, recorded as xxx.xx, and it is to appear on the second printout line following the word PRICE. Write the program assuming that no menu will require more than one page. However, the number of lines on a menu may vary from day to day.

11. Modify the subroutine NEWPG in Example 15.1 so that it numbers the output pages. Use one of the following three choices:

    (a) Starting with PAGE 1, print the page number in the top right hand corner of each page
    (b) Print the page number in the center of the page's first line, putting a dash on either side of the number. Thus, the first page would be numbered -1-, etc.
    (c) Give the user the option of specifying either type of numbering.

12. Prepare a table of natural logarithms for integers 0 through 99. Each entry is to show X and its natural logarithm (LN(X)) and two such entries are to be printed per line. (Of course, there will be no value shown for the natural log of zero, but there will be a space for it so that the table comes out even.) Round the logarithms to 5 places.

13. Prepare the same table as specified in Problem 12 except that each line of print is to show four consecutive values.

14. Generalize the program in Problem 12 or 13 so that the user can specify two integers XMIN and XMAX for which he or she wishes such a table to be prepared. (Note that your program must safeguard against unreasonable input specifications, and you must determine what that means.)

15. Generalize the program in Problem 14 (even further) by designing it so that the user can specify the number of entries that should appear on each line. Note that it is not necessary for the table to come out exactly even. It is perfectly all right for the last line to have fewer entries than the others. Design your program so that it will accept a maximum value of six for the number of entries on each line. You may

handle this in one of the following ways:

(a) The user must specify the number of entries per line
(b) The user has the option of specifying the number of entries per line; if he or she does not, the program uses a default value of *2*.

16. Prepare and print a table of values for the sine, cosine, tangent, cotangent, secant, and cosecant for values of X in degrees from 0 through 45 degrees in increments of 10 minutes. Number the output pages using one of the two ways described in Problem 11. (It is up to you to determine how many entries will be printed on each page.)

17. For values of X from 1 through 10 in increments of 0.1, prepare and print a table in which each entry shows X, e**X, and e**–X. Print two entries on each line, and round the values to 5 decimal places.

18. Modify Example 15.1 so that the program prints two sets of entries on each line.

19. Nariz Chemical Company keeps basic information about its compounds on records formatted as follows:

| *columns* | *item* | *format* |
|---|---|---|
| 1–6 | stock no. | 6-digit integer |
| 11–40 | empirical formula | characters |
| 41–44 | melting point, deg. C | xxx(.)x |
| 51–54 | boiling point, deg. C | xxx(.)x |
| 61–66 | price, dollars/gram | xxx(.)xx |

The empirical formula is a character string consisting of a series of abbreviated element names, each followed by an integer value indicating the number of atoms of that element in the compound. For instance, the string

C6H1206

is an empirical formula for a compound containing 6 atoms of carbon (C), 12 atoms of hydrogen (H), and 6 atoms of oxygen (O). When only one atom of a particular element appears in a compound, there is no number after the abbreviated name; the 1 is implied. Thus, the compounds

C2NH8 and C3H8O

really stand for

$C_2N_1H_8$ and $C_3H_8O_1$

Nariz would like to have a program that prints a line for each compound read in showing the stock number, formula, molecular weight, and price. (Molecular weight simply is the sum of the atomic weights of the elements in the compound, each atomic weight multiplied by the number of atoms of that element.) Provide suitable column headings and page numbers, printing 40 lines per page. The elements used in Nariz's compounds, along with their atomic weights, are given below:

| *element name* | *abbreviation* | *atomic weight* |
|---|---|---|
| carbon | C | 12.011 |
| hydrogen | H | 1.008 |
| nitrogen | N | 14.007 |
| oxygen | O | 15.999 |
| phosphorus | P | 30.974 |
| sulfur | S | 32.064 |

20. Expand the program in Problem 19 so that it handles the following elements in addition to the ones already listed:

| *element name* | *abbreviation* | *atomic weight* |
|---|---|---|
| bromine | BR | 79.909 |
| chlorine | CL | 35.453 |
| iodine | I | 126.900 |
| magnesium | MG | 24.312 |

21. Write a program that reads character strings consisting only of letters, numbers, and blanks. Two or more blanks will never appear in succession in the middle of a string, so that two blanks in a row signals the end

of a string. No string will require more than one line. For each string read in, the program reproduces the string using large block characters. Each block character is constructed as a pattern made up of many copies of that same character. For instance, the string ABC might be reproduced as

```
  AAAA    BBBBB     CCCC
 AA  AA   BBBBBB   CCCCCC
 AA  AA   BB   BB  CC   CC
 AAAAAA   BBBBB    CC
 AA  AA   BBBBB    CC
 AA  AA   BB   BB  CC   CC
 AA  AA   BBBBBB   CCCCCC
 AA  AA   BBBBB     CCCC
```

You may design your block characters in any way that suits you. The same holds true for the spacing between characters. Thus, depending on your group of block characters, you will be able to determine the longest character string that you can print on one "line." Other design decisions you will have to consider include the following:

(a) The number of blank lines between printout "lines."
(b) The number of printed strings per page.
(c) Placement of output strings:
  (i) Always start at the left end of the "line."
  (ii) Center the string on the "line."
(d) How to handle input strings that are too long:
  (i) Reject and go on to the next one.
  (ii) Truncate (on the right) to the maximum length.
  (iii) Use as many "lines" as necessary, printing the maximum number of characters on each "line."
  (iv) Same as (iii) except that, instead of breaking the "line" at its maximum length, break each "line" at a blank. Thus, every "line" is as long as it can be without breaking in the middle of a word.

22. Text is recorded on 80-character records in columns 1–72. Columns 73–80 are used for numbering the records in sequence. The input material is recorded continuously. That is, no attention is paid to the fact that the text ends at column 72. If we have reached column 72 and we are in the middle of a word, the next letter of that word simply is recorded in column 1 of the next record. There may be one or more blanks between words and there may be one or more blanks between sentences. Punctuation marks (such as period, comma, question mark, exclamation point, quotation marks, etc.) appear immediately after the final letter of a word. A text is as long as it is.

    Write a program that reads such a text and prints it in such a way that each line of print is the same width and each line stops at the end of a word. If that word happens to be followed by one or more punctuation marks, those marks also must be included on that line.

    The way to do this, of course, is to determine how much of the remaining text can go on the next line without breaking a word, and then filling the extra positions with blanks. However, two things must be kept in mind: First, whatever else is done, the last letter of the last word must appear in the last position of the line so that the uniform line width is maintained. Second, the blanks used for padding must be distributed throughout the line so that the printout does not give the appearance of having a bunch of blanks plunked in one place. Use a line width of 72 positions. The number of lines per page is up to you.

23. Expand the program in Problem 22 so that the user can specify a line width that the program is to use in preparing the text printout. (Design your program so that it defines a reasonable minimum width and guards against a specified width that exceeds the physical capabilities of the printing device.)

24. Expand the program in Problem 22 or 23 even further by designing it so that the user can specify the number of lines per page.

25. Expand the program in Problems 22, 23, or 24 even further by designing it so that it can process any number of input texts, where each text (still) may be of any length. If you are expanding the version described in Problem 23 or 24, make it possible for the user to change specifications for each input text.

# 16 *Additional Formatting Features*

This chapter takes a look at some further capabilities that HP FORTRAN 77 provides for describing edit-directed data items and positioning them in records. These features are not necessarily more complicated or "advanced" than the ones discussed in the previous chapter. They were separated because their use tends to be for more specialized situations that occur less frequently.

## 16.1 FURTHER SPECIFICATIONS OF NUMERICAL VALUES

The features discussed here provide more flexibility with regard to the interpretation and appearance of numerical data items. As is true with the other numerical format specifications, the value of the number itself is not changed by these descriptions. That value is determined by the processes that are used to produce it.

### 16.1.1 The G-Specification

In the previous chapter, we worked with two types of specifications (F- and E-) for describing single-precision numerical values. FORTRAN also includes a third specification intended to combine the desirable properties of the other two for output. (In fact, while it can be used as an input specifier, it does not serve any particular purpose that way; FORTRAN treats it as if it were an F-specification. Accordingly, our discussion of this feature will focus strictly on its use with output.) The general form is

G*w*. *d*

where *w* represents the total number of positions occupied by the value in the output record, and *d* indicates the number of decimal places.

**Table 16.1** Behavior of the G-Specification (G*w.d*)
Assume *w* = *d* + 7

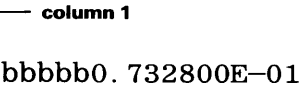
| *numerical value*, n | *appearance* |
|---|---|
| n < 0.1 | 0.dd...dE ee |
| 0.1 < n < 1.0 | 0.dd...dbbbb |
| 1.0 < n < 10.0 | i.dd...dbbbb |
| 10.0 < n < 100.0 | ii.dd...dbbb |
| 100.0 < n < 1000.0 | iii.dd...dbbbb |
| ..... | ..... |
| 10**(d−1) < n < 10**d | ii...i.dbbbb |

In response to this specification, FORTRAN will produce an output value whose format resembles either the E-format or F-format, depending on the numerical value to which the description applies. The rules that underlie this action are shown in Table 16.1. They can be summarized as follows:

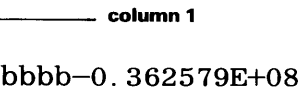
1. When the value being described is less than `0.1` or greater than `10**d` (where *d* has its previous meaning), the result of a G-specification looks like that of an E-specification. For example, suppose `Z1` is declared as a real variable, and it contains a result ready to be printed. Let us say, then, that we print it with the following statements:

```
    WRITE (6,16) Z1
16  FORMAT (1X,5X,G14.6)
```

Now, let us say that `Z1` currently has a value of `0.07328`. The resulting output, then, would look like this:




Since *d* for this specification happens to be 6, the same type of editing will be performed if the absolute value in `Z1` is larger than `10**6`. Thus, if this same statement (in conjunction with the same format description) were to be executed while `Z1`'s value is `-36257900`, the result will look like this:

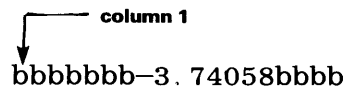



2. If the value being described by a G-specification is not in the ranges specified above, FORTRAN edits the value in an attempt to display it in "natural" form. Specifically, it seeks to use a format similar to the F-specification, with the integer portion to the left of the printed decimal point. The assumption FORTRAN makes here is that the programmer's specification of *w* (in G*w*.*d*) allows enough positions for the value to appear with the E and a signed exponent. Consequently, FORTRAN acts as if there are four extra positions that will not be needed to express the value in F-like format. Thus, instead of using *w* positions for the value, it uses *w*−4 positions. The four unused positions are placed at the right and filled with blanks. The way FORTRAN handles *d*, the number of decimal places, then, depends further on the absolute value. To see how this works, we shall use the same two statements as before with various values assumed in `Z1`:

(1) With `Z1`'s value being `-32.74058`, (i.e., between `1` and `10`), the number of decimal places is reduced automatically by 1 (so that FORTRAN uses 6−1 or 5, even though *d* was 6 in the FORMAT statement). Thus, the 14 original positions would contain the formatted value in the first ten (*w*−4) and blanks in the other four:

bb-3.74058bbbb

Now, if we follow the rest of the specifications in statement 16, the entire output line will be




(2) With `Z1`'s value being `28.3976` (i.e., between `10` and `100`), the number of decimal places will be reduced automatically by two (to make up for the fact that the display will show two digits to the left of the decimal point). Thus, FORTRAN will use 6 − 2 or four decimal places, and the 14 positions set aside for the

value will look like this:

```
bbb28.3976bbbb
```

The entire output line, then, (still using statement 16 for the format description) will appear as follows:

```
bbbbbbbb28.3976bbbb
```

(3) This rule is applied systematically, so that FORTRAN automatically adjusts the number of decimal places (and, therefore, the number of integer positions) in accordance with the value's order of magnitude. Thus, using the same output statements as before, a Z1 of 857463.0 will produce the output line

```
bbbbbbbb857463.bbbb
```

### 16.1.2 Format Descriptions for Sign Control

In all of our dealings with numerical output values, we have not worried about the sign. Instead, we have depended (and shall continue to depend) on FORTRAN's default mechanism to show a negative sign when needed. (Of course, with edit-directed output, it is up to us to make sure that a position is available for that sign to be displayed.)

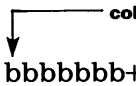
There are special situations, however, in which the programmer may wish to specify some other way of handling signs. The S-, SP-, and SS- format specifications are available for this purpose.

**16.1.2.1 Producing Visible + Signs: The SP-Specification** Normally, FORTRAN will place a visible sign in front of a numerical output value only when that value is negative (i.e., the positive signs are suppressed). It is possible to force the appearance of positive signs as well by using the SP-specification to change FORTRAN's behavior. This feature works as follows: When FORTRAN runs across an SP-specification in an output format description, all numerical descriptions in that statement *following* the SP (i.e., I-, E-, D-, F-, and G-specifications) will produce values with visible + signs. In effect, then, this specification acts like a switch that turns the visible + sign mechanism on for a particular statement or part of a statement.

Using the output statements in the previous section as an example, suppose we rewrite statement 16 so that it reads as shown below. (The associated output statement is included for convenience:)

```
      WRITE (6,16) Z1
   16 FORMAT (1X,5X,SP,G14.6)
```

Suppose that at this instant Z1's value is 28.3976. This time, the resulting output line will look like this:




```
bbbbbbb+28.3976bbbb
```

(Compare this output with the line in the previous section for the same Z1.)

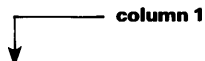
**16.1.2.2 Restoring the Sign Default: The S-Specification** If the SP-specification is used in a particular format statement, the use of visible + signs will remain in effect for that statement, with the system reverting to its normal practice once that statement is com-

pleted. (Of course, if the statement happens to be in a loop, the + signs will appear each time that statement is used.)

The S-specification can be used to turn off this mechanism at any desired point. To illustrate, suppose H1, H2, and H3 all are declared as real variables, and their current values are 65.34, 8.96, and 71.05, respectively. If we were to print these variables with the statements

```
   WRITE (6,26) H1,H2,H3
26 FORMAT (1X,3(5X,F6.2))
```

the resulting output line would be as follows:

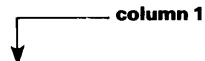



```
bbbbb b65.34 bbbbb bb8.96 bbbbb b71.05
```

Changing the format description to

```
26 FORMAT (1X,SP,3(5X,F6.2))
```

changes the output line so that the + signs are visible:

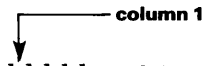



```
bbbbb +65.34 bbbbb +b8.96 bbbbb +71.05
```

If statement 26 were to be written as follows:

```
26 FORMAT (1X,SP,5X,F6.2,S,2(5X,F6.2))
```

the resulting output line would be produced with the visible + sign mechanism activated only for the first output value:

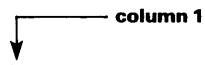



```
bbbbb +65.34 bbbbb bb8.96 bbbbb b71.05
```

Finally, to show the mechanism activated, turned off, and then reactivated, we can rewrite statement 26 as follows:

```
26 FORMAT (1X,SP,5X,F6.2,S,5X,F6.2,5X,SP,F6.2)
```

and the resulting output will be




```
bbbbb +65.34 bbbbb bb8.96 bbbbb +71.05
```

**16.1.2.3 Suppression of Visible + Signs: The SS-Specification** Although suppression of visible + signs in numerical output values is the default used by many FORTRAN systems, it is not necessarily used by all systems. That is, it is legal to have a FORTRAN system in which the default is to show all + signs. For this type of situation, the SS-specification provides a way to turn this mechanism off, so that visible + signs are suppressed. In a sense, then, this is the "opposite" of the SP-specification discussed earlier. (Use of the SP-specification would not make sense for this type of system since the + sign is shown all the time anyway.)

The S-specification works in conjunction with the SS-specification in the same way as it does with SP. That is, after the SS-specification has been used in a particular statement to suppress the appearance of visible + signs, the S-specification can turn that mechanism on again. For HP FORTRAN 77, the SS-specification has the same effect as the S-specification.

### 16.1.3 Scaling of Edit-Directed Values

In certain types of work, the numerical values that are commonly used are inconvenient to handle because they tend to be very large or very small. For various reasons, workers in these fields have not adopted scientific notation as a way around this problem. Instead, the numbers are *scaled*. That is, they are multiplied by some power of ten (a *scale factor*) to bring their magnitude to a more manageable level for input and output. (Of course, the scale factor is taken into account during computational processes so that the results are produced properly.) The scale factor usually is set by agreement to a particular fixed value, so that everyone dealing with those numbers knows how to interpret them.

For instance, many corporations are used to reporting financial data in terms of thousands of dollars. Thus, a figure of $376,123,000 is often reported as $376,123. The reader of such a report is reminded that the figure represents thousands of dollars rather than dollars, and the actual magnitude would be used in performing all of the related computations. Thus, before the value is to be printed, it has to be scaled. In this example, scaling involves division by a thousand or, more precisely, multiplication by 0.001 or `10**-3`. The scale factor, then, is −3. *The scale factor is the power of ten by which the actual number has to be multiplied in order to produce the desired scaled number.*

Another relatively common example is seen in air pollution work where the concentration of a particular substance in the air, while medically significant, is quite low on an absolute basis. For instance, a concentration of 0.0000026 parts of substance per part of air is not an unusual order of magnitude. Although these numbers must be used for computations, they often are reported in terms of parts per million. Thus, the value just shown would be reported as 2.6 parts per million. Using the definition given in the previous paragraph, the scale factor in this case is +6.

FORTRAN's editing facilities can be instructed to apply a designated scale factor to real input or output values under certain conditions. This is done by means of a format specification whose general form is

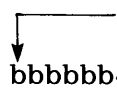
*k*P

where P indicates that scaling is to take place, and *k* is an integer constant. (The + sign may be omitted when *k* is positive, but it is a good idea to show it for all cases.)

For example, if `PROFIT` is a real variable with a value of `42763.68` (representing dollars), and we wish to display it in thousands of dollars with the cents truncated, we can say

```
      WRITE (6,28) PROFIT
28    FORMAT  (1X,5X,-3PF7.3)
                    or
28    FORMAT  (1X,5X,-3P,F7.3)
```

The resulting output line will show




```
bbbbbb42.673
```

Similarly, if a variable named `NO2` is recorded in columns 6–7 of an input record as `83`, and we want it to be interpreted as 0.83 parts per million (i.e., 0.00000083 parts per part of air), we can read it as follows:

```
      READ  (5,25) NO2
25    FORMAT  (5X,6PF2.2)
                    or
25    FORMAT  (5X,6P,F2.2)
```

When FORTRAN begins processing an edit-directed input or output statement, the assumed scale factor is 0 (i.e., no scaling is done at all). If it finds a P-specification in the associated format description, it scales accordingly until another P-specification changes the scaling or eliminates it. Then, when the next edit-directed input or output statement is processed, the scale factor is reset to 0. Note that the P-specification is not attached to any one data item. Rather, like the slash, SP-, and S-specifications, it provides FORTRAN with information on how to perform subsequent editing.

Although scaling is a useful technique in certain circumstances, it is specialized. Its use, therefore, is discouraged unless the programmer is convinced that the application of scaling in a particular situation will simplify things for himself or for the program's users. Accordingly, we shall not study it in any more detail. Instead, the specific rules for input and output scaling will be summarized.

**16.1.3.1 Scaling of Input Data** We have seen the effect of scaling when an input value is read with an F-specification. A scale factor also can be applied when E-, D-, or G-specifications are used. However, regardless of which of these is used, the scale factor will be ignored if the numerical value being described by that specification is recorded in scientific notation, i.e., if it has an exponent.

**16.1.3.2 Scaling of Output Value** The treatment of scaled variables with F-specified output has been introduced and is straightforward enough. Although scaling makes less sense with a D- or E-specified output, FORTRAN does accept such editing. When a scaling factor $k$ is applied to such an output item, FORTRAN multiplies the fractional portion by `10**`$k$. In addition, it compensates by decreasing the exponent value by $k$. Thus, with E- and D- edited output items, the scaling does not change the value; rather, it changes the way the value looks. The exact appearance is governed by the value of $k$ as follows:

1. If $k$ is negative, the fractional portion of the output value is displayed with $-k-$ zeros immediately to the right of the decimal point. Since the specified length of the fraction ($d$) is retained, this means that the rightmost $-k-$ digits are truncated. For example, suppose a variable Z has a value of −324.7 and we print it with a format specification of E14.5. As expected, the result will look like this:

```
bb−0.32470E+03
```

If we apply a scale factor of −2 (so that the new specification is −2P, E14.5), the result will appear as follows:

```
bb−0.0032E+05
```

2. When $k$ is greater than zero, $k-1$ additional digits are placed in front of the decimal point. To keep the total number of digits the same as in the specification, these positions are taken from the ones to the right of the decimal point. For example, if we were to take the same value as above and print it with a scale factor of +2 (so that the new specification would say +2P, E14.5), the result would be

```
bb−32.4700E+02
```

**16.1.3.3 Scaling and the G-Specification** Scaling also is legal for numerical output values edited by a G-specification. (It is ignored for G-edited input data.) However, its use by FORTRAN is restricted as follows: Even when a scale factor is specified, it is not applied when the numerical value is within the range (see Table 16.1) where the F-like output form can be used. Outside of that range, scaling affects G-edited output in the same way it affects E-edited output.

### 16.1.4 Treatment of Input Blanks

Chapter 13 indicated that it is possible (by means of the BLANKS= specifier) to indicate how blanks are to be interpreted when embedded in numerical input on a given unit. Two additional format specifications make it possible for the programmer to control the handling of blanks for individual input variables. The major reason for including these features is to enable FORTRAN to read and interpret numerical values that are recorded anywhere in the columns reserved for them. In earlier days, some people were rather casual about the way they recorded their input data, so that this feature can be of help in reading such old numbers. However, it goes against the more desirable practice (emphasized in the previous chapter) of making sure that all values for a given variable are entered consistently. Consequently, this feature will be mentioned here only to make programmers aware of its existence.

**16.1.4.1 The BN-Specification** When FORTRAN starts executing an edit-directed READ statement, it treats embedded blanks in numerical values either as zeros or null characters, depending on the definition set up for the unit. Usually, such embedded blanks are treated as null characters. In effect, FORTRAN moves the digits to the right, "squeezing out" all the blanks and placing them at the left. For example, suppose an input variable recorded in four columns is supposed to be interpreted as `F4.1`. (That is, the rightmost digit is to be treated as a single decimal place.) If those columns contained `3274`, the value eventually stored is `327.4`. So far, so good. If the four columns were to contain `b327` instead, the same `F4.1` specification would produce a stored value of `32.7`. Fine. Now, suppose these four columns contained `327b`. (Old, messy data.) The rightmost blank would be "squeezed out" and the result would be interpreted as `32.7` again.

This treatment of blanks remains in effect until it is changed (see next section) by the BZ-specification. The BN-specification reverses it.

**16.1.4.2 Blanks as Zeros—The BZ-Specification** The specification

```
BZ
```

instructs FORTRAN to interpret embedded blanks as zeros. Thus, if four columns contained `327b` and their treatment were to be directed by a format specification of BZ, `F4.1`, the stored value would be `327.0`. Similarly, `3b27` would be stored as `302.7`.

The BZ-specification remains in effect until it is changed by the BN-specification or until FORTRAN reaches the end of the statement in which it is used.

## 16.2 POSITION CONTROL IN AN EDIT-DIRECTED RECORD

The X-specification has been introduced and used extensively as a way of skipping positions in an input record or filling output positions with blanks. Often, the programmer may find it more convenient to think in terms of the next position of interest rather than the number of positions that need to be skipped to get there. That is, rather than saying, "Skip the next nine positions; that will get me to the next position at which I want to read or write," the programmer might prefer to say, "Move to position 27, skipping however many positions that need to be skipped to get there."

FORTRAN provides three format specifications that may be used to produce this type of effect. They all begin with the letter `T` to indicate their basic similarity to the tabulator key on a typewriter.

### 16.2.1 The T-Specification

This specification, whose general form is

```
Tn
```

causes FORTRAN to move to position *n* of the current record, where *n* is an unsigned nonzero integer constant. As is true with the X-specification, the T-specification does no editing. It merely sets the position in the record at which the next editing activity is to take place.

As an example, we shall set up an output format and specify it in two ways: once with X-specifications and again with T-specifications. Here is what we want to print:

| *columns* | *variable name* | *typical value* |
|---|---|---|
| 6–10 | B1 | −23.6 |
| 14–20 | CARRY | 338.47 |
| 31–50 | NAME | characters |
| 58–60 | NUM | 328 |

Our output statement, then, says

```
WRITE (6,17) B1,CARRY,NAME,NUM
```

Using X-specifications, statement 17 says

```
17 FORMAT (1X,5X,F5.1,3X,F7.2,10X,A20,7X,I3)
```

We can obtain exactly the same result with the following:

```
17 FORMAT (1X,T6,F5.1,T14,F7.2,T31,A20,T58,I3)
```

Note that the 1X (or ' ') still is needed for carriage control. It also should be pointed out that the T-specification, like the X-specification, does not actually insert blanks in an output record by itself. The blanks are placed in the skipped positions as part of the process that edits and produces the next output item in the list. Thus, in the second version of statement 17, the T6 specification informs FORTRAN that the next item (B1 in this case) is to be printed starting in column 6. However, it is the F5.1 specification that actually fills in the skipped columns (i.e., columns 1–5) with blanks.

### 16.2.2 The TR-Specification

This is similar to the X-specification in that it describes a skip to the right. Thus

TR8 and 8X

will produce the same effect.

### 16.2.3 The TL-Specification

To keep things symmetrical, FORTRAN also enables the programmer to establish a new position to the left of the previous one. Thus, if we write

TL9

it would be the same as saying

−9X

Of course, the −9X is illegal, but the TL9 is perfectly fine. FORTRAN automatically establishes a position at the beginning of the record if there is an attempt to move the position too far to the left. For instance, if the specification TL9 appears in a FORMAT statement at a point where the current position is, say, 7, the move of 9 positions to the left, if carried out, would go past the beginning of the record. Obviously, this is an impossible request, in which case FORTRAN automatically resets at position 1.

The ability to move back and forth in a record may not seem particularly useful right now. That is quite understandable, since one would think that once we have dealt with a particular position in a record, we would be through with it. However, the power of the

TL-specification lies in the flexibility that it can provide for certain input situations: When we use the specification to direct FORTRAN back to a position from which we already have read, we can *reread* the same data in a different way.

We shall illustrate with a simple example: Suppose we have declared variables WR and TTLX as real and integer, respectively. We are ready to read a line that contains bb34892 in the first seven columns. The statements

```
      READ  (*, 19) WR, TTLX
19  FORMAT  (2X, F5.3, TL7, 3X, I4)
```

cause the following to happen:

1. The first two columns are skipped.
2. The F5.3 format specification is applied to columns 3–7. Since this is associated with variable WR in the input list, a value of 34.892 is stored in WR.
3. The TL7 sends FORTRAN back to position 1 of the record.
4. The 3X causes the next three columns to be skipped, but because of the TL7 specification, those three columns are columns 1–3.
5. The final specification, associated with variable TTLX, is applied to columns 4–7, with the result that a value of 4892 is stored in TTLX.

This capability provides advantages in special situations beyond the scope of this text. Consequently, having described what it does and how it works, we shall leave it at that.

## 16.3 RUN-TIME FORMAT DESCRIPTIONS

The editing features that we have examined and used, extensive as they may be, still impose one restriction that may make things difficult for us in certain situations. Regardless of which editing features we use, how we combine them, or where we place them (in a separate FORMAT statement or as part of a READ or WRITE statement), we have been required to describe the format completely and exactly. Thus, for input, we know how many variables there will be, in which columns they will appear, and what they will look like. The same is true for output. This has not caused us any particular hardship in that we have been able to describe any format we want to describe. The only problem arises because we had to describe the format in advance.

There are situations in which we would like to say, "I am going to do some reading or writing at this point in the program. However, I will not know what the input or output is going to look like until this program actually runs. Furthermore, the format may change from run to run." Although it is a good practice to try to avoid such situations if we possibly can, there may be occasions where such flexibility must be provided. In general, what is required is the ability to delay a format description until the program actually runs. Appropriately enough, such a specification is called a *run-time format.* (The completely defined specifications that we have been using all along, then, are called *compile-time formats*, since they are specified as part of the input to the FORTRAN compiler.)

We shall discuss a technique for developing and using run-time formats. The basis for this method lies in a FORTRAN feature that enables the programmer to specify a format as part of the input instead of building it into the program. Specifically, it is possible to read a set of specifications (parentheses and all) as a character string and store the information in a character string variable, declared just like any other character variable. Then, the programmer can refer to that character string by naming it in an input or output statement, in which case FORTRAN will use the formatting information in that string. For example, suppose we have the following declarations:

```
REAL              A1, B, STM
INTEGER*2         CRTS
CHARACTER*80      FRMT
```

(FRMT is an 80-character string into which we shall read a format description.) The format

description is read like any other character string:

```
      READ (5,15) FRMT
   15 FORMAT (A80)
```

Such a line might look like this:

```
(3X,I6,2X,3F5.2)
```

Note that the word FORMAT does not appear here, nor does the format description start in column 7. This is not a format *statement*. It is only a format *description*. Now we can read data using the formatting information just received:

```
READ (5,FRMT) CRTS,A1,B,STM
```

FORTRAN will use this formatting information just as if it were in a regular program statement. Then, at some later point (either in the same run or, more likely, in another run) a new set of specifications can be read in and used on a different data collection. Note, however, that in this example the statement that reads the data is fixed. Regardless of the actual format, this statement expects each data record to contain CRTS, A, B1, and STM in that order. Note also that it does *not* expect these items all to be on a single line. (A little later, we shall look at the technique for handling variations in the input or output list as well as the format.)

**Example 16.1** To illustrate the use of run-time formatting in a practical context, we shall design a general purpose program for accumulating and printing frequency distributions. Basically, this type of program processes integer variables where each variable will have a value selected from a limited number of choices. The program simply counts and prints the number of occurrences of each value for each variable. A typical use for such a program might be to prepare the results of a survey in which people are asked to answer a series of multiple choice questions.

Each record consists of a single line, but there may be any number of records in an input file. Each variable is a one-column integer so that the program may be called upon to process up to 80 variables in any run. (Actually, an 80-variable line is unlikely; for example, in the type of survey mentioned above, several of the columns normally would be reserved for some identifying data such as a questionnaire number.) There are no specific columns in which the variables need be recorded. (Of course, the format, whatever it is, must be consistent for all the records in the file.) The values for a variable may be `0, 1, 2, . . . . . , 9`. There are no blank data. For each variable processed, the program prints a line showing the number of 0s, 1s, 2s, etc. (A possible format is shown in Figure 16.1.)

Since all the variables are integers, the task of reading and storing them becomes simple. We shall set up an 80-element integer array (VARS) and use as many of them for a particular run as we need. The number of variables (NUMV) will be specified as part of the input, and so will the format. Input for each run, then, will consist of:

1. A line with # in column 1 and the number of variables in columns 2–3.
2. A line with the parenthesized format description starting in column 1. We shall store this information in an 80-character string named DESCR. The assumption here is that 80 characters will be sufficient. Since a character string (and a format description) can be much longer than that,

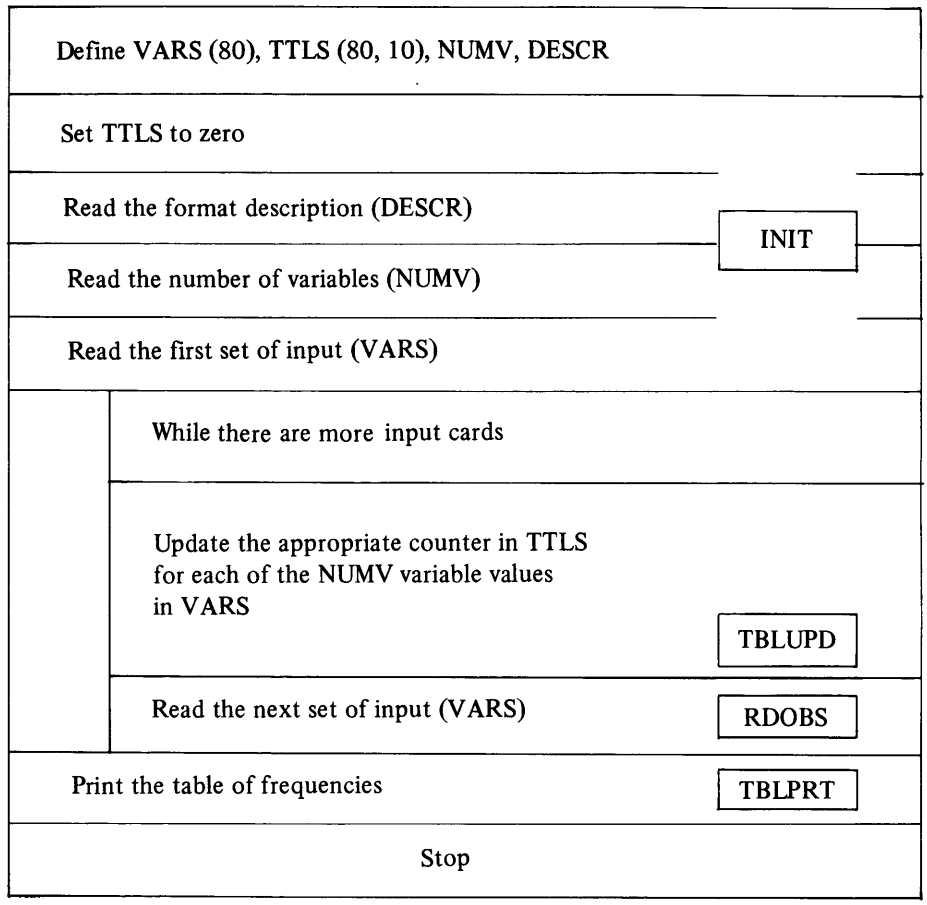
| VAR. NO. | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 2 | 5 | 2 | 0 | 2 | 0 | 1 |
| 2 | 2 | 1 | 1 | 0 | 3 | 2 | 2 | 1 | 0 | 2 |
| 3 | 2 | 1 | 0 | 2 | 3 | 0 | 3 | 2 | 0 | 1 |
| 4 | 2 | 1 | 2 | 1 | 2 | 3 | 1 | 0 | 1 | 1 |
| 5 | 3 | 1 | 2 | 0 | 3 | 1 | 1 | 1 | 2 | 0 |
| 6 | 0 | 1 | 4 | 4 | 1 | 4 | 0 | 0 | 0 | 0 |
| 7 | 4 | 0 | 2 | 3 | 2 | 1 | 1 | 1 | 0 | 0 |
| 8 | 1 | 1 | 3 | 2 | 2 | 2 | 1 | 2 | 0 | 0 |

NUMBER OF OBSERVATIONS: 14

**FIGURE 16.1** Format for Output Showing Frequency Distribution.

```
"Define VARS(80), TTLS(80,10), NUMV, DESCR."
"Set TTLS to zero." "Read the format description (DESCR)."
"Read the number of variables (NUMV)."                          INIT
"Read the first set of observations."
while there are more input data:
    "Look at the values in each of the
    NUMV variables and add 1 to the                             TLBUPD
    appropriate counters in TTLS."
    "Read the next set of observations."                        RDOBS
endwhile
"Print the frequency table."                                    TBLPRT
"Stop."
```

**(a)**




**(b)**

**FIGURE 16.2** **(a)** Pseudocode for Example 16.1's Main Program. **(b)** N-S Diagram for Example 16.1.

there is nothing to stop the programmer from designing for a longer format description if the situation calls for it.

3. Data records (any number) whose organization is consistent with the format described in (2).

A data item may have one of ten possible values, so we need ten counters in which to accumulate the frequencies for each of 80 variables. A straightforward approach is to provide a two-dimensional (80 × 10) *array* of integers (`TTLS`) and use the first `NUMV` rows for a given run.

```
C*************************************************************
C                  EXAMPLE 16.1 - THE MAIN PROGRAM            *
C*************************************************************
C  NUMV:     THE NUMBER OF VARIABLES TO BE PROCESSED IN A RUN  *
C  VARS:     AN ARRAY CONTAINING THE DATA FROM A SINGLE LINE   *
C  DESCR:    A VARIABLE CONTAINING THE FORMAT DESCRIPTION FOR  *
C            THE CURRENT RUN                                   *
C  TTLS:     A SET OF COUNTERS IN WHICH THE FREQUENCIES WILL BE*
C            ACCUMULATED                                       *
C*************************************************************
C  OVERALL PROCESSING STARTS WITH TWO INPUT LINES THAT ARE NOT *
C  PART OF THE DATA TO BE COUNTED. WE SHALL REFER TO THESE AS  *
C  CONTROL LINES. THE FIRST CONTAINS NUMV AND THE SECOND GIVES *
C  THE FORMAT INFORMATION TO BE STORED IN DESCR. THEN, THAT    *
C  INFORMATION WILL BE USED TO INTERPRET THE DATA  SO THAT     *
C  THE FREQUENCY ANALYSIS CAN BE DONE PROPERLY FOR THAT FILE.  *
C  A SEPARATE SUBPROGRAM (INIT) INITIALIZES THE COUNTERS AND   *
C  PREPARES THE PROGRAM FOR THIS PARTICULAR RUN.               *
C*************************************************************
      PROGRAM        EX1601
      IMPLICIT       NONE
      INTEGER*2      VARS(80),NUMV,TTLS(80,10)
      CHARACTER*80   DESCR
      LOGICAL        EOF

      CALL INIT (NUMV,DESCR,VARS,TTLS,EOF)
      DO WHILE (.NOT. EOF)
         CALL TBLUPD (NUMV,VARS,TTLS)
         CALL RDOBS (NUMV,DESCR,VARS,EOF)
      END DO

      CALL TBLPRT (NUMV,TTLS)
      STOP
      END
```

**(a)**

```
C*************************************************************
C                             INIT                            *
C*************************************************************
C  THIS SUBROUTINE SETS THE COUNTERS (TTLS) TO ZERO, READS THE *
C  CONTROL INFORMATION, AND PRIMES THE PROGRAM BY READING THE  *
C  FIRST DATA LINE. NOTE THAT THE LOGICAL VARIABLE EOF, USED AS*
C  AND END-OF-FILE INDICATOR, IS PART OF THE INPUT SUBROUTINE  *
C  (RDOBS) SO THAT THE PROGRAM IS DESIGNED TO DETECT A SITUA-  *
C  TION IN WHICH THERE ARE CONTROL LINES BUT NO DATA LINES.    *
C*************************************************************
      SUBROUTINE       INIT (N,FRMT,OBS,CT,SW)
      IMPLICIT         NONE
      INTEGER*2        N,OBS(80),CT(80,10),R,C
      CHARACTER*80     FRMT
      LOGICAL          SW

      SW = .FALSE.
      DO R = 1,80
        DO C = 1,10
          CT(R,C) = 0
        END DO
      END DO

      FRMT = ' '
      READ (*,15) N
   15 FORMAT (1X,I2)
      READ (*,25) FRMT
   25 FORMAT (A80)
      CALL RDOBS (N,FRMT,OBS,SW)
      RETURN
      END
```

**(b)**

*(Continued)*

**FIGURE 16.3** **(a)** Main Program for Example 16.1. **(b)** INIT Subroutine for Example 16.1.

```
C*****************************************************************
C                          RDOBS                                 *
C*****************************************************************
C  THIS SUBROUTINE USES THE CONTROL INFORMATION (NUMV AND DESCR)*
C  TO READ AND INTERPRET A DATA LINE. IF THE END OF FILE HAS    *
C  BEEN REACHED, RDOBS SETS THE LOGICAL VARIABLE EOF TO .TRUE.  *
C  (NOTE THAT EOF IS INITIALIZED TO .FALSE. BY INIT)            *
C  I IS AN INTEGER VARIABLE THAT IS USED TO STORE THE SUB-      *
C  SCRIPT DENOTING THE COLUMN OF THE COUNTER ARRAY THAT WILL BE *
C  INCREMENTED FOR A PARTICULAR VARIABLE. SINCE THE VALUE IN A  *
C  VARIABLE MAY BE ANYTHING FROM 0-9 AND THE COLUMNS OF THE     *
C  ARRAY OF COUNTERS ARE NUMBERED (I.E., HAVE SUBSCRIPTS OF)    *
C  1-10, THE PROPER SUBSCRIPT, I, IS DETERMINED BY ADDING 1 TO  *
C  THE VALUE FOUND IN THE VARIABLE THE LOOP IS PROCESSING,      *
C  I.E., OBS(R).                                                *
C*****************************************************************
      SUBROUTINE       RDOBS (N,FRMT,OBS,SW)
      IMPLICIT         NONE
      INTEGER*2        M,N,OBS(80)
      CHARACTER*80     FRMT
      LOGICAL          SW

      READ (*,FRMT,END=199) (OBS(M),M=1,N)
      GO TO 77
  199 SW = .TRUE.
   77 RETURN
      END
```

**(c)**

```
C*****************************************************************
C                          TBLUPD                                *
C*****************************************************************
C  THIS SUBROUTINE UPDATES THE COUNTERS TO REFLECT THE EFFECT   *
C  OF THE VALUES IN THE CURRENT DATA RECORD                     *
C*****************************************************************
      SUBROUTINE       TBLUPD (N,OBS,CT)
      IMPLICIT         NONE
      INTEGER*2        I,R,N,OBS(80),CT(80,10)

      DO R = 1,N
        I = OBS(R) + 1
        CT(R,I) = CT(R,I) + 1
      END DO
      RETURN
      END
```

**(d)**

**FIGURE 16.3** **(c)** Data Input Subroutine for Example 16.1. **(d)** Table Updating Routine for Example 16.1.

Now we can look at the program organization. In a general sense, the processing characteristics of this program are not terribly different from other report preparation programs considered earlier. No matter how complicated the report itself may get, the same simple structure continues to be efffective in the case of this example, we identify the following processing components:

```
C*****************************************************************
C                              TBLPRT                             *
C*****************************************************************
C  THIS ROUTINE PRINTS THE RESULTS AFTER CALLING LBLPRT TO       *
C  PRINT THE HEADINGS. IT ALSO COMPUTES AND PRINTS A GRAND       *
C  TOTAL. SINCE THE SPECIFICATIONS SAY THAT THERE ARE NO DATA    *
C  MISSING, THE GRAND TOTAL (GRDTTL) CAN BE OBTAINED BY ADDING   *
C  THE TEN VALUES IN ANY ROW OF COUNTERS. THE EASIEST THING TO   *
C  DO, AND THE CLEAREST TO UNDERSTAND, IS SIMPLY TO USE THE      *
C  FIRST COLUMN FOR THIS PURPOSE.                                *
C*****************************************************************
      SUBROUTINE      TBLPRT (N,CT)
      IMPLICIT        NONE
      INTEGER*2       N,CT(80,10),GRDTTL,LINE,V,FR

      LINE = 0
      GRDTTL = 0

      DO V = 1,10
        GRDTTL = GRDTTL + CT(1,V)
      END DO

      DO V = 1,N
        IF (MOD(LINE,40) .EQ. 0) THEN
          CALL LBLPRT
          LINE = 0
        ELSE
        END IF
        WRITE (*,27) V,(CT(V,FR),FR=1,10)
   27   FORMAT (1X,8X,I2,9X,10(I6,4X))
        LINE = LINE+1
      END DO

      WRITE (*,28) GRDTTL
   28 FORMAT (//10X,'NUMBER OF OBSERVATIONS: ',I6)

      RETURN
      END
```

**(e)**

```
C*****************************************************************
C                              LBLPRT                             *
C*****************************************************************
C  THIS ROUTINE PRINTS THE COLUMN HEADINGS AT THE TOP OF EACH    *
C  PAGE OF OUTPUT. IT IS TRIGGERED BY A LINE COUNTER IN TBLPRT.  *
C  SINCE LBLPRT DEALS WITH NO VARIABLES, THERE IS NO ARGUMENT    *
C  LIST.                                                         *
C*****************************************************************
      SUBROUTINE LBLPRT
      WRITE (*,7)
    7 FORMAT ('1')
      WRITE (*,17)
   17 FORMAT (1X,5X,'VAR. NO.',T23,'0'.9X,'1',9X,'2'.9X,'3',9X,
     1        '4',9X,'5',9X,'6',9X,'7',9X,'8',9X,'9'///)
      RETURN
      END
```

**(f)**

**FIGURE 16.3** **(e)** Output Routine for Example 16.1. **(f)** Label Printing Routine for Example 16.1.

1 .A processing sequence to get the run started. When this sequence concludes, the program should be "primed" with the format information and the first data record, so that it is ready to begin the actual preparation of the frequency tables.
2. A processing loop (in the form of a *DO-WHILE* construct) that processes the data to prepare the frequency count and reads the next set of values. This continues as long as there are input values available.
3. A final processing sequence to print the report.

We can translate this structure directly into a main program before defining the actual details. Pseudocode and N-S representation are seen in Figure 16.2, and the FORTRAN statements are in Figure 16.3(a). Looking at the processing a little more closely, we identify the following activities:

1. Consistent with the duties mentioned, before, a subroutine named `INIT` must set the counters to zero, read the control information (`NUMV` and `DESCR`), and read the first data record.
2. The processing loop calls a routine named `TBLUPD` to update the counters with the data from the current record, and it calls `RDOBS` to read the next record and test for end of file. The same routine also is used by `INIT` to read the first data record.
3. The final component (`TBLPRT`) prints the results. A separate module (`LBLPRT`) takes care of column headings.

The statements for the subprograms are given in Figure 16.3(b) through 16.3(f). The output sample shown in Figure 16.1 is produced from the input sample given in Figure 16.4. (Obviously, it is unnecessary to show all of the data. However, pay particular attention to the format description and compare it with some of the data that follow it.)

Since a run-time format specification is read in and stored as an ordinary character variable, there is nothing in the language to stop the programmer from reading and storing several such format descriptions in a character array. For instance, if we had the following declaration:

```
CHARACTER*80      FMTS(8)
```

and we read and stored eight format descriptions, then one of these eight elements could be the format reference for a subsequent input or output statement. For example:

```
WRITE (6,FMTS(3))    output list
```

## 16.4 INTERNAL FILES AND VARIABLE FORMATTING

Another way of making edit-directed input/output operations more flexible is to provide the program with a choice of several format descriptions, any of which may be used with a particular `READ` or `WRITE` statement. However, the selection is not made until the program actually runs. We can do this by using an extension of the technique described in the previous section: Instead of reading a format description into a

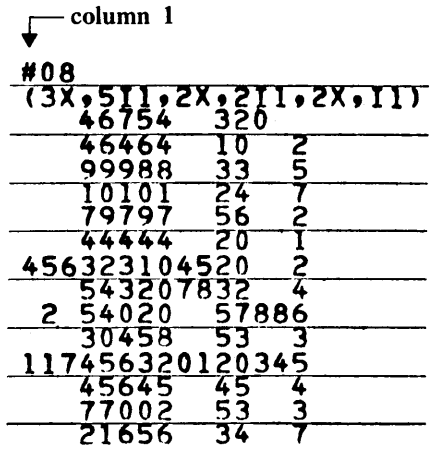



**FIGURE 16.4** Portion of Input Used to Generate Figure 16.1.

character string variable (as Example 16.1 illustrates), we can define the group of format descriptions by assigning them to character string variables. (Alternatively, we can assign them to elements of a character array, or we can even make them substrings of a single character variable, if that is more convenient for our particular requirements.) Then (as the last part of Section 16.3 shows), we can read in (or compute) some indicator that guides the selection of the format appropriate for that situation.

### 16.4.1 Internal Files

Although this does provide some flexibility, we can accomplish much the same thing with an ordinary IF-THEN-ELSE or CASE construction in which an input or output operation is associated with different formats:

```
IF (condition1) THEN
  READ (*,14) inputlist
ELSE IF (condition2) THEN
       READ (*,15) inputlist
     ELSE IF (condition3) THEN
          READ (*,16) inputlist
     ELSE
          CALL RDERR
END IF
```

In fact, this construction can be as flexible as we need to make it since the input/output statements, as well as the formats, may be different here.

In many situations, this flexibility still is inadequate: There are occasions where several formats need to be used in the same run *at the same point in the program.* A typical setup is one in which a file may contain a mixture of records, all of which are to be read and processed as part of a general cyclic operation. We may know what each type of record looks like (and, therefore, how it is to be interpreted and processed), but we do not know how many there are of each type, nor do we know the order in which they will be presented to the program.

A simple way to prepare data for this kind of setup is to include a variable in the same position of each record that indicates what kind of record that is. Then, the program can be designed to use that information in determining how the rest of the record is to be handled. Thus, in a sense, each record defines itself, so that it does not matter how the records are intermixed. Users of programs like that very much. However, it means that in order to make use of such a design, we have to be able to read a record, look at part of it, *make a decision based on what we find*, and then read it again.

At first glance, it might seem that the TL-specification (described in Section 16.2.3) is just the ticket. However, a little thought quickly establishes that it will not do what we want here: Since it is part of a regular format description, we cannot stop in the middle, do some decision making, and then go back to the same record. All the TL-specification lets us do is move backward in a record; but we still are in the midst of an input or output statement and, once we leave that statement, we are through with that record.

FORTRAN 77 provides help with such problems by means of the *internal file*. Recall (from Chapter 13) that we can read an edit-directed input record into a character string and then pretend that this is itself a file, ready to be read again. Thus, we can read the record from its external file (the first time) using a format description that provides the program with the information it needs to determine how to treat the rest of the record. Then, having stored the record in a character string, we treat that string as an internal file by specifying another READ statement, associated with another format description. This time, we do not specify the external unit from which the record came originally. (Such a specification, of course, would bring in the next record.) Instead, we specify the name of

specify the name of the character string (i.e., the internal file) in place of the unit number. As a result, FORTRAN will process the statement like any other edit-directed READ statement, filling the requirements of the input list as guided by the format specifications.

### 16.4.2 Data Transmission with Internal Files

We shall look at this technique in a little more detail by setting up a general illustration of the basic mechanism: Suppose our input records follow one of four possible formats. Each of the formats is known and, therefore, can be defined in the program. The format for a particular record is indicated by an integer (1, 2, 3, or 4) in column 1 of that record. The remaining columns contain information consistent with the designated format.

A straightforward way to handle this is to declare an integer variable (we shall call it FTYPE) in which to store the format indicator, and a character variable (which we shall call INFILE) in which to store the other 79 columns from the input record. INFILE then will be our internal file when we reread its contents (Figure 16.5).

```
                .......
                .......
                INTEGER*2          FTYPE
                CHARACTER*79       INFILE
                .......
                .......
C ---------------READ THE RECORD FROM ITS EXTERNAL UNIT ----------------
                READ (*,12) FTYPE, INFILE
             12 FORMAT (I1,A79)
C -------------NOW WE CAN DETERMINE WHICH FORMAT TO FOLLOW --------------
                IF (FTYPE .EQ. 1) THEN
                  READ (INFILE,21) inputlist1
             21   FORMAT (formatspec1)
                  .......
                ELSE IF (FTYPE .EQ. 2) THEN
                    READ (INFILE,22) inputlist2
             22     FORMAT (formatspec2)
                    .......
                  ELSE IF (FTYPE .EQ. 3) THEN
                      READ (INFILE,23) inputlist3
             23       FORMAT (formatspec3)
                      .......
                    ELSE IF (FTYPE .EQ. 4) THEN
                        READ (INFILE,24) inputlist4
             24         FORMAT (formatspec4)
                        .......
                    ELSE
                      CALL FMTERR
                END IF
                .......
                .......
```

**Figure 16.5**

Remember that the record in the internal file (in this example) is 79 characters long (not 80), because the first position of the original record was not read into the same character string as the other 79. Consequently, the first position of the internal record is what used to be position 2, and the format description must be constructed with that in mind.

**Example 16.2** The Baling Wire and Prayer Company sells a variety of building materials to the construction trades. Each type of material is priced differently because of the way it is sold. For example, some materials are manufactured as standard sheets and, therefore, are sold by the sheet. Others, while still produced in sheets, are cut up and sold by the square meter. Still others, also in sheets, come in various thicknesses which affect the price in such a way that the company finds it appropriate to sell these by the cubic meter. Some liquids are sold by the liter while others are sold only by the barrel.

The company wants its daily sales reports to show this diversity, so that the amount of each purchase is to be displayed with its proper units. Consequently, the first digit of each six-digit product number indicates the general product category.

1. Products identified by i.d. numbers 100000–199999 are sold by the linear meter (like lumber or pipe) and priced to the nearest meter.
2. Products with i.d. numbers 200000–299999 are sold by the rectangular piece, and priced to the nearest hundredth of a square meter. The size of a piece is reported (in the input) by recording its length and width, each to the nearest tenth of a meter. For any given purchase of such material, all the rectangular pieces are the same size.
3. Products with i.d. numbers 300000–399999 are sold by the sheet, with the price per sheet specified to the nearest cent.
4. Products with i.d. numbers 400000–499999 are priced per cubic meter but sold in equal sized rectangular pieces. Accordingly, data for such a purchase shows the length and width, each to the nearest tenth of a meter, and the thickness in millimeters to the nearest tenth of a millimeter.
5. Products with i.d. numbers 500000–599999 are sold in bulk by the cubic meter (like sand or gravel), with the amount of purchase recorded to the nearest tenth of a cubic meter.
6. Products identified with numbers in the range 600000–699999 are sold by the liter (to the nearest tenth of a liter).
7. Products with i.d. numbers of 700000–799999 are sold by the barrel.
8. Products with i.d. numbers of 800000–899999 are sold by the kilogram, with amounts reported to the nearest tenth of a kilogram.

Each purchase is reported on a separate record. The contents and format depend on the product type. However all records, regardless of product type, show the product i.d. in columns 1–6, the 6-digit customer i.d. number in columns 7–12, the date of the order (`mmddyy`) in columns 13–18, and the salesperson's identification (three letters) in columns 68–70. No decimal points are shown; all are implied. These various formats are shown in Table 16.2.The required program is to compute a total price for each purchase and, where it is not given directly as part of the input, it is to compute the total amount of material purchased. For each purchase, the program is to print a line of output echoing the input (except for the date, which need appear only once at the top of each 40-line page). In addition, each line of output is to show the units in which the purchase is reported (`SHEETS`, or `BBL`, or `LITERS`, or `CU. MTRS`, etc.) and the total amount charged. After all the data have been processed, the program is to print the number of purchases in the run, the dollar amount of the total sales, and the average dollar amount per purchase.

By setting up an appropriate structure, we can make the overall program quite simple. One way to do this is to use the product type as a guide in determining how the input is to be (re)read, what kinds of calculations are required, and how the corresponding output line should look. If we do that, the main program becomes little more than a loop which "sees to it" that each record is read and processed properly (based on its type), and that the appropriate output is printed. The details for each product type are hidden in the processing routines `REREAD` (which analyzes the input), `SLSCMP` (which prepares the output), and `SLSPRT` (which prints the appropriately formatted output line).

If we look more closely at the type descriptions in Table 16.2, we see that, regardless of product type, we must end up with a quantity of goods purchased which, when multiplied by a unit price, will tell us the dollar amount of that purchase. This quantity is a real number for types 4, 5, 6, and 8, and an integer for the others. Consequently, we can simplify the data handling by declaring a general variable `RLAMT` for the real usage and an integer `IGRAMT` to handle those cases where an integer value is required.

Since the first digit of the product type determines the kind of processing to be done with that set of input data, the main program will perform the simple task of extracting that digit (and storing it in a variable named `CATEG`) as part of its overall loop. Then, `CATEG` can serve as the "switch" that regulates the `CASE` structures for selecting the rereading format, preparation of the results, and delivery of the output line. With this in mind, we can construct the main program as shown in Figure 16.5(a). Note that, because it

**Table 16.2** Input Format Descriptions for Example 16.2.

| *columns* | *item* | *format* |
|---|---|---|
| | *type 1:* | |
| 21–24 | length, meters | integers |
| 27–30 | price per meter | XX ( . ) XX |
| | *type 2:* | |
| 21–23 | length, m | XX ( . ) X |
| 24–26 | width, m | XX ( . ) X |
| 28–30 | no. of pieces | integers |
| 36–40 | price per sq. m. | XXX ( . ) XX |
| | *type 3:* | |
| 21–23 | no. of sheets | integers |
| 26–30 | price per sheet | XXX ( . ) XX |
| | *type 4:* | |
| 21–23 | length, m | XX ( . ) X |
| 24–26 | width, m | XX ( . ) X |
| 27–29 | thickness, mm | XX ( . ) X |
| 31–33 | no. of pieces | integers |
| 36–40 | price per cu. m. | XXX ( . ) XX |
| | *type 5:* | |
| 21–24 | cubic meters | XXX ( . ) X |
| 26–30 | price per cu. m. | XXX ( . ) XX |
| | *type 6:* | |
| 21–25 | no. of liters | XXXX ( . ) X |
| 27–30 | price per liter | XX ( . ) XX |
| | *type 7:* | |
| 21–24 | no. of barrels | integers |
| 26–30 | price per barrel | XXX ( . ) XX |
| | *type 8:* | |
| 21–25 | no. of kgms | XXXX ( . ) X |
| 36–40 | price per kgm | XXX ( . ) XX |

involves only a single statement, we have chosen to perform the initial input (i.e., from the external unit) as part of the main program rather than constructing it as a separate subprogram. The other major processing duties are assigned to several subprograms as follows:

1. INIT—This subroutine "primes" the program by initializing the counters for the total number of purchases (NUMPCH) and the total purchase amount (GRAND). In addition, it invokes the subroutine NEWPG which initializes the line counter (LINES) and prints the column headings.
2. REREAD—This subroutine, using CATEG as an indicator, reads the appropriate variables from the internal file NEWIN.
3. SLSCMP—This subroutine, also motivated by CATEG, computes the purchase amount (TTLPUR) for the current set of input data.
4. SLSPRT—This subroutine prints the output line for the current set of input data. As is true for REREAD and SLSCMP, the contents and format of the line are dictated by the value in CATEG, i.e., by the product type.

The individual routines are shown in Figures 16.6(b) through 16.6(f). Note that the actual statements for NEWPG and SLSPRT are not given. You will be asked to supply them as part of the problems at the end of the chapter. A possible output format is shown in Figure 16.6(g).

**16.4.3 Another Technique for Using Variable Formats** HP FORTRAN 77 provides another approach for choosing among a set of predefined format specifications: It is

```
C*************************************************************
C                  EXAMPLE 16.2 - THE MAIN PROGRAM            *
C*************************************************************
C  THIS PROGRAM PROCESSES PURCHASE DATA. THE INPUT DATA FORMAT *
C  FOR AN INDIVIDUAL PURCHASE DEPNDS ON WHICH ONE OF 8 POSSIBLE*
C  PRODUCT TYPES IS BEING PURCHASED. THE VARIABLES ARE:       *
C    LENGTH, WIDTH, THK: - THESE ARE LENGTH, WIDTH, THICKNESS *
C                  DIMENSIONS REPORTED FOR SOME PRODUCT TYPES. *
C    RLAMT: - THIS REPRESENTS THE QUANTITY BEING PURCHASED WHEN*
C             THAT QUANTITY IS A REAL NUMBER (LENGTH, SQUARE  *
C             METERS, CUBIC METERS, LITERS, OR KILGRAMS).     *
C    IGRAMT: - THE QUANTITY BEING PURCHASED WHEN THE QTY IS AN *
C         INTEGER (NO. OF SHEETS, NO. OF PCS, NO. OF BARRELS). *
C    UNITPR: - THE UNIT PRICE FOR A PARTICULAR PURCHASE.       *
C    TTLPUR: - THE UNIT PRICE TIMES THE PURCHASE AMOUNT.       *
C    GRAND: - THE SUM OF ALL THE TTLPUR VALUES IN THE RUN.     *
C    NUMPUR: - THE NUMBER OF PURCHASES FOR THE RUN.            *
C    AVGPUR: - THE AVERAGE PURCHASE AMOUNT (GRAND/NUMPUR).     *
C    TYPE, CATEG: - THE PRODUCT TYPE, AND ITS LEFTMOST DIGIT.  *
C    CUSTID: - THE CUSTOMER'S IDENTIFICATION NUMBER.           *
C    MO, DAY, YR: - THE PURCHASE DATE.                         *
C    LINES: - THE LINE COUNTER (40 LINES OF OUTPUT PER PAGE).  *
C    SLSPRS: - A 3-CHARACTER SALESPERSON'S IDENTIFIER.         *
C    NEWIN: - THE INTERNAL FILE CONTAINING COLUMNS 19-67 OF THE*
C             MOST RECENTLY READ INPUT LINE.                   *
C*************************************************************
      PROGRAM      EX1602
      IMPLICIT     NONE
      REAL         LENGTH,WIDTH,THK,UNITPR,TTLPR,GRAND,AVGPUR,RLAMT
      INTEGER*2    TYPE,CATEG,CUSTID,MO,DAY,YR,LINES,NUMPUR,IGRAMT
      CHARACTER    SLSPRS*3,NEWIN*49
      CALL INIT (LINES,NUMPUR,GRAND)
      READ (*,'2I6,3I2,A49,A3',END=99) TYPE,CUSTID,MO,DAY,YR,
     1                                 NEWIN,SLSPRS

      DO WHILE (.TRUE.)   !AN INFINITE LOOP BROKEN BY END-OF-FILE!
         IF (LINES .EQ. 40) CALL NEWPG (LINES)
         CATEG = TYPE/100000
         CALL REREAD (CATEG,NEWIN,LENGTH,WIDTH,THK,RLAMT,
     1                IGRAMT,UNITPR)
         CALL SLSCMP (CATEG,LENGTH,WIDTH,THK,RLAMT,IGRAMT,UNITPR,
     1               TTLPUR,GRAND,NUMPUR)
         CALL SLSPRT (CATEG,TYPE,CUSTID,MO,DAY,YR,LENGTH,WIDTH,THK,
     1               RLAMT,IGRAMT,UNITPR,TTLPUR,SLSPRS,LINES)
         READ (*,'2I6,3I2,A49,A3',END=99) TYPE,CUSTID,MO,DAY,YR,
     1                                 NEWIN,SLSPRS
      END DO

      AVGPUR = GRAND/NUMPUR
      WRITE (*,199) NUMPUR,GRAND,AVGPUR
  199 FORMAT (///1X,10X,'NUMBER OF PURCHASES: ',I4/1X,10X,
     1        'TOTAL SALES AMOUNT: ',F9.2/1X,10X,
     2        'AVERAGE AMOUNT PER PURCHASE: ',F8.2)
      STOP
      END
```

*(Continued)*

**FIGURE 16.6 (a)** Main Program for Example 16.2.

```
C*****************************************************************
C                              INIT                             *
C*****************************************************************
C  THIS LITTLE SUBROUTINE STARTS THE RUN BY INITIALIZING NUMPUR*
C  AND GRAND, THE TWO CUMULATIVE TOTALS. INDIRECLY, IT ALSO    *
C  INITIALIZES LINES, THE LINE COUNTER, BY CALLING NEWPG.      *
C*****************************************************************
      SUBROUTINE          INIT (LNS,NUM,GR)
      IMPLICIT            NONE
      REAL                GR
      INTEGER*2           LNS,NUM

      NUMPUR = 0
      GR = 0.0
      CALL NEWPG (LNS)
      RETURN
      END
```

**(b)**

```
C*****************************************************************
C                             REREAD                            *
C*****************************************************************
C  AFTER THE MAIN PROGRAM HAS READ A LINE FROM THE EXTERNAL    *
C  UNIT AND EXTRACTED THE PART CATEGORY (CATEG), REREAD USES   *
C  CATEG'S VALUE TO SELECT THE PROPER INPUT FORMAT AND READS   *
C  THE APPROPRIATE SET OF VARIABLES FROM THE INTERNAL FILE     *
C  (NEWIN) INTO WHICH THE ENTIRE SEQUENCE OF CHARACTERS WERE   *
C  STORED. THE ROUTINE IS BASED ON THE ASSUMPTION THAT CATEG   *
C  HAS A VALUE OF 1, 2, 3, 4, 5, 6, 7, OR 8.                   *
C  REMEMBER THAT THE FIRST POSITION (CHARACTER) IN NEWIN       *
C  CORRESPONDS TO POSITION 19 IN THE ORIGINAL RECORD.          *
C*****************************************************************
      SUBROUTINE          REREAD (CAT,CHR,L,W,T,RL,IL,U)
      IMPLICIT            NONE
      REAL                L,W,T,RL,U
      INTEGER*2           CAT,IL
      CHARACTER*49        CHR

      GO TO (21,22,23,24,25,26,27,28),CAT
   21 READ (CHR,31) IL,U
   31 FORMAT (2X,I4,2X,F4.2)
      GO TO 88
   22 READ (CHR,32) L,W,IL,U
   32 FORMAT (2X,2F3.1,1X,I3,5X,F5.2)
      GO TO 88
   23 READ (CHR,33) IL,U
   33 FORMAT (2X,I2,2X,F5.2)
      GO TO 88
   24 READ (CHR,34) L,W,T,IL,U
   34 FORMAT (2X,3F3.1,1X,I3,2X,F5.2)
      GO TO 88
   25 READ (CHR,35) RL,U
   35 FORMAT (2X,F4.1,1X,F5.2)
      GO TO 88
   26 READ (CHR,36) RL,U
   36 FORMAT (2X,F5.1,1X,F4.2)
      GO TO 88
   27 READ (CHR,37) IL,U
   37 FORMAT (2X,I4,1X,F5.2)
      GO TO 88
   28 READ (CHR,38) RL,U
   38 FORMAT (2X,F5.1,F5.2)
   88 CONTINUE

      RETURN
      END
```

**(c)**

**FIGURE 16.6** **(b)** INIT Subroutine for Example 16.2. **(c)** REREAD Subroutine for Example 16.2.

```
C*****************************************************************
C                          SLSCMP                                *
C*****************************************************************
C  THIS SUBROUTINE USES CATEG TO SELECT THE WAY IN WHICH IT      *
C  OBTAINS TTLPUR, THE DOLLAR AMOUNT FOR THE CURRENT PURCHASE.   *
C  IN SOME CASES, THE PURCHASE QUANTITY IS AVAILABLE DIRECTLY AS*
C  AN INPUT VALUE; IN OTHERS, IT HAS TO BE COMPUTED. SINCE THE   *
C  ACTIONS ARE IDENTICAL FOR MORE THAN ONE PRODUCT TYPE, THE     *
C  PROCESSING TAKES ADVANTAGE OF THIS AND THE CASE STRUCTURE     *
C  SENDS THE ROUTINE TO THE SAME DESTINATION FOR MORE THAN ONE   *
C  PRODUCT TYPE.                                                 *
C*****************************************************************
      SUBROUTINE          SLSCMP (CAT,L,W,T,RL,IL,U,TTL,GR,N)
      IMPLICIT            NONE
      REAL                L,W,T,RL,U,TTL,GR
      INTEGER*2           CAT,IL,N

      N = N + 1
      GO TO (41,42,41,44,45,45,41,45),CAT
   41 TTL = IL * U
      GO TO 188
   42 TTL = IL * U * L * W
      GO TO 188
   44 TTL = IL * U * L * W * 0.001*T
      GO TO 188
   45 TTL = RL * U
  188 CONTINUE

      GR = GR + TTL
      RETURN
      END
```

**(d)**

```
C*****************************************************************
C                          NEWPG                                 *
C*****************************************************************
C  THIS SUBROUTINE STARTS A NEW OUTPUT PAGE BY INITIALIZING OR   *
C  RESETTING THE LINE COUNTER TO ZERO AND BY PRINTING A SET OF   *
C  COLUMN HEADINGS.                                              *
C*****************************************************************
```

**(e)**

```
C*****************************************************************
C                          SLSPRT                                *
C*****************************************************************
C  THIS SUBROUTINE PRINTS A LINE OF OUTPUT USING THE INPUT AND   *
C  THE RESULTS DEVELOPED BY SLSCMP. IN ADDITION, IT SUPPLIES A   *
C  CHARACTER STRING FOR THE ENTRY IN THE COLUMN HEADED "UNITS".*
C  THIS VALUE (EXPRESSED EITHER AS METERS, SQ MTRS, CU MTRS,     *
C  LITERS, BARRELS, OR KGMS) IS BASED ON THE PRODUCT TYPE. FOR   *
C  SOME PRODUCT TYPES, IT IS AVAILABLE DIRECTLY AS PART OF THE   *
C  INPUT DATA. FOR OTHERS, IT MUST BE COMPUTED (E.G., NUMBER OF*
C  PIECES X THE AREA OF A PIECE).                                *
C*****************************************************************
```

**(f)**

**FIGURE 16.6** **d)** SLSCMP Subroutine for Example 16.2. **(e)** Description of NEWPG Subroutine for Example 16.2. **(f)** Description of SLSPRT Subroutine for Example 16.2.

possible to associate a name with a statement label and then refer to that name (instead of the statement number) in a READ and WRITE statement. The association is established by an ASSIGN statement. Suppose FMT_LABEL is declared as in INTEGER*2 variable. Then, the statement

```
ASSIGN 16 TO FMT_LABEL
```

associates the value 16 with the variable FMT_LABEL. HP FORTRAN 77 expects 16 to be a statement label somewhere in the program. If we arrange for it to be the label of a FORMAT statement, we can say

```
READ (*, FMT_LABEL)  inputlist
```

and the program will read the input list as directed by the format specifications in statement 16. If we wanted to choose among three different input formats described, respectively, in statements 16, 26, and 36, we could do that as follows:

```
      .......
      IF  condition1  THEN
        ASSIGN 16 TO FMT_LABEL
      ELSE IF  condition2  THEN
             ASSIGN 26 TO FMT_LABEL
         ELSE
           ASSIGN 36 TO FMT_LABEL
      END IF
      READ (*, FMT_LABEL)  inputlist
      .......
16    FORMAT  (description1)
26    FORMAT  (description2)
36    FORMAT  (description3)
      .......
```

**16.5 SUMMARY** FORTRAN provides a variety of additional format control mechanisms that make it possible to set up and use more complex and versatile systems for edit-directed data.

1. The G-specification (G*w*. *d*) enables the programmer to describe real numerical values whose appearance will depend on their magnitude. When FORTRAN is given a G-specification, it tries to edit the value so that it looks like an F-specified value. If this cannot be done (because the value is too large or too small for the *w* or *d* that was specified for it), FORTRAN automatically produces the value in E-format.

2. The T-, TL-, and TR-specifications make it possible to treat the edit-directed input or output record as if it were a line on a typewriter, with the specifications acting like versatile tabulator keys.

   (1) The plain T-specification causes FORTRAN to move to a specified position in the record.

   (2) The TL-specification causes FORTRAN to move to the left (i.e., backward) in the record.

   (3) The TR-specification causes FORTRAN to move to the right (i.e., forward) in the record.

3. The P-specification allows numerical values to be scaled (i.e., multiplied by a power of ten) so that their apparent value can be different from the actual value (i.e., the value stored internally). If *k* (a signed integer constant) is the scale factor, then

   apparent (scaled) value = internal value * 10 ** *k*

4. Additional features and techniques allow the FORTRAN programmer to delay the precise specification of a data format until the information actually is used by the program as it runs:

   (1) A *run-time* format can be supplied as an ordinary input character string. Then, when the input data are to be read (or the output data are to be written), the name of that character string is used in the READ or WRITE statement instead of a FORMAT statement number. Consequently, FORTRAN is instructed to look at a variable (which, of course, may change) instead of a program statement (which may not, once the program is compiled).

   (2) Input data can be read into an *internal file*. The programmer then can pretend that the resulting record is in another "input device" and "reread" it, using a different format description, and even a different READ statement with a different input list.

   (3) One of several predefined format descriptions can be selected at run time by specifying a variable statement label in the READ or WRITE statement and using the ASSIGN statement to associate the appropriate FORMAT statement number with the integer label variable.

## PROBLEMS

1. Assume that each number given below is the value of a variable about to be printed. Show what it will look like when edited according to the format specification that appears with it:

| *numerical value* | *format specification* | *numerical value* | *format specification* |
|---|---|---|---|
| (a) 23.0 | F14.6 | (b) 23.0 | G14.6 |
| (c) 0.09724 | G14.6 | (d) 0.09724 | G14.5 |
| (e) 0.09724 | G14.4 | (f) −3.00728 | E14.7 |
| (g) −3.00728 | G14.7 | (h) 7863.414 | G14.7 |
| (i) 7863.4148 | G14.3 | (j) −0.0000919 | G10.3 |
| (k) 0.32655E+00 | G11.4 | (l) 554227.0 | G11.4 |
| (m) −0.11748E−04 | 2PE12.5 | (n) −0.11748E−04 | −2PE12.5 |
| (o) −0.11748E−04 | 3PF12.5 | (p) 425678000000.0 | −6PF13.6 |
| (q) −887565000.0 | −3PF7.1 | | |

2. The following variables are in storage:

   PRTS has a value of 0.0934
   CONC has a value of 0.000000087 lbs of B per lb of mixture
   VOL has a value of 327484 cubic meters
   REVN has a value of 289764000 dollars

   We would like to print these values on a single output line as follows:

   PRTS in Cols. 11–15, expressed as a percent, to the nearest hundredth of a percent
   VOL in Cols. 21–28, expressed as thousands of cubic meters, with all six significant figures included
   CONC in Cols. 31–35, expressed in parts per million (lbs of B per million lbs of mixture), with all significant figures included
   REVN in Cols. 51–60, expressed as hundreds of thousands of dollars, with all significant figures included

   Write a FORMAT statement that will meet these requirements.

3. Rewrite the FORMAT statements in each of the following problems from Chapter 15 so that they produce exactly the same results with T-specifications used instead of the X-specifications. (Note that you are not to use TR- or TL- specifications; use just the ordinary T-specification:

| | | | |
|---|---|---|---|
| (a) Problem 1(a) | (b) Problem 1(c) | (c) Problem 1(d) | (d) Problem 1(f) |
| (e) Problem 1(i) | (f) Problem 2(b) | (g) Problem 2(d) | (h) Problem 2(e) |
| (i) Problem 3(a) | (j) Problem 3(d) | (k) Problem 4(c) | (l) Problem 4(f) |
| (m) Problem 5(b) | (n) Problem 5(d) | (o) Problem 6(c) | (p) Problem 6(e) |

4. Repeat Problem 3 using the TR-specification in place of each X-specification for the indicated FORMAT statements.
5. Rewrite the NEWPG subroutine for Example 15.1 using G-specifications for the numerical output and T- or TR-specifications for positioning.
6. Rewrite the TABPRT subroutine for Example 15.1 using G-specifications for the numerical output and T- or TR-specifications for positioning.
7. Rewrite the READER subroutine for Example 15.2 using G-specifications for the numerical output and T- or TR-specifications for positioning.
8. Rewrite the NEWPG subroutine for Example 15.2 using G-specifications for the numerical output and T- or TR-specifications for positioning. (Note that the G-specification may not be applicable for *all* the numerical output.)
9. Rewrite the PAYPRT subroutine for Example 15.2 making the substitutions described in the previous problem.
10. Using the information given in the example, write the NEWPG and SLSPRT subroutines for Example 16.2.
11. Claws and Talons, Ltd. is a conglomerate that has acquired a large number of companies, each of which is engaged in selling a certain line of products. Each of these companies reports its individual sales by recording the following information on one line for each sale:

| | |
|---|---|
| product code: | three letters followed by three digits |
| date of sale: | mm/dd/yy for some companies; mmddyy for others |
| no. of items: | a five-digit integer for some companies; four digits for others |
| unit price: | a real number (in the form xxxxx. )xx; the decimal point is not recorded |
| type of payment: | a single digit (1=cash; 2=on account |
| method of shipment: | a single digit (1=taken by customer; 2=truck; 3=rail; 4=air; 5=boat or barge; 6=oxcart; 7=dogsled; 8=other) |

Since each company was independent of all the others before Claws and Talons swallowed it, it had its own way of recording this information. Thus, no two companies' input formats are alike. In fact, the order in which the information is recorded varies from company to company. However, in order to make it possible to handle all the data together without forcing each company to change its format, C and T took advantage of the fact that no company used columns 78–80 for anything. How nice. Consequently, what they did was to assign a 3-digit company code to each organization and required the companies to record their respective codes in those columns of each sales record. Thus, wherever else the other input values may be, one could always look in columns 78–80 and find the proper company code there.

The parent company would like to produce a weekly report in which each line of output shows the data for an individual sale: Company code, an echo of the input values (with English names for type of payment and method of shipment instead of numerical codes), and the total amount of the sale. On a separate page, the program is to report the total number of sales, the total amount of the sales, the total amount of cash sales, and the total amount of credit (on account) sales.

Based on the (incomplete) information given above, design an appropriate program and prepare a detailed pseudocode or N-S representation.

12. Claws and Talons (from Problem 11) currently owns six companies. The company codes and input formats are shown below:

| | *Company Code* | *Columns* | *Format* |
|---|---|---|---|
| Product Code | 072 | 1–6 | See Problem 11 |
| | 041 | 3–8 | See Problem 11 |
| | 741 | 11–16 | See Problem 11 |
| | 760 | 4–9 | See Problem 11 |
| | 108 | 41–46 | See Problem 11 |
| | 359 | 17–22 | See Problem 11 |
| Date of Sale | 072 | 11–18 | mm/dd/yy |
| | 041 | 9–14 | mmddyy |
| | 741 | 1–6 | mmddyy |
| | 760 | 11–18 | mm/dd/yy |
| | 108 | 1–8 | mm/dd/yy |
| | 359 | 5–10 | mmddyy |

| | | | |
|---|---|---|---|
| No. of Items | 072 | 21–25 | Integers |
| | 041 | 21–24 | Integers |
| | 741 | 18–21 | Integers |
| | 760 | 31–35 | Integers |
| | 108 | 11–14 | Integers |
| | 359 | 1–4 | Integers |
| Unit Price | 072 | 31–37 | See Problem 11 |
| | 041 | 26–32 | See Problem 11 |
| | 741 | 24–30 | See Problem 11 |
| | 760 | 21–27 | See Problem 11 |
| | 108 | 20–26 | See Problem 11 |
| | 359 | 34–40 | See Problem 11 |
| Type of Payment | 072 | 30 | See Problem 11 |
| | 041 | 1 | See Problem 11 |
| | 741 | 51 | See Problem 11 |
| | 760 | 56 | See Problem 11 |
| | 108 | 9 | See Problem 11 |
| | 359 | 71 | See Problem 11 |
| Method of Shipment | 072 | 31 | See Problem 11 |
| | 041 | 2 | See Problem 11 |
| | 741 | 52 | See Problem 11 |
| | 760 | 57 | See Problem 11 |
| | 108 | 10 | See Problem 11 |
| | 359 | 72 | See Problem 11 |

Using this information, together with the design prepared for Problem 11, write a complete program that meets the processing requirements given in Problem 11. Note that even though the input formats are different for each company, their echo must follow a consistent format; otherwise, it would be impossibly difficult to read the printout properly.

13. Modify the program for Example 16.1 so that it will produce any number of frequency distribution tables for different input data collections with different formats.

14. Modify the program for Problem 13 so that it skips an entire set of input data if there is something wrong with the control information, or if the information is missing.

15. Modify the program for Example 16.1 (or for Problem 13 or 14) so that it will recognize blanks and treat them as missing data. This means that the total number of observations for a given variable may be different from that of any other variable. Consequently, the "grand total" (see Figure 16.3(e)) no longer makes any sense. Instead, modify the output format by adding two more columns to it: One of these is to show the number of missing values, and the other is to show the total number of observations for that variable. Then, after the table has been printed, the revised program is to print the total number of input records, and the total number of missing values (for all the variables).

16. Modify the program in Problem 15 so that, in addition to recognizing and dealing with missing data, it also handles errors in the data. (An error is a value other than 0 through 9 or blank.) Add another column to the output showing the number of erroneous (i.e., mistyped) values for each variable. Then, print the total number of errors along with the total number of input records and the total number of missing values.

17. Modify the program in Example 16.2 so that the appropriate format is selected by means of the ASSIGN statement technique described in Section 16.4.3.

# 17
# *Additional Input/Output Techniques*

In this chapter we shall examine some additional input/output processes that provide greater flexibility for certain types of applications. Once the characteristics of these processes have been established, we shall discuss appropriate FORTRAN features that support them.

## 17.1 UNFORMATTED FILES AND RECORDS

The discussions in the previous two chapters make it clear that HP FORTRAN 77 provides the programmer with an elaborate collection of capabilities for interpreting and formatting edit-directed data. By now, we can write a set of format specifications that will describe just about anything we want to describe. However, there are many occasions when all of these convenient and powerful capabilities are wasted. The reason lies in the simple fact that not all data are produced for humans to examine. (There are related problems in that much of the information that *is* intended for human inspection does not get looked at, but that is another matter.)

In Section 1.1.3 of Chapter 13 we discussed a type of situation in which a process consists of several programs such that the output produced by one program may serve as input to one or more others. Often, it is only the information at the beginning and end of this overall process that is of interest to humans. The initial input usually is prepared by typing it into a terminal of some kind. Consequently, it needs to be in human compatible form so that it can be read and checked for proper entry. Similarly, the usefulness of final output often depends on how easily the human user can read and understand it. Thus, it is worthwhile for the programmer to put in the work and care required for the preparation of convenient format designs. The same holds true for the extra computer time that might be involved in interpreting the format specifications and performing the appropriate editing processes.

When a human observer is not involved, FORTRAN makes it possible to avoid this additional overhead by enabling the programmer to set up and use *unformatted files*. Recall (from Chapter 13) that the data stored in unformatted file records are an exact physical copy of what was in the processor's main memory. Since the machine is designed specifically to handle data in that form, there is no need for FORTRAN to step in and provide translating services.

### 17.1.1 Preparation of Unformatted Files

The typical process for preparing unformatted files starts with human-compatible input. (This is not always the case. There are situations where initial input to a computing process consists of electrical signals produced by something connected directly to the processor. Such signals may come from a variety of sources ranging from a temperature probe on a spacecraft to a set of electrodes picking up a human heartbeat. However, we shall not be considering such systems here.) The input, usually edit-directed, is read and

processed to produce an (unformatted) output file. As part of this process, of course, it may be perfectly reasonable to produce human-readable output (e.g., a printed report) as well, but our focus here is on the machine-compatible file that will be submitted for further processing. (A glance at Figure 13.4 will provide a helpful reminder of this overall process.)

Since the standard systems input and output units usually are associated with formatted data, it is necessary to create an unformatted file by explicit program specifications in which the file is described and associated with a particular unit. A convenient way to do this is by means of the OPEN statement (Section 13.2.3.1). This establishes the file's existence (even before any records are produced for it), connects the file to a specified unit, and establishes the necessary bookkeeping to make the file available for use.

As an example, suppose we wanted to create a file named CKSUBS and connect it to unit 2. The appropriate OPEN statement would look like this:

```
      OPEN  (UNIT=2,FILE='CKSUBS',FORM='UNFORMATTED',
     1        ACCESS='SEQUENTIAL',STATUS='NEW')
```

In this particular instance, FORTRAN would have opened the file for sequential access anyway (by default). Similarly, another default mechanism would have assigned a status of 'NEW' if the STATUS= specification had not appeared. However, the practice of specifying everything is as desirable here as it is in other types of declarations. Inclusion of the file name (CKSUBS in this example) activates a mechanism that incorporates the name as part of the physical data recorded in the file. Then, when those data are to be read later on (either in the same program or in an entirely different one), the file can be identified by name and the system will be able to match that name with information in the file.

All of this happens automatically as part of the services provided by the HP operating system in which FORTRAN programs operate. Consequently, the programmer is provided with an additional safety measure: If a program uses a named file, and that file's data are recorded on tape or magnetic disk, the program will not run unless the disk or tape used with that program includes the same file name as part of its recorded information. Thus, in our example, suppose that unit 2 is associated with a magnetic tape device and we write a series of records onto a reel of tape installed on that device. Information about the file name will be included automatically as part of that recorded output. Then, suppose we put the tape away and use it as input for another program three days later. If that program is designed to read from a file named CKSUBS, that program will not be able to run unless it "assures" itself that the reel of tape submitted as input contains a file named CKSUBS.

### 17.1.2 Transmission of Unformatted Data Records

Since an unformatted record (by its nature) requires no editing or interpretation, its transmission requires nothing more than a specification of the unit number and an input or output list. For example, the statement

```
READ (2) A,XFR,STAT,BIGW
```

reads the next record from unit 2 and stores the four items in that record in variables A, XFR, STAT, and BIGW, respectively. The operation will not work if there are inconsistencies between the information in the statement and the characteristics of the file associated with the unit specified in that statement. Specifically, the programmer must make sure that:

1. The file connected to the specified unit (2 in this case) is an unformatted file.
2. There are *at least* as many items in the unformatted record as there are in the input list. Since each input or output operation transmits exactly one record, additional items in the record (over and above those specified in the input or output list) are ignored.

3. The type of each data item in the record must match that declared for the item in the corresponding position in the input list. For example, if STAT is declared as a REAL variable, then the third item in the record must be a real number. Similarly, if BIGW were declared as CHARACTER*17, the fourth item in the record must be a 17-character string.

The programmer can take appropriate precautions when the program is designed by making sure that the organization of the file to be used is completely understood so that it is reflected accurately in the program's statements. Further checking can be done just before the program is run to make sure that the proper file is used. While these measures are important, it may be necessary to provide additional insurance by building it into the program itself.

To begin with, we are already familiar with the END= and ERR= specifiers, either or both of which work here exactly as they do with formatted input or output statements. Thus,

```
READ  (2, END=99, ERR=88) A, XFR, STAT, BIGW
```

sends the program to statement 99 if the endfile record is encountered or to statement 88 if something goes wrong during the input operation.

Another way to provide the same type of service is by means of the IOSTAT= specifier (Section 13.2.1.1). For instance, we could say

```
READ  (2, IOSTAT=HOW) A, XFR, STAT, BIGW
```

where HOW is the name of a variable declared as INTEGER*2. Then, instead of transferring to other statements as the result of an error or endfile condition, a statement can be placed immediately after the READ statement to test the variable HOW to determine the outcome of the input attempt. Since a normal transmission (no error or endfile condition) produces a zero in the IOSTAT variable, we shall be optimists and use that as the default in a CASE construction as follows:

```
REAL              A, STAT
INTEGER*2         XFR, HOW
CHARACTER*17      BIGW
      .......
      .......
READ  (2, IOSTAT=HOW) A, XFR, STAT, BIGW
IF (HOW . GT. 0) THEN
   CALL ERRCMP (arguments)
ELSE IF (HOW . LT. 0) THEN
        CALL EOFCMP (arguments)
     ELSE
END IF
normal processing
   .......
   .......
```

Facilities for further investigation can be built into the program by means of the INQUIRE statement outlined in Section 13.2.3.3. For example, the statements in Figure 17.1. open a file connected to unit 2, check to determine whether the file just opened is an unformatted file named CKSUBS, read a record from that file, and check to determine whether the input operation proceeded normally. In this type of construction, ERRCMP is a general error-handling subroutine that checks all of IOERR's elements to see which kinds of errors were encountered. This enables it to respond to each type of error with an appropriate action. (The IF statement that determines whether or not ERRCMP should be called uses the sum of IOERR's elements to establish whether they all have values of zero;

```
C*************************************************************
C  IOERR IS AN ARRAY OF INTEGERS, EACH OF WHICH SIGNALS THE *
C  OCCURRENCE (1) OR NON-OCCURRENCE (0) OF A PARTICULAR TYPE*
C  OF ERROR:                                                *
C    IOERR(1) : REPORTS FILE EXISTENCE                      *
C    IOERR(2) : REPORTS PROPER FILE NAME                    *
C    IOERR(3) : REPORTS WHETHER FILE IS UNFORMATTED         *
C    IOERR(4) : REPORTS TYPE OF FILE ACCESS                 *
C    IOERR(5) : REPORTS AN ERROR DURING INPUT               *
C  IOERR IS INITIALIZED TO ZERO, AND THEN THE ELEMENTS ARE  *
C  CHANGED OR NOT, DEPENDING ON THE OUTCOME OF THE INQUIRE  *
C  OPERATION.                                               *
C*************************************************************

      ..........
      REAL           A,STAT
      INTEGER*2      XFR,HOW,I,IOERR(5)
      CHARACTER      BIGW*17,FILNAM*6,FRMTST*11,ACCTST*10
      LOGICAL        EXTEST
      ..........
      ..........
      DO I=1,5
        IOERR(I) = 0
      END DO
      OPEN (2,FILE='CKSUBS',FORM='UNFORMATTED')
      ..........
      ..........
      INQUIRE (UNIT=2,NAME=FILNAM,FORM=FRMTST,ACCESS=ACCTST,
     1         EXIST=EXTEST)
      IF (EXTEST .EQ. .FALSE.) IOERR(1) = 1
      IF (FILNAM .NE. 'CKSUBS') IOERR(2) = 1
      IF (FRMTST .NE. 'UNFORMATTED') IOERR(3) = 1
      IF (ACCTST .NE. 'SEQUENTIAL' ) IOERR(4) = 1
      ..........
      READ (2,IOSTAT=HOW)
      IF (HOW .GT. 0) IOERR(5) = 1
      IF (IOERR(1) + IOERR(2) + IOERR(3) + IOERR(4) + IOERR(5)
     1    .GT. 0) THEN
        CALL ERRCMP (IOERR)
      ELSE IF (HOW .LT. 0) THEN
             CALL EOFCMP (arguments)
           ELSE
      END IF
      normal processing
```

**FIGURE 17.1** File Checking with the INQUIRE Statement.

If IOERR had many more elements, this type of technique would be impractical and another approach would have to be devised.)

**Example 17.1** Heavy Facts, Inc. supplies its customers with tables of scientific data on request. Lately, they have been noticing an active demand for information about a certain group of chemicals. As a result, they have decided to make this part of the operation more efficient by building a computerized file

for these data, together with a set of supporting programs that will search the file and produce reports to fulfill customers' requirements.

**Table 17.1**

| *Columns* | *Item* | *Format* |
|---|---|---|
| 1–6 | chemical's i.d. | integers |
| 7–12 | molecular wt. | `xxxx ( . ) xx` |
| 13–16 | year first produced | integers |
| 17–36 | trade name | characters |
| 37–61 | empirical formula | characters |
| 62–64 | state at room temp. | `SLD`, `LIQ`, or `GAS` |
| 65–69 | melting pt., deg. C. | sign, followed by `xxx ( . ) x` |
| 70–75 | boiling pt., deg. C. | sign, followed by `xxxx ( . ) x` |
| 76 | bitterness index | 1, 2, 3, or 4 |

Data for each chemical are recorded on a separate line as shown in Table 17.1. In this example we shall develop the first support program, i.e., one that uses these lines as input (in order by chemical i.d. number) to produce an unformatted output file (named `CHDATA`) on unit 3. The program will print a line for each chemical record added to `CHDATA`. An addition test to make sure that the input records are in proper sequence. Specifically, the i.d. (columns 1–6) on each input record will be checked against that of the previous one. If the new i.d. number is not larger, the program will add that (new) record's information to another unformatted file named `CHMERR`, this one connected to unit 2. After all the data have been processed, a summary will be printed showing the total number of records read, the number accepted, (i.e., the number of chemical records in `CHDATA`), and the number of records out of order (i.e., the number of chemical records in `CHMERR`). Then, `CHMERR` will be repositioned to its first record and used as input for a printed error report showing the rejected data.

Pseudocode and N-S representations are shown in Figure 17.2 and the program itself is shown in Figure 17.3.

```
"Define THISID, PREVID, MWT, YR, TRADE, EMPRCL, STATE,
      MPT. BPT, BITTER, TTLNUM, NUMACC, NUMREJ."
"Initialize the program:
   Set TTLNUM, NUMACC, NUMREJ, PREVID to zero;
   Open CHDATA on unit 3 and CHMERR on unit 2;
   Read the first input card;                          } OPENER
   Write the column headings for the output report."
while there are more input data to process:
   if
      the current input card is out of order
   then
      "write the data on CHMERR;                       } ERRPRC
      increment NUMREJ."
   else
      "write the data on CHDATA;                       } ADDREC
      increment NUMACC;
      print an output line."
   endif
   "Read the next input card."                         RDCARD
endwhile
"Print the summary information."                       SUMPRT
"Print the error report."                              ERRPRT
"Stop."
```

**FIGURE 17.2 (a)** Pseudocode Representation for Example 17.1.

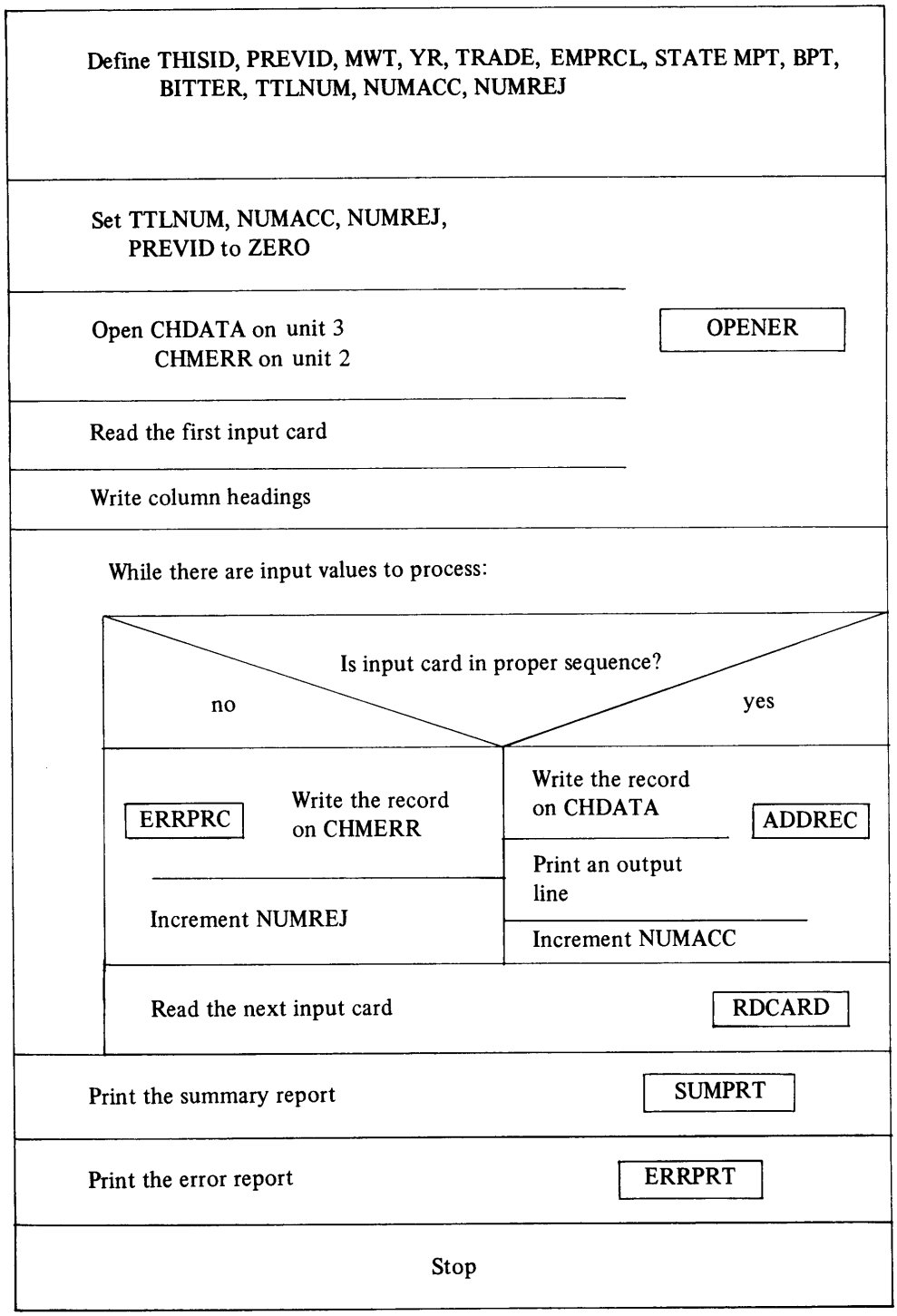



**FIGURE 17.2 (b)** N-S Representation for Example 17.1.

Once the file is built, we can process it in a variety of ways to extract information requested by Heavy Facts' customers. Just to illustrate the point, let us suppose that a customer wants to obtain a list of all compounds having a boiling point of at least so many degrees. For each compound that meets this requirement, the customer would like to see a line of print showing the compound's identification number, trade name, empirical formula, and (of course) the boiling point.

```
C*************************************************************
C                  EXAMPLE 17.1 - THE MAIN PROGRAM            *
C*************************************************************
C  THIS PROGRAM BUILDS AN UNFORMATTED FILE NAMED CHDATA BY PRO-*
C  CESSING A COLLECTION OF INPUT RECORDS, EACH CONTAINING DATA *
C  FOR A CHEMICAL COMPOUND. THE RECORDS ARE SUPPOSED TO BE IN  *
C  SEQUENTIAL ORDER BY INSCREASING I.D. NUMBER (COLUMNS 1-6).  *
C  WHEN THIS IS THE CASE, THE DATA ARE ADDED TO CHDATA ON UNIT *
C  3. LINES OUT OF ORDER ARE ADDED TO AN UNFORMATTED ERROR FILE*
C  (NAMED CHMERR) ON UNIT 2. EACH NEW CHDATA RECORD IS PRINTED *
C  ON A SEPARATE OUTPUT LINE, 40 LINES PER PAGE. SUMMARY DATA  *
C  ARE PRINTED AFTER ALL INPUT HAS BEEN PROCESSED, AND AN ERROR*
C  REPORT IS PRINTED AFTER THE SUMMARY. HERE ARE THE VARIABLES:*
C   THISID:  CURRENT I.D. NUMBER                              *
C   PREVID:  I.D. NUMBER OF PREVIOUS RECORD (INITIALIZED TO 0) *
C   MWT:     MOLECULAR WEIGHT                                 *
C   YR:      YEAR DISCOVERED OR INVENTED                      *
C   TRADE:   TRADE NAME                                       *
C   EMPRCL:  EMPIRICAL FORMULA (E.G., C17H3202S)              *
C   STATE:   PHYSICAL STATE AT ROOM TEMPERATURE (E.G., LIQ)   *
C   MPT:     MELTING POINT;     BPT:   BOILING POINT          *
C   BITTER:  BITTERNESS INDEX  (1, 2, 3, OR 4)                *
C   NUMACC, NUMREJ, TTLNUM: COUNTERS FOR NUMBER ACCEPTED, ETC. *
C   LINES:   A LINE COUNTER FOR PRINTED OUTPUT                *
C   EOF:     END-OF-FILE SIGNAL                               *
C*************************************************************
      PROGRAM    EX1701
      IMPLICIT   NONE
      REAL       MWT,MPT,BPT
      INTEGER*4  THISID,PREVID,TTLNUM,NUMACC,NUMREJ
      INTEGER*2  YR,BITTER,LINES
      CHARACTER  TRADE*20,EMPRCL*25,STATE*3
      LOGICAL    EOF

      CALL OPENER (TTLNUM,NUMACC,NUMREJ,PREVID,THISID,MWT,YR,TRADE,
     1             EMPRCL,STATE,MPT,BPT,BITTER,LINES,EOF)

      DO WHILE (.NOT. EOF)
         IF (THISID .LE. PREVID) THEN
           CALL ERRPRC (THISID,MWT,YR,TRADE,EMPRCL,STATE,MPT,BPT,BITTER)
           NUMREJ = NUMREJ + 1
         ELSE
           IF (LINES .EQ. 40) CALL NEWPG (LINES)
           CALL ADDREC (THISID,MWT,YR,TRADE,EMPRCL,STATE,MPT,BPT,
     1                  BITTER,LINES)
           NUMACC = NUMACC + 1
         END IF
         CALL RDCARD (PREVID,THISID,MWT,YR,TRADE,EMPRCL,STATE,MPT,BPT,
     1                BITTER,EOF,TTLNUM)
      END DO

      CALL SUMPRT (NUMACC,NUMREJ,TTLNUM)
      CALL ERRPRT (THISID,MWT,YR,TRADE,EMPRCL,STATE,MPT,BPT,BITTER)
      STOP
      END
```



**FIGURE 17.3 (a)** Main Program for Example 17.1.

```
C*****************************************************************
C                              OPENER                            *
C*****************************************************************
C  THIS ROUTINE OPENS TWO UNFORMATTED FILES: CHDATA ON UNIT 2,  *
C  AND CHMERR ON UNIT 3. IN ADDITION, OPENER INITIALIZES THE    *
C  COUNTERS TO ZERO, PRINTS A SET OF COLUMN HEADINGS (BY A CALL*
C  TO NEWPG), AND READS THE FIRST INPUT LINE BY CALLING RDCARD *
C*****************************************************************
      SUBROUTINE       OPENER (T,A,R,PR,TH,MW,Y,TR,EMP,ST,M,B,
     1                         BTR,L,EOF)
      IMPLICIT         NONE
      REAL             MW,M,B
      INTEGER*4        T,A,R,PR,TH
      INTEGER*2        Y,BTR,L
      CHARACTER        TR*20,EMP*25,ST*3
      LOGICAL          EOF

      OPEN (3,FILE='CHDATA',FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
      OPEN (2,FILE='CHMERR',FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
      T = 0
      A = 0
      R = 0
      TH = 0
      CALL NEWPG (L)
      CALL RDCARD (PR,TH,MW,Y,TR,EMP,ST,M,B,BTR,EOF,T)
      RETURN
      END
```

**(b)**

```
C*****************************************************************
C                              NEWPG                             *
C*****************************************************************
C  THIS ROUTINE STARTS A NEW PAGE OF OUTPUT BY WRITING A SET OF*
C  COLUMN HEADINGS AND (RE)INITIALIZING THE LINE COUNTER.       *
C*****************************************************************
```

**(c)**

```
C*****************************************************************
C                              RDCARD                            *
C*****************************************************************
C  THIS ROUTINE READS AN INPUT RECORD AND INCREMENTS A COUNTER  *
C  FOR EACH RECORD. IF THERE IS NO MORE INPUT, AN END-OF-FILE   *
C  INDICATOR IS SET TO .TRUE..                                  *
C*****************************************************************
      SUBROUTINE       RDCARD (PR,TH,MW,Y,TR,EMP,ST,M,B,BTR,EOF,T)
      IMPLICIT         NONE
      REAL             MW,M,B
      INTEGER*4        PR,TH,T
      INTEGER*2        Y,BTR
      CHARACTER        TR*20,EMP*25,ST*3
      LOGICAL          EOF

C---JUST PRIOR TO THE ACTUAL INPUT OPERATION, THE CURRENT CHEMICAL
C---IDENTIFIER (TH) IS COPIED INTO THE VARIABLE FOR THE PREVIOUS
C---IDENTIFIER (PR) SO THAT THE RECORDS CAN BE COMPARED FOR PROPER
C---SEQUENCING.

      PR = TH
      EOF = .FALSE.
      READ (*,15,END=99) TH,MW,Y,TR,EMP,ST,M,B,BTR
   15 FORMAT (I6,F6.2,I4,A20,A25,A3,F5.1,F6.1,I1)
      T = T+1
      GO TO 77
   99 EOF = .TRUE.
   77 RETURN
      END
```

**(d)**

*(Continued)*

**FIGURE 17.3** **(b)** Initialization Routine for Example 17.1. **(c)** NEWPG Routine for Example 17.1. **(d)** RDCARD Routine for Example 17.1.

```
C*****************************************************************
C                          ADDREC                                *
C*****************************************************************
C  THIS ROUTINE WRITES THE NEXT CHEMICAL RECORD ONTO THE FILE   *
C  CHDATA AND PRINTS THE INFORMATION AS WELL, INCREMENTING THE  *
C  LINE COUNTER AS PART OF THE PROCESS.                         *
C*****************************************************************
      SUBROUTINE       ADDREC (TH,MW,Y,TR,EMP,ST,M,B,BTR,L)
      IMPLICIT         NONE
      REAL             MW,M,B
      INTEGER*4        TH
      INTEGER*2        Y,BTR,L
      CHARACTER        TR*20,EMP*25,ST*3

      WRITE (2) TH,MW,Y,TR,EMP,ST,M,B,BTR
      CALL PRTREC (TH,MW,Y,TR,EMP,ST,M,B,BTR,L)
      RETURN
      END
```

(e)

```
C*****************************************************************
C                          PRTREC                                *
C*****************************************************************
C  THIS ROUTINE PRINTS A CHEMICAL RECORD ON A SINGLE OUTPUT     *
C  LINE AND INCREMENTS THE LINE COUNTER. THE ORDER IN WHICH THE*
C  INDIVIDUAL DATA ITEMS ARE PRINTED IS CONSISTENT WITH THAT OF*
C  THE COLUMN HEADINGS PRODUCED BY THE ROUTINE NEWPG.           *
C*****************************************************************
```

(f)

```
C*****************************************************************
C                          SUMPRT                                *
C*****************************************************************
C  THIS ROUTINE PRINTS THE SUMMARY INFORMATION AFTER THE OUTPUT*
C  FILE HAS BEEN PREPARED AND ITS RECORDS PRINTED.              *
C  SUMPRT ALSO WRITES ENDFILE RECORDS ON THE TWO OUTPT FILES.   *
C*****************************************************************
      SUBROUTINE       SUMPRT (A,R,T)
      IMPLICIT         NONE
      INTEGER*4        A,R,T

      WRITE (*,16)
   16 FORMAT ('1'//1X,33X,'SUMMARY REPORT:'//)
      WRITE (*,17) T,A,R
   17 FORMAT (1X,10X,'TOTAL NUMBER OF CHEMICALS READ: ',I6/
     1        1X,10X,'TOTAL NUMBER OF RECORDS ACCEPTED: ',I6/
     2        1X,10X,'TOTAL NUMBER OF RECORDS REJECTED:',I6)
      ENDFILE 2
      ENDFILE 3
      RETURN
      END
```

(g)

**FIGURE 17.3** **(e)** ADDREC Routine for Example 17.1. **(f)** PRTREC Routine for Example 17.1. **(g)** SUMPRT Routine for Example 17.1.

```
C*************************************************************
C                          ERRPRC                             *
C*************************************************************
C  THIS ROUTINE ADDS THE CURRENT CHEMICAL RECORD TO THE ERROR  *
C  FILE CHMERR. WHEN ALL THE INPUT RECORDS HAVE BEEN PROCESSED,*
C  ANOTHER ROUTINE (ERRPRT) WILL PRINT THE RECORDS FROM THIS   *
C  FILE.                                                       *
C*************************************************************
      SUBROUTINE       ERRPRC (TH,MW,Y,TR,EMP,ST,M,B,BTR)
      IMPLICIT         NONE
      REAL             MW,M,B
      INTEGER*4        TH
      INTEGER*2        Y,BTR
      CHARACTER        TR*20,EMP*25,ST*3

      WRITE (3) TH,MW,Y,TR,EMP,ST,M,B,BTR
      RETURN
      END
```

(h)

```
C*************************************************************
C                          ERRPRT                             *
C*************************************************************
C  THIS ROUTINE IS CALLED BY THE MAIN PROGRAM AFTER ALL THE    *
C  INPUT HAS BEEN READ AND PROCESSED AND THE SUMMARY REPORT HAS*
C  BEEN PRINTED. ERRPRT REWINDS UNIT 3 (I.E., REPOSITIONS THE  *
C  FILE TO ITS FIRST RECORD) AND THEN GOES THROUGH A CYCLIC    *
C  PROCESS (AFTER PRINTING A SET OF HEADINGS) IN WHICH EACH OF *
C  OF CHMERR'S RECORDS IS READ AND PRINTED.                    *
C*************************************************************
```

(i)

**FIGURE 17.3** **(h)** ERRPRC Routine for Example 17.1. **(i)** ERRPRT Routine for Example 17.1.

The required program, then, can be simple: After opening the file CHDATA (remember that the file already exists; this program needs only to connect it to a unit), the program reads an input record containing the customer's identification and the minimum required boiling point. Then, it reads and examines each of CHDATA's records, selecting and printing the information from those showing boiling points in the required range. Since the details of printing and formatting are of secondary interest here, Figure 17.4 shows only the main program.

## 17.2 DIRECT FILES

All of the file processing we have done so far has been with sequential files. As a result, each input or output operation has been limited to the particular record at which the file happened to be positioned. The kinds of processes in which we have used these files have not suffered seriously from this restriction.

However, there are numerous situations in which this limitation can make computer usage impractical. The process in the previous example (17.1) is a case in point. Having built the file CHDATA (as Figure 17.3 indicates), let us suppose now that we are interested in extracting certain information on demand. For instance let us assume that each compound's identification number (THISID) is nothing more than a sequence number, and that the file CHDATA was built with the records in order by increasing value of THISID with no records missing or out of sequence. In other words, the first input record's THISID value was 000001, the second one had a THISID value of 000002, and so on. Correspondingly, then, the records in the resulting output file also would be in consecutive order.

```
C*****************************************************************
C                        BPFIND - THE MAIN PROGRAM                *
C*****************************************************************
C  THIS PROGRAM SEARCHES THE UNFORMATTED FILE CHDATA (BUILT BY *
C  EXAMPLE 17.1) IN RESPONSE TO A CUSTOMER'S REQUEST TO FIND   *
C  AND PRINT THE COMPOUND IDENTIFICATION NUMBERS, TRADE NAMES, *
C  EMPIRICAL FORMULAS, AND BOILING POINTS FOR ALL COMPOUNDS    *
C  WHOSE BOILING TEMPERATURES ARE EQUAL TO OR GREATER THAN A   *
C  VALUE SPECIFIED BY THE INPUT.                               *
C*****************************************************************
C  THISID:  A COMPOUND'S IDENTIFICATION NUMBER                  *
C  MWT:     MOLECULAR WEIGHT                                   *
C  YR:      YEAR FIRST DISCOVERED OR PRODUCED                  *
C  TRADE:   A COMPOUND'S TRADE NAME                            *
C  EMPRCL:  EMPIRICAL FORMULA                                  *
C  STATE:   PHYSICAL STATE AT ROOM TEMPERATURE                 *
C  MP,BP:   MELTING PT., BOILING PT., BOTH IN DEGREES C.       *
C  BITTER:  BITTERNESS INDEX                                   *
C  CUSTID:  CUSTOMER'S IDENTIFICATION (CHARACTERS) COLS 1-6    *
C  BOIL:    REQUIRED MINIMUM BOILING POINT (COLS 11-16)        *
C  LINE:    A LINE COUNTER                                     *
C  NEWPG:   A ROUTINE (NOT SHOWN) FOR PRINTING PAGE HEADINGS   *
C  EXPRNT:  A ROUTINE (NOT SHOWN) TO PRINT AN OUTPUT LINE      *
C*****************************************************************
      PROGRAM         BPFIND
      IMPLICIT        NONE
      REAL            MWT,MP,BP,BOIL
      INTEGER*4       THISID
      INTEGER*2       YR,BITTER,LINE
      CHARACTER       CUSTID*6,TRADE*20,EMPRCL*25,STATE*3

      OPEN (4,FILE='CHDATA',STATUS='OLD',FORM='UNFORMATTED')
      READ (*,15) CUSTID,BOIL
   15 FORMAT (A6,4X,F6.1)
      CALL NEWPG (LINES)

      DO WHILE (.TRUE.)
         READ (4,END=99) THISID,MWT,YR,TRADE,EMPRCL,STATE,MP,BP,BITTER)
         IF (BP .GE. BOIL) THEN
           IF (LINE .EQ. 40) CALL NEWPG (LINE)
           CALL EXPRNT (THISID,TRADE,EMPRCL,BP)
           LINE = LINE+1
         ELSE
         END IF
      END DO

   99 REWIND 4
      WRITE (*,16)
   16 FORMAT (//1X,15X,'END OF RUN. NORMAL TERMINATION.')
      STOP
      END
```

**FIGURE 17.4** Search Program using the Unformatted File Built in Example 17.1.

Instead of looking for records whose contents meet certain conditions, the situation often is reversed. The user knows which record he or she wants. In response, the procedure must find the record and display its contents. (For example, such a file might contain records of all patients who visited a particular medical center during 1977, and a physician might ask to see the information relating to patient 000168.) Similarly, someone interested in Heavy Facts' chemical data (from Example 17.1) might request to see the data for the compound having i.d. 003287. With CHDATA organized as a sequential file, the program would have to search the file, record by record, until the right one is found. Then, when another request came in, the search for *that* record would be done the same way, starting again from the first record in the file. One way to avoid this search is to process several requests during the same run. For this to work with a sequential file, we would have to arrange the requests in the same order as the records appear in the file (i.e., by increasing i.d. number.) Then, a program can be designed to process several requests while performing only a single record-by-record search of the file. As is true in Example 17.1's program, such a procedure would have to include a mechanism to find and reject all input requests that are out of sequence. If there is a request for a particular record whose position is prior to the file's current position, it could not be filled unless the file were rewound to its first record and the search started over. Alternatively, if the equipment were designed for it, the program could laboriously backspace through the file until it reached the desired record. These strategies generally are unacceptable in practice, so that most processes requiring searches of sequential files are designed to reject improperly sequenced input data.

Even though the processing of multiple requests with a single sequential search increases the efficiency of such procedures, there still are times when this entire approach is unacceptable. For instance, there may be only one request ready to be processed and the user cannot wait until enough additional requests arrive to start another run. This type of situation is becoming more and more common with the growing use of programs designed to support ongoing conversations between user and machine.

A powerful solution for such circumstances is the *direct file*. In this type of organization (introduced in Section 13.1.1.2), every record in the file is accessible immediately, regardless of which record had been processed previously. We shall examine these files in more detail and see how their characteristics can be used to advantage.

### 17.2.1 Construction of Direct Files

When reading or writing a sequential file's record, there is no choice as to which record we are talking about. It is the record whose physical position is next in line. However, since a direct file allows access to any record at any time, there must be a way to identify each record, so that the programmer is assured of working with the record that he or she expects. FORTRAN's way of providing this assurance is to require that each record in a direct file have a *record number*. This simply is an integer value that indicates the record's sequential position in the file. (For instance, the first record in a direct file will have a record number of 1, the next one will have a record number of 2, and so on.)

Because of the importance of the position of a record in a direct file, it is not possible to build such a file in exactly the same record-by-record manner as is used for a sequential file. Instead, it is necessary to create an empty file with room enough for each record. This provides an organizational skeleton into which records can be placed in their proper positions, no matter in which order they are written. In other words, when a record is written into a particular position in a direct file, it fills a spot already reserved for it. The actual process of describing and reserving this file storage occurs outside the FORTRAN program. Accordingly, we shall assume that such a process has been made available and concentrate on the connection and use of direct files.

In addition to the requirement that each record in a direct file must have a record number, it is necessary for FORTRAN to impose the following restrictions on the organization and construction:

1. All records in a direct file must have the same length.
2. Records in a direct file all must be formatted or unformatted; they cannot be mixed. If they are formatted, they must be edit-directed.
3. A direct file cannot have an endfile record. (Since room on a disk or other direct access device is reserved for the entire file ahead of time, the system "knows" the size of the file, and the endfile is irrelevant.) There are special cases (which we shall not consider here) in which a file, originally constructed for sequential access, can be made available for direct access. When this happens, the endfile record (which is required in a sequential file) is ignored by FORTRAN and not considered part of the direct file.
4. Since the extent of an entire direct file is defined even before any actual data records are placed in that file, that size remains fixed as long as that file exists. Consequently, the programmer cannot remove a record from a direct file. Even if that record no longer is of any interest, it still occupies physical space in the file, and "uses up" a record number. For instance, we would not "remove" record 137 from a direct file and then have record 138 become the new 137, and so on. Everything remains the way it was when the file was created, and record 137 becomes a ghost town.

### 17.2.2 Opening a Direct File

A direct file is connected to a unit in the same way described earlier for sequential files, i.e., by means of the OPEN statement. Certain specific requirements should be noted:

1. The ACCESS='DIRECT' specifier must be present. (FORTRAN assumes sequential access if the ACCESS= specifier is left out of the OPEN statement.)
2. The RECL= specifier must be included. If the file is formatted, the record length is the number of characters. For unformatted direct files, the record length, for a given number of data items having a particular combination of types, depends on how those data are represented in the specific type of computer being used.

Thus, the statement

```
      OPEN (11,FILE=GROUP1,ACCESS='DIRECT',FORM='UNFORMATTED',
     1      RECL=120)
```

connects a direct, unformatted file with record length 120 to unit 11. (We shall use two-digit unit numbers for direct files to help emphasize the contrast with sequential units.) Similarly, the statement

```
      OPEN (UNIT=14,ACCESS='DIRECT',FORM='UNFORMATTED',
     1      RECL=180,IOSTAT=FILE14)
```

connects an unnamed, formatted direct file with record length 180 to unit 14. The variable named FILE14 will contain information relating to the outcome of the OPEN operation.

### 17.2.3 Direct Input/Output

Transmission to and from direct files is specified with the same READ and WRITE statements used for all other such operations. The following specific points are important:

1. The END= specifier cannot appear. As pointed out earlier, the endfile record is irrelevant in direct files.
2. The REC= specifier *must* appear. This is the only mechanism that FORTRAN has for identifying (and finding) records in a direct file. Consequently, even if the desired record happened to be the next in line after the record just used, FORTRAN cannot "know" that.
3. Records may be read or written in any order. For example, record 7 may be written before record 4 is written. Similarly, record 81 may be read before record 46 is read.
4. Although space is provided for each record in a newly constructed direct file, the storage is marked with special symbols that make it impossible for a program to read from a particular record unless that record had been written previously. The output operation does not have to take place in the same program that is doing the reading, but it does have to take place in order to wipe out the "vacant" marker in that record.
5. Once a record is written in a particular position, it can be rewritten. That is, new information can be written into that same position, replacing what was there before.

For example we can define a direct file named DIRCHM, with each record to contain the same information as described for CHDATA in the abovementioned example. For 16-bit computers like the HP 1000, for example, the information in such a record (molecular weight, year introduced, etc.) would require something like 66 characters (as unformatted data), so we shall specify as a record length:

```
      OPEN (UNIT=12,FILE=DIRCHM,ACCESS='DIRECT',
     1      FORM='UNFORMATTED',RECL=66)
```

Now, let us assume that we have read an input record and we wanted to write that information as a record in DIRCHM with the record number being taken from the compound's identifier (THISID). A statement like

```
      WRITE (12,REC=THISID,IOSTAT=HOW12) MWT,YR,TRADE,EMPRCL,
     1      STATE,MPT,BPT,BITTER)
```

is all that is required. Note that the value given for the REC= specifier is a variable. This, of course, is perfectly legal, as long as the result is a positive integer. Now, having written that record (let us suppose that THISID's value was 7163), we can read it from the file simply by saying

```
      READ (12,REC=7163) MWT,YEAR,TRADE,EMPRCL,STATE,
     1     MPT,BPT,BITTER
```

**Example 17.2** In this example we shall rewrite the program in Example 17.1 so that the file it produces (named DIRCHM) is a direct unformatted file with the record numbers taken from the chemicals' i.d. numbers (columns 1–6 of the input records). Since the records can be written in any order, it will not be necessary to test for "proper sequence" or to produce the error report required in the previous example. Figure 17.5 shows the revised program. (The names of the variables and subprograms have the same meanings as before, so they will be omitted to keep things brief. Similarly, Figure 17.5 includes only those subprograms requiring significant changes.

**Example 17.3** Now we shall write a program that retrieves records from the direct file developed in Example 17.2. Each request, submitted on a separate line, specifies the desired compound identification. Since this value also serves as the record number, the process of finding and printing the required record is straightforward. The main program is shown in Figure 17.6.

```
C*****************************************************************
C                  EXAMPLE 17.2 - THE MAIN PROGRAM                *
C*****************************************************************
C  THIS PROGRAM BUILDS A DIRECT FILE NAMED DIRCHM USING THE SAME*
C  INPUT INFORMATION DESCRIBED IN EXAMPLE 17.1. THE COMPOUND'S  *
C  IDENTIFIER, FOUND IN COLS. 1-6 OF EACH INPUT LINE, IS USED AS*
C  THE RECORD NUMBER. WE ASSUME HERE THAT THE APPROPRIATE AMOUNT*
C  OF DIRECT ACCESS STORAGE FOR DIRCHM HAS BEEN RESERVED BY A   *
C  MECHANISM EXTERNAL TO THIS PROGRAM.                          *
C*****************************************************************
$FILES 0,1
      PROGRAM         EX1702
      IMPLICIT        NONE
      REAL            MWT,MPT,BPT
      INTEGER*4       THISID,TTLNUM
      INTEGER*2       YR,BITTER,LINES
      CHARACTER       TRADE*20,EMPRCL*25,STATE*3
      LOGICAL         EOF

      CALL OPENER (TTLNUM,THISID,MWT,YR,TRADE,EMPRCL,STATE,MPT,BPT,
     1             BITTER,LINES,EOF)

      DO WHILE (.NOT. EOF)
         IF (LINES .EQ. 40) CALL NEWPG (LINES)
         CALL ADDREC (THISID,MWT,YR,TRADE,EMPRCL,STATE,MPT,BPT,
     1                BITTER,LINES)
         CALL RDCARD (THISID,MWT,YR,TRADE,EMPRCL,STATE,MPT,BPT,BITTER,
     1                EOF,TTLNUM)
      END DO

   99 WRITE (*,16) TTLNUM
   16 FORMAT ('1',1X,10X,'TOTAL NUMBER OF RECORDS: ',I6)
      STOP
      END
```

**(a)**

```
C*****************************************************************
C                              OPENER                            *
C*****************************************************************
C  THIS ROUTINE OPENS THE DIRECT FILE DIRCHM. THE SPECIFIED      *
C  RECORD LENGTH OF 66 IS BASED ON INTERNAL STORAGE REQUIREMENTS*
C  FOR STORING THE ITEMS USED IN THE EXAMPLE.                    *
C*****************************************************************
      SUBROUTINE      OPENER (T,TH,MW,Y,TR,EMP,ST,M,B,BTR,L,EOF)
      IMPLICIT        NONE
      REAL            MW,M,B
      INTEGER*4       T,TH
      INTEGER*2       Y,BTR,L
      CHARACTER       TR*20,EMP*25,ST*3
      LOGICAL         EOF

      T = 0
      CALL NEWPG (L)
      OPEN (12,FILE='DIRCHM',FORM='UNFORMATTED',
     1      ACCESS='DIRECT',RECL=66)
      CALL RDCARD (TH,MW,Y,TR,EMP,ST,M,B,BTR,EOF,T)
      RETURN
      END
```

**(b)**



**FIGURE 17.5** **(a)** Main Program for Example 17.2. **(b)** Initialization Routine for Example 17.2.

```
C*****************************************************************
C                              ADDREC                             *
C*****************************************************************
C   THIS ROUTINE WRITES A NEW RECORD INTO THE FILE DIRCHM USING  *
C   THE INFORMATION FROM THE CURRENT INPUT RECORD AND THE        *
C   COMPOUND IDENTIFIER AS A RECORD NUMBER. AS IS TRUE IN THE    *
C   VERSION FOR THE PREVIOUS EXAMPLE, IT CALLS PRTREC TO PRINT   *
C   THE INFORMATION JUST WRITTEN INTO DIRCHM.                    *
C*****************************************************************
      SUBROUTINE        ADDREC (TH,MW,Y,TR,EMP,ST,M,B,BTR,L)
      IMPLICIT          NONE
      REAL              MW,M,B
      INTEGER*4         TH
      INTEGER*2         Y,BTR,L
      CHARACTER         TR*20,EMP*25,ST*3

      WRITE (12,REC=TH) MW,Y,TR,EMP,ST,M,B,BTR)
      CALL PRTREC (TH,MW,Y,TR,EMP,ST,M,B,BTR,L)
      RETURN
      END
```

**(c)**

**FIGURE 17.5 (c)** Output routine for Example 17.2.

```
C*****************************************************************
C                     EXAMPLE 17.3 - THE MAIN PROGRAM             *
C*****************************************************************
C  THIS PROGRAM PROCESSES REQUESTS TO RETRIEVE AND PRINT DATA     *
C  OBTAINED FROM SPECIFIED RECORDS OF THE DIRECT FILE DIRCHM      *
C  PRODUCED BY THE PROGRAM IN EXAMPLE 17.2. EACH REQUEST IS       *
C  SUBMITTED AS A COMPOUND I.D. NUMBER IN COLUMNS 31-36 OF A      *
C  TYPED LINE THAT ALSO CONTAINS THE REQUESTOR'S IDENTIFICATION   *
C  AS A CHARACTER STRING IN COLUMNS 1-25.                         *
C  PRTREQ IS A ROUTINE (NOT SHOWN) THAT PRINTS A LINE SHOWING     *
C  THE REQUESTOR'S IDENTIFICATION AND THE RECORD'S CONTENTS.      *
C*****************************************************************
      PROGRAM         EX1703
      IMPLICIT        NONE
      REAL            MWT,MPT,BPT
      INTEGER*4       THISID
      INTEGER*2       YR,BITTER
      CHARACTER       USERID*25,TRADE*20,EMPRCL*25,STATE*3

      OPEN (12,FILE="DIRCHM',FORM='UNFORMATTED',ACCESS='DIRECT',
     1      RECL=66,STATUS='OLD')
      CALL NEWPG

      DO WHILE (.TRUE.)
         READ (*,15,END=99) USERID,THISID
         READ (12,REC=THISID) MWT,YR,TRADE,EMPRCL,STATE,MPT,BPT,BITTER)
         CALL PRTREQ (USERID,THISID,MWT,YR,TRADE,EMPRCL,STATE,
     1                MPT,BPT,BITTER)
      END DO

   99 WRITE (*,17)
   17 FORMAT (//1X,10X,'END OF RUN. NORMAL TERMINATION.')
      STOP
      END
```

**FIGURE 17.6** Main Program for Example 17.3.

## 17.3 SUMMARY

An *unformatted file* is one in which the data are recorded as an exact copy of their internal representation in the processor's storage. Consequently, there is no need for FORTRAN to interpret (i.e., edit) the data in order for the processor to be able to work with the values. This reduces the time it takes to transmit the data to or from the processor and make it ready for use. At the same time, it also means that the information is not in human-readable form. Thus, unformatted files are used to store data that are to be transferred from one program to another, rather than between a program and a human. Unformatted files may be sequential or direct, and they usually are stored on magnetic media (such as magnetic tape or disk).

A *direct file* is one in which the records can be read or written in any desired sequence. This is made possible by assigning a unique *record number* to each record. This identifies the record's physical position in the file and also serves as a "handle" so that the record can be found at any time. Direct files may be formatted or unformatted, but they must be stored on a direct access device (such as a magnetic disk).

## PROBLEMS

1. Write FORTRAN statements for each of the following activities:

   (a) Create an unformatted sequential file named MASTER and connect it to unit 8,
   (b) Create an unformatted unnamed sequential file and connect it to unit 3.
   (c) Create a formatted sequential file named NEWMTS and connect it to unit 4.
   (d) Connect an unformatted unnamed sequential file to unit 1.
   (e) Connect a formatted direct file named RNDREC with record length 120 to unit 14.
   (f) Connect an unformatted unnamed direct file with record length 80 to unit 12, storing the outcome of the operation in a variable named HOW012.

2. Through a combination of circumstances beyond your control, the President of your company has put his favorite nephew Farnsworth (affectionately known as Dumb Farnsworth) in charge of the computer tapes. Oh my. Somehow, the talented nephew has managed (he will not say how) to create a situation in which the adhesive labels on the tape reels have fallen off. Yes, you say this is highly unlikely, but that is because you do not know Farnsworth.

   Anyway, there you are, with all these tapes. It is necessary to find out which tape has what. The situation is not hopeless because all of the files are sequential, and all of them are named. (There is only one file on any reel of tape.) Accordingly, you have been asked to write a program that will examine a reel and find out the following things about it:

   (1) The name of the file.
   (2) Whether the file is formatted or unformatted;
   (3) The number of records in the file.

   NOTE: The unit number to which you connect the file will depend on the unit numbers available in your particular system.

   (a) Write a brief narrative description of how you are going to handle this situation.
   (b) Prepare a pseudocode or N-S representation of your design.
   (c) Write the program.

3. More mischief from Dumb Farnsworth: There are two reels of tape, each containing an unnamed, unformatted, sequential file. It is suspected that the two tapes contain copies of the same file, but it is not known whether the copies are of the same *version* of that file. Each record contains the following data items:

| *Item No.* | *Name* | *Data Type* |
|---|---|---|
| 1 | CRTID | integers |
| 2 | CRTNAM | 20 characters |
| 3 | BRTAMT | real |
| 4 | TIME(1) thru TIME(8) | integer values |
| 5 | TEMP(1) thru TEMP(8) | real values |
| 6 | RATING | a one-digit integer |

It is known that each file has the same number of records, and that corresponding records have matching values for CRTID and matching values for CRTNAM. The order of the data items in the table given above is the same as that in which they appear in the record. Write a program designed to determine whether the files are identical. That is, the values in each record must match those in the corresponding record of the other file. The program is to print:

(1) The number of records in each file.
(2) The record numbers (the first record is record 1, etc.) for those records in which the contents on one file differ from those on the other.
(3) The CRTID values for those records having mismatched values.

4. Modify the program in Problem 3 so that it provides the following additional details: When the contents of a record on one file differ from those on the corresponding record of the other file, the program is to print the record number and CRTID as in the previous problem. In addition, it is to print the name of the item and the two (differing) values for each item in which the values are different. For example,

```
RECORD NO.   127     TIME(5)  27      TEMP(3) 0.348567E+03
CRTID       3564              31              0.365897E+03
```

(This is just a suggested format; heading information at the top of the page could inform the user as to which of the two output lines goes with which unit number.)

5. Write the NEWPG subroutine for Example 17.1 (see Figure 17.3(c)).

6. Write the PRTREC subroutine for Example 17.1 (see Figure 17.3(f)).

7. Write the ERRPRT subroutine for Example 17.1 (see Figure 17.3(i)).

8. Write the NEWPG and EXPRNT subroutines for the main program shown in Figure 17.4.

9. Modify the program in Figure 17.4 so that it processes several search requests in a single run. (This idea is discussed briefly just before the beginning of Section 17.2.1, but in a slightly different context.) That is, Heavy Facts would like to be able to submit a collection of customer requests, each asking for information on compounds having a specified minimum boiling point. Note that the minimum boiling point specified in a particular request may be equal to or different from that indicated in other requests. Thus, a particular compound may fulfill the requirements of several requests. Note also that Heavy Facts recognizes that such a program will print the output for all the various requests mixed together. This will be satisfactory, even though it will mean that Heavy Facts will have to go through the output and prepare separate reports for each request. To help unscramble the output, figure out a way of identifying each request. Then, after the entire file has been inspected and all the appropriate records have been printed, print a summary (on a new page) indicating the number of records (i.e., the number of compounds) that meet the requirements for each of the requests processed. Note that there may be one or more requests for which the file contains no appropriate compounds. Make sure that your program is designed to recognize this, and that it lets the user know. Limit your program to ten requests for each run and include appropriate protection so that attempts to submit more than ten requests are detected and handled in a reasonable way.

10. Modify the program in Problem 9 so that it can handle any number of collections of requests, with up to ten requests in each collection. (It is up to you to decide how you will separate the batches of requests.)

11. Heavy Facts knows that there is more to this business than minimum boiling points. Accordingly, they would like you to expand the capabilities of the program in Problem 9 or Problem 10 so that it recognizes and handles any mixture of requests that may include the following types:

(1) Print information for all compounds having a boiling point equal to or greater than the specified input value.
(2) Print information for all compounds having a boiling point equal to or less than the specified input value.
(3) Print information for all compounds having a melting point equal to or greater than the specified input value.
(4) Print information for all compounds having a melting point equal to or less than the specified input value.
(5) Print information for all compounds having a bitterness index equal to or greater than the specified input value.

(5) Print information for all compounds having a bitterness index equal to or greater than the specified input value.
(6) Print information for all compounds having a bitterness index equal to or less than the specified input value.
(7) Print information for all compounds having a molecular weight equal to or greater than the specified input value.
(8) Print information for all compounds having a molecular weight equal to or less than the specified input value.
(9) Print information for all compounds whose year of introduction is equal to or greater than the specified input value.
(10) Print information for all compounds whose year of introduction is equal to or less than the specified input value.

Note that only one of these search specifications can be given in a single request, but a single run may contain up to ten requests consisting of any mixture of these items. Make sure that your program includes detailed comments describing how these requests are specified and recognized.

12. Add the following list to the types of search requests accepted by the program in Problem 11:

(1) Print information for all compounds whose molecular weights lie within a specified range of temperatures.
(2) Print information for all compounds whose melting points lie within a specified range of temperatures.
(3) Print information for all compounds whose boiling points lie within a specified range of temperatures.
(4) Print information for all compounds whose bitterness indexes lie within a specified range of values.
(5) Print information for all compounds whose respective years of introduction lie within a given range.

13. One of the data items present in each record of the CHDATA file is a three-character string indicating the state of the compound at room temperature (SOL, LIQ, or GAS). Using a room temperature value of 20 degrees Centigrade, write a program that checks each record of CHDATA to make sure that the recorded state at room temperature is consistent with the melting point and boiling point recorded in that record. Assume that the melting and boiling points are correct. For example, if a compound melts at 31 degrees Centigrade and boils at 112.5 degrees Centigrade, it would be inconsistent to report its state as LIQ. (It should be SOL.) List the i.d. number, trade name, molecular weight, melting point, boiling point, and reported state at room temperature for all records in which such mismatches have been found. After all the selected records have been printed, the program is to print the number of records found to be inconsistent.
14. Expand the program in Problem 13 by producing a new file named NEWCHM on unit 8. In addition to performing all the processing described in Problem 13, this program is to copy all of CHDATA's records into NEWCHM in the same order, changing the inconsistent values in accordance with its findings.

15. A number of Heavy Facts' customers have expressed an interest in being able to request information on compounds in the CHDATA file based on more than one criterion. For example, a customer may want to see information on all compounds having a molecular weight of at least, say, 350.0 and a boiling point less than, say 135 degrees Centigrade. To do that with the program described in Problems 9 through 14, it would be necessary for the customer to submit two requests: One for molecular weight and one for boiling point. Then, the customer would have to inspect the two output lists thus produced and find those compounds that appeared on both lists. Unhappy customers. To convert them to happy customers, modify one of the versions of this program so that it will accept and process a request consisting of a combination of any two of the criteria listed in Problem 11.

16. Expand the capabilities of the program in Problem 15 so that it will accept requests consisting of a combination of any two of the criteria listed in Problem 12 as well as in Problem 11.

17. Write a general version of the program specified in Problem 15 or 16 so that each request may be a combination of as many criteria as the user wishes to specify, consistent with the list(s) given in Problem 11 (and 12). For example this program should be able to accept a request to print the information about all compounds whose molecular weights, melting points, boiling points, bitterness indexes, and years of introduction are within specified ranges. As in previous problems, there is no guarantee that a given request will produce information about any compounds at all. Consequently, the program must be prepared to inform the user when this happens.

18. Reorganize any of the programs specified in Problems 9 through 17 so that it is no longer necessary to mix together the output for all of the requests. Specifically, the following approach is suggested: Assume units 21–30 are available for connection to output files that will contain records responding, respectively, to each of the ten possible requests. Then, as each of CHDATA's records is examined to see which (if any) of the requests it fulfills, it can be written on (one or more of) the appropriate output files. When all of CHDATA's records have been processed, the program can produce a separate report for each request. Now, everyone will be happy.

19. All compounds in the CHDATA file contain at least five atoms of carbon. Expand any of the CHDATA programs discussed in Problems 10 through 18 so that the user also can request information on compounds containing at least a specified number of carbon atoms.

20. Design a version of the program in Problem 19 so that the user can request information on all compounds containing a number of carbon atoms within a specified range.

21. One of the most frequently used file processes is called *file updating*. In general, this involves making changes in a file to reflect newly available data. There are three basic types of operations:

    (1) *Addition:* Enlarging the file by incorporating new records.
    (2) *Deletion:* Removing records from a file.
    (3) *Modification:* Changing the values of various data items in selected records of a file. This latter operation usually is performed by replacing (i.e., deleting and then adding) each affected record.

    When a sequential file is updated, especially one that happens to be stored on magnetic tape, the most common practice is to use two input files: One of these is the tape containing the file to be updated (with the records in a particular sequence). The second (usually entered from a terminal) contains the information required for the updating run. Each record of this second input file usually contains information relating to the addition, deletion, or modification of a single record:

    (1) An input record describing an addition contains sufficient information so that the program can build the new record *and define its position in the updated file.*
    (2) An input record describing a deletion often need contain little more than the position of the record to be deleted, and some kind of signal to indicate that deletion of that record is desired.
    (3) An input record describing modification defines the position of the record to be modified, along with the particular items in that record whose values are to be changed.

    The records in this second input file (i.e., the updating information) are arranged so that their sequencing is the same as that used for the other input file. For example, in the CHDATA file constructed in Example 17.1, the records are sequenced by increasing compound i.d. number. Accordingly, each updating record to be processed against CHDATA must contain a compound i.d. number, and the records must be arranged so that the first one has the lowest i.d., number, and so on.

    Once these arrangements have been made, the updating program is designed to go through a cycle in which, at any given time, there is a current record from the old file and a current record from the updating file. The program compares these records and systematically uses the information to build a new version of the file on a separate unit. The old file is retained for safekeeping in case something goes wrong, so that the difficulty can be repaired and the process repeated. Assuming the records in both input files are in proper sequence, the updating cycle considers the following three basic possibilities:

    (1) If the sequence numbers in the two current records match (i.e., the data in the current updating record refer to the current record from the old file), the program performs the activity indicated by the updating record. If a deletion is called for, the program deletes the current record, i.e., it neglects to write the record onto the new file and simply reads the next record from the old file (as well as the next updating record). If the instruction is to modify the current record, the program introduces the indicated changes and then writes the new version into the output file. If the instruction is to add the record, something is wrong, since the record already is there.
    (2) If the sequence number of the old record refers to a position earlier in the file than the one on the updating record, it means that there is no updating on that record. Accordingly, it can be copied into the output file without change, after which the next record can be read from the old file.
    (3) If the sequence number of the old record refers to a position later in the file than the one on the updating record, it means that a new record is to be built and placed in the output file ahead of the current record from the old file. Once this is done, the next updating record can be read and its sequence number compared with the (still untouched) current record from the old file. If the

sequence numbers compare as stated above and the instructions with the current updating record do not call for the record to be added, something is wrong: It makes no sense to delete or change a record that is not there.

This is just a bare outline of the basic updating process, but it is enough to give you a good idea of what must go. Write a program that updates the CHDATA file built in Example 17.1 by producing an unnamed output file on unit 8. Whenever an old record is copied into the new file without change, the program is to print nothing. Whenever a record is added, deleted, or changed, the program should produce a line of print showing what happened. (It is up to you to determine what to print in each case.) After the updating process is complete, the program is to print (starting on a new page) a summary indicating the number of records originally on the file, the number of records added, the number deleted, the number changed, and the number of records in the new (updated) file.

22. Write the same program described in Problem 21, but design it so that the file being updated is the direct file DIRCHM built in Example 17.2. Thus, instead of producing a new version of the file as a separate copy connected to a separate unit, this program rewrites the updated records back into their respective positions in DIRCHM. Note that an "addition" in this situation does not make the file any larger. You are to assume that positions already exist for all the records, and an addition simply fills an "empty" record. Similarly, a "deletion" does not make the file any smaller. Output requirements still are the same as in the previous problem.

23. Expand the program in Problem 22 so that it provides some insurance in the following way: While it updates the direct file DIRCHM, it also produces on unit 3 a sequential file named OLDCHM that represents the data prior to the updating process.

# 18
# *Shared Data Among Program Components*

The method that we have used for transferring information between subprograms or between a main program and a subprogram is one of two basic techniques for providing a program component with external information on which to operate. In this chapter we shall examine another approach: Instead of invoking a subprogram with an accompanying list of explicit arguments, it is possible to organize a program so that one or more areas of storage are reserved and set aside for data to be shared among a main program and its associated subprograms. Such areas are called *common areas* or *common blocks*, and their use can make it unnecessary to supply subprograms with arguments. Correspondingly, they can make it possible to design subprograms without dummy argument lists, even though they process information supplied from outside the subprogram.

The use of common storage seems attractive at first glance, but there are serious drawbacks to it. In fact, some of the structural disadvantages lie behind our delay in introducing this feature till now. This chapter is intended to point out these strengths and weaknesses, and it will provide some useful guidelines to help determine those situations in which there is a clear preference for one approach over the other.

## 18.1 CHARACTERISTICS OF COMMON STORAGE

When we specify an argument for a subprogram, we are telling the subprogram where to find the value that it is to use this time. That value may be located anywhere in the processor's storage. With common storage, on the other hand, FORTRAN sets aside a special area in which all shared values are stored together. Each program component that needs it is given access to this area by including the area's description in the form of a special declaration. In this way, the program component always "knows" what the common area looks like and it can find any variable in that area without further help. We can take advantage of this organizational feature by designing programs so that they use shared data, thereby eliminating (or drastically reducing) the need for argument lists.

The basic concept of shared storage is illustrated in Figure 18.1. Pictured there is a program consisting of a main program `M` and two subroutines `S1` and `S2`. Available to all three program components (or *program units*), and described in each one, is a common area containing (among other items) variables `V1`, `V2`, `V3`, and `V4` and these are used all over the program in various ways. Their values may or may not change, depending on how they are used. Note that the calls to `S1` and `S2` do not have arguments. Correspondingly, `S1` and `S2`'s `SUBROUTINE` statements do not have dummy argument lists. All communication having to do with which data to use is handled through the shared area.

It should be pointed out that the use of common storage does not rule out the use of arguments. Often, the most convenient arrangement is one in which a subprogram's data are supplied by a combination of a small argument list and a common storage area.

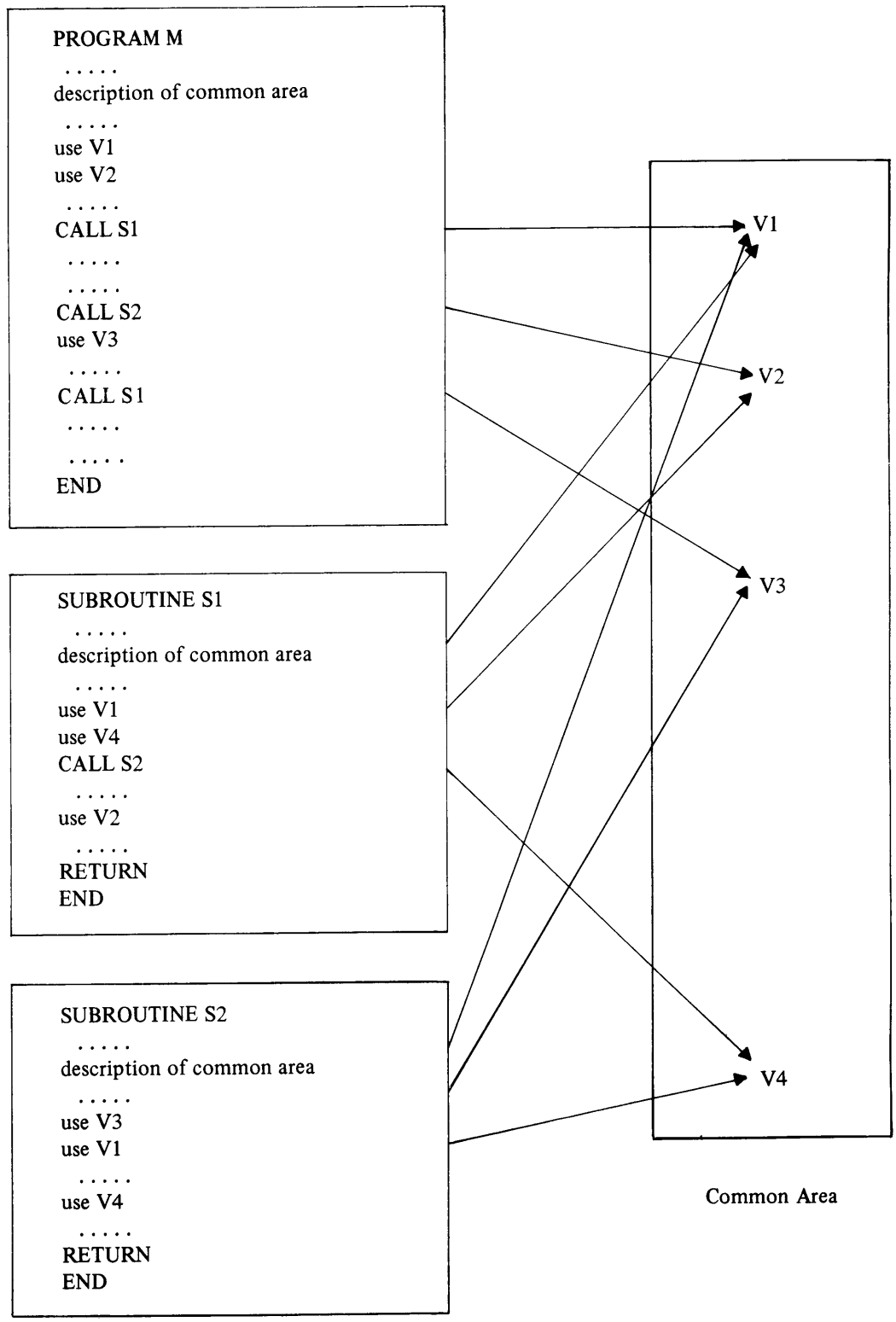



**FIGURE 18.1** Use of a Common Area by Several Program Components.

### 18.1.1 Unnamed Common Storage

The most basic form of common storage is called *blank common storage* or just *blank common*. This simply is an area whose size is determined by the programmer's declarations. These are specified by means of a COMMON statement:

```
COMMON      name, name, name, etc.
```

The list of names tells HP FORTRAN 77 which variables to include in the special area, i.e., the *common block*. There can be only one blank common area in an entire program, regardless of the number of subprograms associated with that program. Each variable's data type can be defined by means of an ordinary declaration, just as we have been doing all along. To illustrate, let us take a look at the following statements:

```
REAL          A, SLT, BKWTH
REAL          STRN, CRTV
INTEGER*2     TTLS, SBTL, RJTL
CHARACTER*4   WORD, ACRONYM
COMMON        A, SLT, BKWTH, TTLS, RJTL, WORD, ACRONYM
```

In this instance, we have set up a blank common area containing real variables A, SLT, BKWTH, and integer variables TTLS and RJTL, and character variables WORD and ACRONYM. Note that these variables are not really declared twice. The REAL, INTEGER, and CHARACTER statements describe the variables and request storage for them; the COMMON statement tells FORTRAN that the storage reserved for the variables in its list should be part of the common area.

The real variables have been declared in two separate statements (while the integers and characters have not) just to show that either form is acceptable. In general, the use of separate statements makes it easier to keep track of which variables are in the common block and which are not.

#### 18.1.1.1 Arrangement of Space in Common Storage

The blank common area is made available to any subprogram simply by including a copy of the appropriate declarations and the COMMON statement. (This is another reason for separating the shared variables' declarations from the others.)

Variables in a common area are stored in the order in which they are listed in the COMMON statement. Thus, in the example just listed, the common block would look like this:

| A | SLT | BKWTH | TTLS | RJTL | WORD | ACRONYM |
|---|---|---|---|---|---|---|
| 1–2 | 3–4 | 5–6 | 7 | 8 | 9–10 | 11–12 |

**Sequential Word Numbers (for 16-bit computers)**

The order can be important because it is possible for each of several subprograms to refer to the same shared variable by a different name. When this is done, it is necessary to know exactly how the common block is organized. This is discussed in Section 18.1.1.4.

#### 18.1.1.2 Arrays in Blank Common Storage

Arrays may be included in a common block along with single-valued variables. There are no restrictions on the array's position in the common area. Along with everything else, its placement will depend on its appearance in the COMMON statement. For instance, the

declarations

```
REAL              PR, TMP, VEL(4), ABS
INTEGER*2         MTN, REL
CHARACTER*2       WD
COMMON            PR, TEMP, VEL, ABS, MTN, REL, WD
```

will produce a common block that looks like this:

```
PR   TMP   VEL(1)   VEL(2)   VEL(3)   VEL(4)   485   MTN
```

If we rewrite the COMMON statement so that it says

```
COMMON      VEL, PR, TMP, ABS, MTN, REL
```

the organization of the resulting common block will be

| VEL(1) | VEL(2) | VEL(3) | VEL(4) | PR | WD | TMP | ABS | MTN | REL |
|---|---|---|---|---|---|---|---|---|---|
| 1–2 | 3–4 | 5–6 | 7–8 | 9–10 | 11 | 12–13 | 14–15 | 16 | 17 |

FORTRAN also allows the array description to be given in the COMMON statement, with the other declarations just providing the data type descriptions. Thus, the statements

```
REAL              PR, TMP, VEL, ABS
INTEGER*2         MTN, REL
CHARACTER*2       WD
COMMON            VEL(4), PR, TMP, ABS, MTN, REL
```

will produce the same common region as the one given above.

Multidimensional arrays are handled the same way. FORTRAN "knows" how large the array is (from its declared dimensions), and so the appropriate amount of storage is made available in the common block. To illustrate, the statements

```
REAL              BT(2), RS
INTEGER*2         C(2,3)
COMMON            BT, RS, C
```

produce a common block that is organized as follows:

| BT(1) | BT(2) | RS | C(1,1) | C(2,1) | C(1,2) | C(2,2) | C(1,3) | C(2,3) |
|---|---|---|---|---|---|---|---|---|
| 1–2 | 3–4 | 5–6 | 7 | 8 | 9 | 10 | 11 | 12 |

**Example 18.1** We shall illustrate the simple use of blank common by rewriting Example 17.1 so that the dummy argument lists (and corresponding argument lists) are replaced by a blank common block containing the variables we shall use. The main program, for instance, (Figure 18.2(a)) performs all of its invocations to the subroutines without using any arguments (See Figure 17.3(a) for comparison.)

Two of Example 18.1's subroutines also are shown (Figures 18.2(b) and 18.2(c)) to illustrate the corresponding declaration of the common block. Compare this version of OPENER with the previous one in Figure 17.3(b). You will note that the variable names have been changed so that they are the same as the ones declared for the common block. The same is true for the processing in the revised version of RDCARD, which can be compared with its predecessor in Figure 17.3(d). The construction of the other subroutines follows the same approach: Each contains a copy of the declarations, thereby making the shared variables accessible to them as well.

```
C*****************************************************************
C                    EXAMPLE 18.1 - THE MAIN PROGRAM              *
C*****************************************************************
C  THIS PROGRAM PERFORMS THE SAME PROCESSING AS THE ONE IN        *
C  EXAMPLE 17.1. THE ONLY DIFFERENCE IS THAT THE ARGUMENTS        *
C  (AND, THEREFORE, THE CORRESPONDING DUMMY ARGUMENTS)            *
C  HAVE BEEN TAKEN OUT AND RECONSTRUCTED AS A BLANK COMMON        *
C  BLOCK. THE VARIABLE NAMES HAVE THE SAME MEANINGS AS BEFORE.    *
C*****************************************************************

      PROGRAM   EX1801
      IMPLICIT  NONE
      REAL      MWT,MPT,BPT
      INTEGER*4 THISID,PREVID,TTLNUM,NUMACC,NUMREJ
      INTEGER*2 YR,BITTER,LINES
      CHARACTER TRADE*20,EMPRCL*25,STATE*3
      LOGICAL   EOF
      COMMON    MWT,MPT,BPT,THISID,PREVID,TTLNUM,NUMACC,NUMREJ,
     1          YR,BITTER,LINES,EMPRCL,TRADE,STATE,EOF

      CALL OPENER

      DO WHILE (.NOT. EOF)
         IF (THISID .LE. PREVID) THEN
           CALL ERRPRC
           NUMREJ = NUMREJ+1
         ELSE
           IF (LINES .EQ. 40) CALL NEWPG
           CALL ADDREC
           NUMACC = NUMACC+1
         END IF
         CALL RDCARD
      END DO

      CALL SUMPRT
      CALL ERRPRT
      STOP
      END
```

*(Continued)*

**FIGURE18.2 (a)** Main Program for Example 18.1.

**18.1.1.3 Variable Names in a Common Block** The reason Example 18.1 will operate properly is that the common block is described in exactly the same way in each program component in which its variables are used. We did this simply by copying the COMMON statement from the main program into each subroutine that uses the shared variables. This is an easy way to make sure that the descriptions match, and it is a good practice to follow whenever possible.

However, there are occasions where it may be impractical to set up such exact copies. For instance, an organization may obtain a subroutine from somewhere for use in its programs, and that subroutine may be set up to use variables from a common block. As a result, there will be no list of dummy arguments, and the subroutine will use names that probably are different from those used in other program components. FORTRAN makes it unnecessary to change the various names so that they match. A common block may be

```
C*************************************************************
C                          OPENER                             *
C*************************************************************
C  THIS ROUTINE PERFORMS EXACTLY THE SAME FILE OPENING OPERA- *
C  TIONS AS THE CORRESPONDING ROUTINE IN EXAMPLE 17.1         *
C  (FIGURE 17.3(B)).                                         *
C*************************************************************
      SUBROUTINE   OPENER
      IMPLICIT     NONE
      REAL         MWT,MPT,BPT
      INTEGER*4    THISID,PREVID,TTLNUM,NUMACC,NUMREJ
      INTEGER*2    YR,BITTER,LINES
      CHARACTER    TR*20,EMP*25,ST*3
      LOGICAL      EOF
      COMMON       MWT,MPT,BPT,THISID,PREVID,TTLNUM,NUMACC,NUMREJ,
      1            YR,BITTER,LINES,EMPRCL,TRADE,STATE,EOF

      OPEN (3,FILE='CHDATA',FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
      OPEN (2,FILE='CHMERR',FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
      TTLNUM=0
      NUMACC=0
      NUMREJ=0
      THISID=0
      CALL NEWPG
      CALL RDCARD
      RETURN
      END
```

**(b)**

```
C*************************************************************
C                          RDCARD                             *
C*************************************************************
      SUBROUTINE  RDCARD
      IMPLICIT    NONE
      REAL        MWT,MPT,BPT
      INTEGER*4   THISID,PREVID,TTLNUM,NUMACC,NUMREJ
      INTEGER*2   YR,BITTER,LINES
      CHARACTER   TR*20,EMP*25,ST*3
      LOGICAL     EOF
      COMMON      MWT,MPT,BPT,THISID,PREVID,TTLNUM,NUMACC,NUMREJ
      1           YR,BITTER,LINES,EMPRCL,TRADE,STATE,EOF

      PREVID = THISID
      EOF = .FALSE.
      READ (*,15,END=99) THISID,MWT,YR,TR,EMP,ST,MPT,BPT,BITTER
   15 FORMAT (I6,F6.2,I4,A20,A25,A3,F5.1,F6.1,I1)
      TTLNUM = TTLNUM+1
      GO TO 77
   99 EOF = .TRUE.
   77 RETURN
      END
```

**(c)**

**FIGURE 18.2** **(b)** Initialization Routine for Example 18.1. **(c)** RDCARD Routine for Example 18.1.

described with different names in different subprograms sharing its use as *long as the specifications describe the exact same organization.*

We shall set up a simple case to see how the principle works: Suppose we have a main program in which the following declarations define a common block:

```
REAL            PV, R1 (4) , CS
INTEGER*2       T (5) , NUM
COMMON          PV, T, R1, CS, NUM
```

The resulting block consists of storage for a real number, five integers, five real numbers (the four from array RUN1 followed by CS, and an integer. A subprogram sharing this common area can do so quite legally regardless of the names it uses, as long as it describes the same block with the same number of items having the same data types in the same order. For instance, the declarations

```
REAL            V1 (4) , TR, BL
INTEGER*2       CT, S (5)
COMMON          BL, S, V1, TR, CT
```

describe an identical organization, so that FORTRAN can make the following matchup:

```
PV T(1)  T(2)  T(3)  T(4)  T(5)  R1(1)  R1(2)  R1(3)  R1(4)  CS NUM
BL S(1)  S(2)  S(3)  S(4)  S(5)  V1(1)  V1(2)  V1(3)  V1(4)  TR CT
```

Another subprogram designed to use the same common block can describe it with still a third set of names, and FORTRAN will be able to work it out as long as the organization thus described is the same as that described by the other specifications. If the situation becomes confusing, it is not FORTRAN that will be confused; it is the programmer. Consequently, although these elaborate bookkeeping facilities are available, every effort should be made to avoid having to use them.

**18.1.1.4 Different Declarations for a Common Block** The ability to describe a common block in different ways enables the programmer to go beyond the use of different names for a particular variable or array. It is possible, for instance, to specify an array in one description and to treat that same storage area as a list of single-valued variables in a description installed in another subprogram. The only restriction is that the data types must match. (Even *that* restriction can be overcome as well, but the reasons for doing so are special enough to rule out further discussion in this text.) To illustrate the use of different descriptions for blank common, we shall write a third description for the common block constructed in the previous section:

```
REAL            K(2) , R2 (3) , X
INTEGER*2       Y (4) , CN, ND
COMMON          X, CN, Y, K, R2, ND
```

If we look at the COMMON statement carefully, we see that the organization described there still follows the same pattern as the one in the other two COMMON statements: We still have a real number (X), five integers (CN and the four elements of array Y), five real numbers (K's two elements followed by R2's three elements) and, finally an integer (ND). The correspondence is shown below:

```
PV T(1)  T(2)  T(3)  T(4)  T(5)  R1(1)  R1(2)  R1(3)  R1(4)  CS     NUM
BL S(1)  S(2)  S(3)  S(4)  S(5)  V1(1)  V1(2)  V1(3)  V1(4)  TR     CT
 X CN    Y(1)  Y(2)  Y(3)  Y(4)   K(1)   K(2)  R2(1)  R2(2)  R2(3)  ND
```

This feature allows great flexibility in defining common areas, but there is no point in using it unless it makes things simpler and clearer. In many situations, it has the opposite effect. Consequently, now that the basic capability has been described, it will not be pursued beyond this point.

### 18.1.2 Named Common Blocks

There are times when it is convenient to work with several common blocks in the same program. This is done by declaring such areas as *named common blocks:*

COMMON /*blockname*/*name*,*name*,*name*,etc.

The name used for *blockname* is constructed like any other variable name, and the associated list of variable names defines the size and organization of that common block in the same way as we discussed for blank common. This is a flexible mechanism because we can declare any number of named common blocks. (Of course, each block must have a unique name, and a variable cannot appear in more than one block.) By dividing the shared variables among several common blocks, we can regulate the construction of our programs so that program units needing to share only certain items will be able to deal with common areas whose contents are restricted just to those items.

```
C*****************************************************************
C                      EXAMPLE 18.2 - THE MAIN PROGRAM            *
C*****************************************************************
C  THE PROCESSING IN THIS PROGRAM IS EXACTLY THE SAME AS THAT IN *
C  THE PREVIOUS EXAMPLE. HOWEVER, THE SHARED AREAS ARE REARRANGED*
C  INTO SEVERAL DISTINCT NAMED COMMON BLOCKS. THE FOLLOWING      *
C  SPECIFIC ADJUSTMENTS SHOULD BE NOTED:                         *
C    THE THREE CHARACTER VARIABLES TRADE, EMPRCL AND STATE ARE   *
C    SET UP AS A SEPARATE NAMED COMMON AREA (CHARS);             *
C    THE LINE COUNTER (LINES) IS NOT INCLUDED IN ANY OF THE      *
C    COMMON BLOCKS. BECAUSE OF THE WAY IT IS USED, IT WAS DECIDED*
C    TO TRANSMIT IT AS AN ARGUMENT. SAME WITH EOF.               *
C*****************************************************************
      PROGRAM       EX1802
      IMPLICIT      NONE
      REAL          MWT,MPT,BPT
      INTEGER*4     THISID,PREVID
      INTEGER*2     YR,BITTER
      INTEGER*4     TTLNUM,NUMACC,NUMREJ
      INTEGER*2     LINES
      CHARACTER     TRADE*20.EMPRCL*25,STATE*3
      LOGICAL       EOF
      COMMON        /REC/THISID,MWT,YR,MPT,BPT,BITTER,PREVID
      COMMON        /CHARS/TRADE,EMPRCL,STATE
      COMMON        /TTLS/TTLNUM,NUMACC,NUMREJ

      CALL OPENER (LINES,EOF)

      DO WHILE (.NOT. EOF)
         TTLNUM = TTLNUM + 1
         IF (EOF .EQ. .TRUE.) GO TO 199
         IF (THISID .LE PREVID) THEN
           CALL ERRPRC
           NUMREJ = NUMREJ+1
         ELSE
           IF (LINES .EQ. 40) CALL NEWPG (LINES)
           CALL ADDREC
           LINES = LINES + 1
           NUMACC = NUMACC + 1
         END IF
         CALL RDCARD (EOF)
      END DO

      CALL SUMPRT
      CALL ERRPRT
      STOP
      END
```

**FIGURE 18.3 (a)** Main Program for Example 18.2.

```
C*****************************************************************
C                          OPENER                                *
C*****************************************************************
C  OPENER USES TWO OF THE THREE NAMED COMMON AREAS SINCE IT SETS  *
C  THE COUNTERS TO THEIR INITIAL VALUES AND IT INITIALIZES THISID *
C  AS WELL. FOR THE LATTER, IT NEEDS THE ENTIRE COMMON AREA REC   *
C  (SINCE IT IS IMPOSSIBLE TO SHARE PART OF A COMMON AREA). IN    *
C  ADDITION, IT USES THE LINE COUNTER AND ENDFILE INDICATOR, WHICH*
C  IT TRANSMITS AS ARGUMENTS TO NEWPG AND RDCARD, RESPECTIVELY.   *
C*****************************************************************
      SUBROUTINE   OPENER (LN,FL)
      IMPLICIT     NONE
      REAL         MWT,MPR,BPT
      INTEGER*4    THISID,PREVID
      INTEGER*2    YR,BITTER
      INTEGER*4    TTLNUM,NUMACC,NUMREJ
      INTEGER*2    LN
      LOGICAL      FL
      COMMON       /REC/THISID,MWT,YR,MPT,BPT,BITTER,PREVID
      COMMON       /TTLS/TTLNUM,NUMACC,NUMREJ

      OPEN (3,FILE='CHDATA',FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
      OPEN (2,FILE='CHMERR',FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
      TTLNUM = 0
      NUMACC = 0
      NUMREJ = 0
      THISID = 0
      CALL NEWPG (LN)
      CALL RDCARD (FL)
      RETURN
      END
```

**(b)**

```
C*****************************************************************
C                            RDCARD                              *
*****************************************************************
C  THIS ROUTINE STILL READS AN INPUT RECORD, AS IT DID BEFORE.    *
C  HOWEVER, IT NO LONGER INCREMENTS TTLNUM, THE NUMBER OF RECORDS*
C  READ. THAT IS LEFT TO THE MAIN PROGRAM. AS A RESULT, RDCARD    *
C  DOES NOT NEED (AND, THEREFORE, IS NOT GIVEN ACCESS TO) THE     *
C  COMMON AREA TTLS CONTATINING THE THREE TOTALS.                 *
C*****************************************************************
      SUBROUTINE   RDCARD (FL)
      IMPLICIT     NONE
      REAL         MWT,MPT,BPT
      INTEGER*4    THISID,PREVID
      INTEGER*2    YR,BITTER
      CHARACTER    TRADE*20,EMPRCL*25,STATE*3
      LOGICAL      FL
      COMMON       /REC/THISID,MWT,MPT,BPT,BITTER,PREVID
      COMMON       /CHARS/TRADE,EMPRCL,STATE

      PREVID = THISID
      FL = .FALSE.
      READ (*,15,END=99) THISID,MWT,YR,TR,EMP,ST,MPT,BPT,BITTER
   15 FORMAT (IS,F6.2,I4,A20,A25,A3,F5.1,F6.1,I1)
      RETURN
   99 FL = .TRUE.
      RETURN
      END
```

**(c)**

**FIGURE 18.3** **(b)** Initialization Routine for Example 18.2. **(c)** RDCARD Subroutine for Example 18.2.

**Example 18.2** Note that it was necessary in Example 18.1 to declare the entire common block in every subroutine regardless of how many or how few of the variables a particular subroutine actually used. We shall reorganize the variables to illustrate how several common blocks might be used to reduce

this necessity. It should be pointed out that this is not necessarily the "best" or the "official" way to provide various program units with the data they need. The main purpose is to illustrate the organization and use of several common blocks. Concerns regarding the usefulness of shared variables in general are discussed in Section 18.3. The revised main program is shown in Figure 18.3, along with modifications of the subroutines shown for the previous example. Note that the names of the shared common blocks must match exactly in each program component.

If one or more named common blocks are used in a given program, that does not necessarily mean that a blank common area cannot be used. In any case, of course, the programmer must make sure that there is no more than one blank common area and that no variable is listed in more than one common area, named or blank.

FORTRAN accepts COMMON statements in which the lists of variables in a given block are separated by lists of variables in other blocks. This is a sloppy practice and should be avoided. However, since it is encountered from time to time in programs that you may be obligated to examine and change, this "capability" is mentioned so that you are aware of it. To illustrate, let us take the three named common areas declared in Figure 18.3(a):

```
COMMON          /REC/THISID, MWT, YR, MPT, BPT, BITTER, PREVID
COMMON          /CHARS/TRADE, EMPRCL, STATE
COMMON          /TTLS/TTLNUM, NUMACC, NUMREJ
```

Exactly the same results are obtained when the lists are combined like this:

```
  COMMON        /REC/THISID, MWT, YR, MPT, BPT, BITTER, PREVID,
 1              /CHARS/TRADE, EMPRCL, STATE,
 2              /TTLS/TTLNUM, NUMACC, NUMREJ
```

or like this:

```
  COMMON        /REC/THISID, MWT, YR, /CHARS/TRADE, /REC/MPT, BPT,
 1              /TTLS/TTLNUM, NUMACC, /CHARS/EMPRCL, STATE,
 2              /REC/BITTER, PREVID, /TTLS/NUMREJ
```

Note that FORTRAN just picks up where it left off in a given list. One instance in which this feature might be useful is a situation where the programmer is defining a named common area and has forgotten an item. Even there, it is clearer and easier to follow if the additional item is listed on a separate line. Moreover, once everything has been defined to the programmer's satisfaction, there is no good reason to leave a mess like the one above. It certainly is worth the small amount of extra effort it takes to rewrite the declarations so that they are clear and orderly.

## 18.2 INITIALIZATION OF VALUES IN COMMON BLOCKS

There are many applications in which a program unit needs to obtain values from a table as part of its processing. We saw earlier (Section 7.2.2) that the construction of such a table is handled quite conveniently by means of the DATA statement. When this usage is extended to procedures in which several program units need to consult a table at various times, then the DATA statement, combined with common storage, offers a particularly handy resource. The idea is straightforward: By setting up a table as a common block (or part of a common block), we can make that table available to any number of program units without the need for argument lists.

This can be done easily in FORTRAN, and it provides one of the most useful applications of common blocks and the COMMON statement. However, there are specific rules that must be followed in order to do this:

1. The DATA statement may not be used to assign values to variables listed in blank common.
2. When the DATA statement is used for variables in a named common block, the DATA statement must appear only in a special kind of subprogram called a *block data* subprogram.

This second rule needs some discussion: In order to make it possible to share a data table among several program units, FORTRAN provides a special type of subprogram whose

general form looks like this:

```
BLOCK DATA
  . . . . .
declarations for data type (REAL, INTEGER, etc.)
  . . . . .
COMMON statements
  . . . . .
DATA statement(s)
  . . . . .
END
```

The block data subprogram is special in several ways: It begins with the word BLOCK DATA and ends with an ordinary END statement. However, there are no actual processing statements in it; it consists entirely of declarations. Consequently, since it does no actual computations, it has no dummy argument list and no RETURN statement. In fact, a block data subprogram cannot contain anything but the following statements: Data type declarations, COMMON, DATA, DIMENSION, END, EQUIVALENCE, IMPLCIT, PARAMETER, SAVE, and, of course, a BLOCK DATA statement. It is not invoked by any other program unit in the program. Instead, it just sits there, providing FORTRAN with bookkeeping information. (As is true with other subprograms, its relative position in the sequence of program units is unimportant.)

A block data subprogram may or may not be named (unlike any other types of subprograms, which *must* be named). When a name is used, it appears in the first statement, as illustrated below:

```
BLOCK DATA    TABLES
```

Several block data subprograms may appear as part of the same program. However, no more than one of them can be unnamed.

**Example 18.3** To illustrate the use of the block data subprogram, we shall set up the following program outline: Input consists of a series of records, each containing a customer identification (columns 1–6), an order number (columns 11–16), a model number (column 17), and the number of units purchased (columns 21–24). For each input record, the program is required to compute a price, taxes, and shipping costs. These computations are based on information stored for each of the five models:

1. The price is computed from the base price, the model's length, width and height, and the number of units purchased.
2. The taxes are computed based on the price.
3. The shipping cost is based on the number of units and the model's length, width, height, and weight.

We can organize the product data as a collection of five-element arrays: Model number (MDLNUM), length (LGTH), width (WDTH), height (HT), weight (WT), and base price (BASEPR). In this way the subscript for a given model number also will be the subscript for that model's length, width, etc. When a group of arrays is set up this way, they are said to be *coupled arrays*.

The main program (Figure 18.4(a)) uses a subroutine (PRICER) to determine the total price for the goods and the tax, a second subroutine (SHIPPR) to determine the shipping costs, and a third subroutine (RDLOOK) to read an input record and look up the model number in the tables to find its subscript. Two other subprograms (OUTPRT and EOFPRC) print output and handle endfile processing, respectively.

Values for the tables are defined in a block data subprogram shown in Figure 18.4(b), and the RDLOOK subroutine is shown in Figure 18.4(c). It is worth pointing out again that the named COMMON statements appear in each of the program units, but the DATA statement initializing the table values appears only in the block data subprogram. (The details of the other subroutines are unimportant, and so the statements are not shown.)

```
C*****************************************************************
C                    EXAMPLE 18.3 - THE MAIN PROGRAM              *
C*****************************************************************
C  THIS PROGRAM PROCESSES A SERIES OF INPUT LINES, WITH EACH LINE*
C  REPRESENTING AN ORDER FOR A SPECIFIED NUMBER OF UNITS OF A    *
C  PARTICULAR PRODUCT, IDENTIFIED BY ITS MODEL NUMBER. PRODUCT   *
C  PRICES, TAXES, AND SHIPPING COSTS ARE COMPUTED USING DATA     *
C  STORED FOR EACH MODEL IN A TABLE WHOSE VALUES ARE DEFINED BY A*
C  DATA STATEMENT IN A BLOCK DATA SUBPROGRAM. THE VARIABLE NAMES *
C  ARE SELF-EXPLANATORY.                                         *
C*****************************************************************
      PROGRAM      EX1803
      IMPLICIT     NONE
      REAL         WT(10),LGTH(10),WDTH(10),HT(10),BASEPR(10)
      REAL         PRICE,TAX,SHPCST
      INTEGER*2    MDLNUM(10)
      INTEGER*2    MODEL,UNITS,INDEX
      INTEGER*4    CUSTID,ORDRNO
      LOGICAL      EOF

      COMMON       /MDLTBL/MDLNUM,WT,LGTH,WDTH,HT,BASEPR
      COMMON       /IN/CUSTID,ORDRNO,MODEL,UNITS,EOF,INDEX
      COMMON       /OUT/PRICE,TAX,SHPCST

      CALL RDLOOK

      DO WHILE (.NOT. EOF)
         CALL PRICER
         CALL SHIPPR
         CALL OUTPRT
      END DO

      CALL EOFPRC
      STOP
      END
```

**FIGURE 18.4 (a)** Main Program for Example 18.3.

## 18.3 THE SAVE STATEMENT

FORTRAN 77 extends our ability to share data among subprograms by enabling us to retain local values that normally are unavailable once a subprogram's processing has been completed. This is done by means of the SAVE statement whose general form is

SAVE list of names or simply
SAVE

When this statement appears in a subprogram, the values associated with the names in the list are retained after control is returned to the invoking program or subprogram. (A SAVE statement in a main program will not be rejected, but it is meaningless.) If a SAVE statement appears without a list, FORTRAN 77 takes this as a request to save all of the subprogram's local values. If we need to set up a subprogram in which locally developed values are to be retained for use in a subsequent invocation, the SAVE statement is a convenient way to do this.

A SAVE statement's list may include names of single-valued variables, arrays, and

```
C*****************************************************************
C                      BLOCK DATA SUBPROGRAM                     *
C*****************************************************************
      BLOCK DATA
      IMPLICIT      NONE
      REAL          WT(10),LGTH(10),WDTH(10),HT(10),BASEPR(10)
      REAL          PRICE,TAX,SHPCST
      INTEGER*2     MDLNUM(10)
      INTEGER*2     MODEL,UNITS,INDEX
      INTEGER*4      CUSTID,ORDRNO
      LOGICAL       EOF

      COMMON        /MDLTBL/MDLNUM,WT,LGTH,WDTH,HT,BASEPR
      COMMON        /IN/CUSTID,ORDRNO,MODEL,UNITS,EOF,INDEX
      COMMON        /OUT/PRICE,TAX,SHPCST

      DATA          MDLNUM/1,3,4,6,8/,
     1              WT/81.6,85.8,90.0,92.7,101.5/,
     2              LGTH/17.4,19.0,22.0,23.4,25.5/,
     3              WDTH/10.0,12.0,14.0,16.0,18.5/,
     4              HT/2.0,2.0,3.4,5.1,5.5/,
     5              BASEPR/17.50,18.25,19.50,21.75,23.90/

      END
```

*(Continued)*

**FIGURE 18.4 (b)** Block Data Subprogram for Example 18.3.

common blocks. A common block name must appear with a slash on either side. For example, the following fragment

```
SUBROUTINE      ZCOMP
IMPLICIT        NONE
REAL            B_VAC, M_FAC(24), F_SOT, X_PUNT
    .......
COMMON          /ALL_FAC/S_BET, P_GRID, T_RUF
SAVE            M_FAC, F_SOT, /ALL_FAC/
    .......
RETURN
END
```

arranges for the values in F_SOT, the entire array M_FAC, and all of the values in the common block named ALL_FAC to be retained after ZCOMP completes its processing. If the common block ALL_FAC also is declared in the main program (recall that it is legal to declare a common block in several subprograms without a corresponding declaration in the main program), the values in the block's variables are saved anyway, so the SAVE is redundant in that situation. However, inclusion of the block name in the SAVE list provides useful documentation, thereby improving the program's readability.

The following restrictions apply to the use of SAVE statements and SAVE lists:

1. A SAVE list may not include the name of a subprogram, a formal argument, or the name of a variable in a common block. When a block's name appears in a SAVE statement's list, the values in all of that block's variables will be saved.
2. If a common block is listed in a subprogram's SAVE statement, that block's name must appear in a SAVE statement in each subprogram that uses that block.

```
C*****************************************************************
C                              RDLOOK                             *
C*****************************************************************
C  RDLOOK READS AN INPUT LINE AND USES THE MODEL NUMBER GIVEN ON *
C  THAT LINE (MODEL) TO FIND THE LOCATION (INDEX) IN THE VARIOUS *
C  ARRAYS (WT,LGTH,WDTH,HT,BASEPR) CORRESPONDING TO THAT MODEL.  *
C  THE ASSUMPTION IN THIS CONSTRUCTION IS THAT THE MODEL NUMBER  *
C  ALWAYS WILL BE ONE OF THE FIVE (LEGAL) MODEL NUMBERS LISTED IN*
C  THE ARRAY MDLNUM.                                             *
C*****************************************************************
      SUBROUTINE  RDLOOK
      IMPLICIT    NONE
      REAL        WT(10),LGTH(10),WDTH(10),HT(10),BASEPR(10)
      INTEGER*2   MDLNUM(10)
      INTEGER*2   MODEL,UNITS,INDEX
      INTEGER*4   CUSTID,ORDRNO
      INTEGER*2   I
      LOGICAL     EOF

      COMMON      /MDLTBL/MDLNUM,WT,LGTH,WDTH,HT,BASEPR
      COMMON      /IN/CUSTID,ORDRNO,MODEL,UNITS,EOF,INDEX

      EOF = .FALSE.
      READ (*,15,END=299) CUSTID,ORDRNO,MODEL,UNITS
   15 FORMAT (I6,4X,I6,I1,3X,I4)
      DO I=1,5
        IF (MDLNUM(I) .EQ. MODEL) THEN
          INDEX = I
          GO TO 77
        ELSE
        END IF
      END DO

  299 EOF = .TRUE.
   77 RETURN
      END
```

**FIGURE 18.4 (c)** RDLOOK Subroutine for Example 18.3.

**18.4 USE AND MISUSE OF COMMON STORAGE**

At the beginning of the chapter, the point was made that shared storage may not always be as convenient as it looks. Certainly, the idea of eliminating long argument lists is an appealing one. However, there are some important factors in program construction that make it less tempting.

One of the greatest advantages of breaking a computational process into subprograms is that each subprogram can be treated as a separate unit. Not only is it possible to develop and perfect that unit by itself; this approach also offers the opportunity to use such a unit in many different programs simply by "plugging" it into any program that needs the kind of processing it is designed to perform. (This was discussed at some length in Chapter 11.) The ease with which a particular subprogram can be installed in various programs depends on how independent that subprogram is. If a subprogram has to be changed and reshaped for each program in which it is to be incorporated, its usefulness as a building block may decrease to a point where the benefits of the approach are wiped out.

The same holds true when we look at this issue the other way around. If we have a subprogram that is known to work properly, there is every reason to resist the idea of changing anything in it in order to make it fit properly into another program. Consequently, if we have to make a choice in such a situation, we might prefer to change the program (which is still developing) so that it will accept the (already developed) subprogram. If we do not have to change anything, then the subprogram is truly an independent building block that does not "know" anything (or have to "know" anything) about the program with which it will be used. This type of independence is called *functional isolation* (very fancy), and it is a good thing to aim for when designing program units.

In general, the use of shared storage works against this independence because it ties a subprogram to the organization and contents of a particular program's common areas. This makes it more difficult to apply that subprogram to other uses. There also is another problem: If a program still is under development and the common areas are subject to change and refinement, then every time there is a change in the structure or contents of a common block, that change has to be carried into every program unit sharing that block. Even with the greatest care, it is easy to overlook something, thereby throwing the program out of balance. As pointed out in Chapters 11 and 12, when a subprogram is designed to expect an argument list, that unit is not at all "concerned" with the way those arguments are produced or where they come from. The organizational simplicity that this independence encourages often is worth the bother of having to supply an argument list every time a subprogram is invoked.

With these considerations in mind, we can indicate some general guidelines as to when shared storage can be used to advantage. Bear in mind that these cannot be hard and fast rules. In every circumstance, they must be considered together with your judgment in determining what it is that will result in the simplest, clearest, and most useful computational process:

1. As illustrated in Example 18.3, common storage is an effective and convenient way of making a table of values available to several program units that need to refer to it.
2. When a subprogram is invoked from several different places in a program, and each invocation is made with the same list of arguments, it is reasonable to replace the argument list (and the corresponding dummy argument list) with one or more common areas.
3. The type of situation just described often means that the particular program may be so specialized that there is little chance for its units to be of general use elsewhere. If the programmer is strongly convinced that this is the case, then the use of shared data (instead of argument lists may be warranted.)

## 18.5 SUMMARY

Data may be shared among a program's units by setting up *common blocks*, i.e., organized groupings of storage that are made accessible to more than one program unit. Common blocks may be *named* or *unnamed*. Either type is declared by means of a COMMON statement:

COMMON *name,name*,etc.

for unnamed (*blank*) common, and

COMMON /*blockname*/*name,name*,etc.

for named (*labeled*) common. The variables listed in a COMMON statement are defined by ordinary type declarations, with the COMMON statement indicating the order in which they are to be stored in the common block. These variables are made available to the program

units (i.e., a main program and its associated subprograms) needing them by including a copy of the appropriate type declarations and COMMON statements in each program unit. This replaces the argument list that otherwise would serve as the conduit for the data. In general terms, then, use of shared storage trades the flexibility of an argument list for the convenience of not having to use one.

The following general rules govern the use of common storage:

1. No more than one area of blank common may be defined for any program.
2. Any number of named common areas may be defined for a program, but the names must be unique.
3. Both single-valued variables and arrays may be included in a common block, but a particular variable may be listed only in one common block.
4. A variety of data types may be included in any order in a given common block.
5. Descriptions of the same common block (placed in different program units) need not use the same names for the variables in given positions. However, their data types should match.
6. Initial values may be assigned to shared variables by using the DATA statement, but this must be done within the following restrictions:
   (1) The DATA statement cannot be used with variables in blank common.
   (2) When a DATA statement is used with variables in named common areas, the declarations may appear only in a *block data subprogram*. This special subprogram has the general form

```
BLOCK DATA      [name]
   .....
declarations
   .....
END
```

   (The designation [*name*] indicates that a block data subprogram may be named or unnamed. No more than one unnamed block data subprogram may appear as part of any program. This type of subprogram contains nothing but declaration statements.

**PROBLEMS**

1. Show the organization of the common region as a result of each of the following sequences. (Some may be illegal; indicate the ones that are and give the reasons):

```
(a) REAL              VTS, RN, CHK, ISP
    INTEGER*2         CR4, NTS
    COMMON            VTS, CR4, RN, CHK, NTS, ISP
(b) REAL              AMP, PRT, JTH
    INTEGER*2         ARTH, BRS
    CHARACTER*4       INVK
    COMMON            ARTH, INVK, AMP, JTH, PRT, ARTH, BRS
(c) REAL              CRK(4), BL, SETS(4)
    INTEGER*4         MPT, DL(3), R
    CHARACTER*2       MEND
    COMMON            CRK, DL, SETS, MPT, BL, MEND, R
(d) REAL              SLD(2,3), R2(3), NQ
    INTEGER*2         APS(3,2), KTS
    COMMON            SLD, APS, R2, KTS, NQ
(e) INTEGER*2         ROAR*2, 4, 3)
    REAL              FSD
    CHARACTER*10      NM
    COMMON            NM, ROAR, FSD
```

```
(f) REAL          GDR(3,2,3),KH,TRS
    INTEGER*4     K1,LTL,AHS
    COMMON        TRS,KH(7),K1,GDR,AHS(3,2),LTL
(g) REAL*8        BIGY(2,4),RES
    REAL          A8,B9
    INTEGER*2     OBS(2,4),P1
    INTEGER*4     BIGW
    LOGICAL       SW1,SW2
    COMMON        RES,A8,A9,BIGY,OBS,SW1,BIGW
(h) REAL          GRPH,TM(0:6),FD
    INTEGER*2     HT(-3:3,2),TTL
    COMMON        TM,FD,HT,TTL,GRPH
```

2. Write the appropriate declarations to set up each of the following common blocks:

   (a) An unnamed block containing three real single-valued variables DMAX, DMIN, and DMID.
   (b) An unnamed block containing
   a real single-valued variable VCRF
   a 24-element one-dimensional integer array named NGR
   four real single-valued variables DW, F3, YTU, and IJ8
   (c) An unnamed block containing
   a 3 × 2 array of integers named UPR
   two 14-element one-dimensional real arrays C1 and C2
   two real single-valued variables named FS and Q5
   a 12-element real array named DWN with −3 as the first subscript
   (d) An unnamed block containing
   a 32-character string named HDG
   a 2 × 4 array of 7-character strings named VERF
   four 11-character strings named COM, VRV, G5 and DFR
   (e) Three blocks named CM1, CM2, and CM3, respectively, each containing
   a real variable named R1, R2, or R3, respectively
   a 4 × 3 integer array named I1, I2 or I3, respectively
   a 3 × 4 double precision array named D1, D2 or D3, respectively
   (f) A block named CAV containing
   six 12-element real arrays named A1 through A6
   A block named TRE containing
   12 6-element integer arrays named J1 through J12
   (g) A block named BRG containing
   four real variables named D, G, HT and UR
   a 2 × 5 real array named SIDES with the
   row subscripts beginning at 0 and the
   column subscripts beginning at −3
   An unnamed block containing
   a 4-element array of 5-character strings named QER
   three 17-character strings named W1, W2 and W3

3. Based on the description given in the text for Example 18.3, write the *declarations* (including the appropriate COMMON statements) for the subroutines PRICER and SHIPPR.

4. Take a look at Problem 27 in Chapter 11. Based on your analysis of that problem and its requirements:

   (a) Are there advantages to using shared storage for that problem? If there are, indicate which variables you would make common, and give reasons for your choices.
   (b) If you decided that shared variables would improve the program, rewrite it in accordance with your answer in (a).

5. Rewrite Example 12.2 so that it takes advantage of the convenience of shared storage. If you choose to keep certain variables out of common blocks (i.e., you continue to supply them as arguments), give your reasons for doing this.

6. Rewrite Problem 12 in Chapter 12 so that there are no argument lists.

   (a) How does this affect the function specified in Problem 11 of Chapter 12?
   (b) Do you consider the revised version of this program to be an improvement over the original? Why or why not?

7. Revise Problem 15 or 16 in Chapter 12 so that it takes advantage of common storage.

8. Revise Problem 18 in Chapter 12 so that it takes advantage of common storage. Describe why the situation with this type of table search is similar to or different from that described in Example 18.3.

9. Revise the program in Problem 20 of Chapter 12 so that it uses shared storage wherever you think it improves the program. Compare the table usage in this program with that in Problem 8 and Example 18.3.

10. Modify the program in Example 18.3 so that it no longer needs to be guaranteed that the model number specified on a particular input record is a legal (recognizable) model number. It is up to you how your program responds to illegal model numbers.

11. Refer to Example 18.3 again. Everything is fine until, one Thursday morning, here comes the manufacturer of these items (whatever they are) and decides to add two more model numbers, along with their weights, lengths, etc. Oboy.

    (a) Describe what has to be done to each of the program units in the program `EX1803` in order to accommodate this change.
    (b) Would the job have been any easier if we had not used shared storage? Give the reasons for your answer.

# 19
# *Logical Variables*

Our previous use of logical variables makes it unnecessary to introduce them anew. Consequently, this chapter concentrates on logical expressions and input/output operations with logical data.

To review, recall that logical variables are named like other FORTRAN variables and are declared with the LOGICAL statement. A logical variable in HP FORTRAN 77, when declared without an explicit length designator, occupies one word of storage (16 bits on HP's 16-bit computer, or 32 bits on HP's 32-bit computer). The compilers for either type of HP computer can be forced to allocate a specified length by using the LOGICAL*2 declaration (16 bits) or LOGICAL*4 declaration (32 bits). This has no effect on logical operations. Such explicit declarations are useful in aligning COMMON areas and setting up equivalences among variables. Since logical data are organized and treated just like integer data, character data, or any other type, we can set up logical arrays, initial values, and parameters using standard techniques. For example, the statements

```
LOGICAL*2          SETSW(2,3)
DATA               SETSW /.FALSE.,.TRUE.,.FALSE.,3*.TRUE./
```

reserve storage for a six-element array named SETSW in which SETSW(1,1) and SETSW(1,2) are initialized to .FALSE., and the other four elements are initialized to .TRUE..

## 19.1 LOGICAL OPERATIONS AND EXPRESSIONS

As it is true with other types of information, the manipulation of logical data is organized around the *logical expression*. This section examines the ways in which these expressions can be built and the rules by which they are processed.

### 19.1.1 Basic Logical Operations

The simplest logical expression consists of a single logical constant or variable. For example, the second statement in the following sequence

```
LOGICAL*2      WHICH,HOW
WHICH = .TRUE.
HOW = WHICH
```

sets logical variable WHICH to the constant .TRUE., and the third statement assigns the value in WHICH to the logical variable HOW.

Logical constants or variables also can serve as building blocks in the construction of more extensive logical expressions. The way these expressions are formed is by connecting constants and variables with combinations of FORTRAN 77's five *logical operators* (OR, AND, NOT, EQV and NEQV). This enables the programmer to produce an endless range of expressions, each of which will result in a final value of .TRUE. or .FALSE..

**19.1.1.1 The OR Operation** The OR operation (written in FORTRAN as .OR.) combines two logical values to produce a result of .TRUE. or .FALSE. according to the following rule:

> If both values are .FALSE., the resulting value is .FALSE.; all other combinations produce a result of .TRUE..

An effective way to summarize the action of the OR operation (and other logical operations) is by means of a simple display called a *truth table*. This is nothing more than an organized list showing the outcomes of the OR operation (or whatever activity is being examined) applied to all possible combinations of logical values. This is easy because each logical variable can have only one of two possible values. Thus, assuming variables SW1, SW2 and OUT have been declared as logical variables, and that SW1 and SW2 have values in them, the statement

```
OUT = SW1 .OR. SW2
```

has only four possibilities:

1. SW1 and SW2 both may be .TRUE.;
2. SW1 may be .TRUE. and SW2 may be .FALSE.;
3. SW1 may be .FALSE. and SW2 may be .TRUE.;
4. SW1 and SW2 may both be .FALSE..

When we apply the rule stated above for the OR operation to these four possibilities, we see that three of the four outcomes will be .TRUE.; only when SW1 and SW2 are both .FALSE. will a value of .FALSE. be stored in variable OUT. The truth table for this expression, then, would look like this:

| A | B | A .OR. B |
|---|---|---|
| .TRUE. | .TRUE. | .TRUE. |
| .TRUE. | .FALSE. | .TRUE. |
| .FALSE. | .TRUE. | .TRUE. |
| .FALSE. | .FALSE. | .FALSE. |

Since the final result is stored directly in OUT (i.e., without further processing), the same table describes the properties of the entire statement:

| A | B | OUT = A .OR. B |
|---|---|---|
| .TRUE. | .TRUE. | .TRUE. |
| .TRUE. | .FALSE. | .TRUE. |
| .FALSE. | .TRUE. | .TRUE. |
| .FALSE. | .FALSE. | .FALSE. |

As soon as the expression becomes a little longer, the number of possible combinations goes up, and the truth table grows accordingly. However, the idea still is simple. For instance, suppose we use a third logical variable SW3 (assuming again that it has a value in it) and we write the statement

```
OUT = SW1 .OR. SW2 .OR. SW3
```

Since our expression now involves three variables, each of which can have one of two values, there are 2**3 or 8 different possibilities. As a result, our truth table will be twice as long as it was before (Table 19.1). For this table, we have evaluated the first part of the expression (A .OR. B) and applied the OR operation to that outcome and the variable C. As a result, the overall expression can have a value of .FALSE. only when all three variables are .FALSE..

**Table 19.1**

| A | B | A . OR. B | C | OUT=A . OR. B . OR. C |
|---|---|---|---|---|
| . TRUE. | . TRUE. | . TRUE. | . TRUE. | . TRUE. |
| . TRUE. | . TRUE. | . TRUE. | . FALSE. | . TRUE. |
| . TRUE. | . FALSE. | . TRUE. | . TRUE. | . TRUE. |
| . TRUE. | . FALSE. | . TRUE. | . FALSE. | . TRUE. |
| . FALSE. | . TRUE. | . TRUE. | . TRUE. | . TRUE. |
| . FALSE. | . TRUE. | . TRUE. | . FALSE. | . TRUE. |
| . FALSE. | . FALSE. | . FALSE. | . TRUE. | . TRUE. |
| . FALSE. | . FALSE. | . FALSE. | . FALSE. | . FALSE. |

**19.1.1.2 The AND Operation** The AND operation (written in FORTRAN as . AND. ) operates on two logical values in accordance with the following rule:

> If both logical values are . TRUE. , the AND operation produces a result of . TRUE. ; otherwise the resulting value is . FALSE. .

Thus, three of the four possible combinations result in a value of . FALSE. . This is seen in the following truth table. (As before, we assume that SW1, SW2 and OUT are logical variables, with values having been assigned previously to SW1 and SW2:)

| SW1 | SW2 | OUT = SW1 . AND. SW2 |
|---|---|---|
| . TRUE. | . TRUE. | . TRUE. |
| . TRUE. | . FALSE. | . FALSE. |
| . FALSE. | . TRUE. | . FALSE. |
| . FALSE. | . FALSE. | . FALSE. |

**19.1.1.3 The NOT Operation** The NOT operation (written in FORTRAN as . NOT. ) is a unary operator that simply produces the reverse of the single value on which it operates. For example, the truth table for the statement

```
OUT = .NOT. SW1
```

produces the following truth table:

| SW1 | OUT = . NOT. SW1 |
|---|---|
| . TRUE. | . FALSE. |
| . FALSE. | . TRUE. |

**19.1.1.4 The EQV Operation** The logical equivalence operation (written in FORTRAN as . EQV. ) operates on two logical values to produce a result in accordance with the following rule:

> If both logical values are the same (regardless of what they are), logical equivalence produces a value of . TRUE. ; otherwise a value of . FALSE. results.

Using SW1, SW2 and OUT as we did before, here is what the truth table looks like:

| SW1 | SW2 | OUT = SW1 . EQV. SW2 |
|---|---|---|
| . TRUE. | . TRUE. | . TRUE. |
| . TRUE. | . FALSE. | . FALSE. |
| . FALSE. | . TRUE. | . FALSE. |
| . FALSE. | . FALSE. | . TRUE. |

**19.1.1.5 The NEQV Operation** The logical nonequivalence operation (written in FORTRAN as .NEQV.) operates on two logical variables to produce a result that is directly opposite to that associated with the EQV operation:

> If both logical variables are the same (regardless of what they are), logical nonequivalence produces a value of .FALSE.; otherwise a value of .TRUE. results.

Using variables SW1, SW2 and OUT as we have been doing all along, the truth table looks like this:

| SW1 | SW2 | OUT = SW1 .EQV. SW2 |
|---|---|---|
| .TRUE. | .TRUE. | .FALSE. |
| .TRUE. | .FALSE. | .TRUE. |
| .FALSE. | .TRUE. | .TRUE. |
| .FALSE. | .FALSE. | .FALSE. |

HP FORTRAN 77 accepts EOR and XOR as synonyms for NEQV.

### 19.1.2 Priorities in Logical Expressions

As is the case with arithmetic expressions, FORTRAN makes sure that it handles logical expressions consistently by following a carefully defined set of rules. These set the priorities for logical operations just as the rules for arithmetic priorities (Section 5.2.2) regulate the processing of numerical expressions:

1. Logical expressions are processed from left to right. FORTRAN works its way through an expression as many times as necessary, performing selected activities based on predefined priorities assigned to each type of operator;

2. Priorities of logical operations are set at four levels:

(1) NOT operations (highest);
(2) AND operations;
(3) OR operations;
(4) EQV and NEQV operations (lowest).

3. Priorities in logical expressions can be regulated by the use of parentheses in exactly the same way as is true for arithmetic expressions (Section 5.1.1.4).

Thus, in the statement

```
OUT = SW1 .OR. SW2 .AND. SW3
```

the AND operation will be performed first (on SW2 and SW3). Consequently, the OR operation will be applied to SW1 and the result of SW2 .AND. SW3. We see, then, that FORTRAN's priority rules make it behave as if we had written

```
OUT = SW1 .OR. (SW2 .AND. SW3)
```

the corresponding truth table is shown below:

| SW2 | SW3 | SW2 .AND. SW3 | SW1 | OUT=SW1.OR.SW2.AND.SW3 |
|---|---|---|---|---|
| .TRUE. | .TRUE. | .TRUE. | .TRUE. | .TRUE. |
| .TRUE. | .FALSE. | .FALSE. | .TRUE. | .TRUE. |
| .FALSE. | .TRUE. | .FALSE. | .TRUE. | .TRUE. |
| .FALSE. | .FALSE. | .FALSE. | .TRUE. | .TRUE. |
| .TRUE. | .TRUE. | .TRUE. | .FALSE. | .TRUE. |
| .TRUE. | .FALSE. | .FALSE. | .FALSE. | .FALSE. |
| .FALSE. | .TRUE. | .FALSE. | .FALSE. | .FALSE. |
| .FALSE. | .FALSE. | .FALSE. | .FALSE. | .FALSE. |

We shall complicate things a little further by changing the statement so that it reads as follows:

```
OUT = SW1 .OR. .NOT. SW2 .AND. SW3
```

Applying the priority rules again, we see that the NOT operation will be performed first, followed by AND, with the OR operation bringing up the rear. Thus, FORTRAN behaves as if we had written the statement as

```
OUT = SW1 .OR. ((.NOT. SW2) .AND. SW3)
```

Now the truth table looks like Table 19.2. The (1) beneath the subexpression (.NOT. SW2. AND. SW3) is just a shorthand way of referring to that group of terms; we use it for convenience in the last column heading.

As a final complication, we shall add one more logical variable (SW4) and one more operation (EQV):

```
OUT = SW1 .OR. .NOT. SW2 .EQV. SW4 .AND. SW3
```

The evaluation proceeds as follows:

1. The NOT operation is performed on SW2. (For purposes of illustration, let us say that the result is stored temporarily in T1;
2. The AND operation is performed on SW4 and SW3. (The result is stored temporarily in T2);
3. The OR operation is performed on SW1 and T1. (The result is stored temporarily in T3);
4. The EQV operation is performed on T3 and T2. Since the result of this operation is the desired value, it is stored in the final destination, OUT.

Thus, the result is computed as if the statement had been written as

```
OUT = (SW1 .OR. (.NOT. SW2)) .EQV. (SW4 .AND. SW3)
```

To make sure this is clear, we shall repeat the process, this time with specific values for the variables. Let us say that SW1 and SW3 have values of .TRUE., and SW2 and SW4 have values of .FALSE.. Now we shall follow it through again:

1. A value of .TRUE. (i.e., .NOT. SW2) is stored in T1;
2. A value of .FALSE. (SW4 .AND. SW3) is stored in T2;
3. T3 receives a value of .TRUE. (i.e., SW1 .OR. T1);
4. Finally, the EQV operation on T3 (.TRUE.) and T2 (.FALSE.) produces a value of .FALSE., and that value is stored in OUT.

**Table 19.2**

| SW2 | .NOT. SW2 | SW3 | .NOT. SW2 .AND. SW3 (1) | SW1 | OUT=SW1 .OR. (1) |
|---|---|---|---|---|---|
| .TRUE. | .FALSE. | .TRUE. | .FALSE. | .TRUE. | .TRUE. |
| .TRUE. | .FALSE. | .FALSE. | .FALSE. | .TRUE. | .TRUE. |
| .FALSE. | .TRUE. | .TRUE. | .TRUE. | .TRUE. | .TRUE. |
| .FALSE. | .TRUE. | .FALSE. | .FALSE. | .TRUE. | .TRUE. |
| .TRUE. | .FALSE. | .TRUE. | .FALSE. | .FALSE. | .FALSE. |
| .TRUE. | .FALSE. | .FALSE. | .FALSE. | .FALSE. | .FALSE. |
| .FALSE. | .TRUE. | .TRUE. | .TRUE. | .FALSE. | .TRUE. |
| .FALSE. | .TRUE. | .FALSE | .FALSE. | .FALSE. | .FALSE. |

(Remember: We are processing the unparenthesized expression; if the parentheses physically were there, the processing sequence would be somewhat different, but the result would be the same.) The statement made in Chapter V applies just as well here: If there is any doubt about what FORTRAN will do when processing a particular logical expression, it always is a good idea to remove that doubt by using parentheses.

### 19.1.3 Relational Operators and Logical Expressions

We have been using another form of logical expression as the basis for decision-making in the IF statement. For instance, if X and Y are numerical variables, we are well aware that we can construct a decision mechanism as follows:

```
IF (X**2 .LE. 3.0*Y+17.8) THEN
   .......
      action1
   .......
ELSE
.....
      action2
   .......
END IF
```

Recall that .LE. is one of FORTRAN's six *relational operators*, and that the choice of action1 or action2 is based on whether the outcome of the comparison is .TRUE. or .FALSE.. Consequently, the IF statement's operation is based on the evaluation of a logical expression. In fact, if we go back to one of the earlier program examples (say, Example 17.1), we see that we could write the comparison

```
IF (EOF .EQ. TRUE) etc.
```

simply as

```
IF (EOF) etc.
```

since EOF is a logical variable and, therefore, forms a logical expression when used by itself. (This form is used later on, in Example 18.1.)

Thus, a comparison forms a logical expression and, as such, it can act as a component of a more extensive logical expression. For example, we have used such combinations to build multiple comparisons such as those developed in Chapter 9. For instance, the comparison

```
IF (SEX .EQ. FEMALE .AND. 1979-BRTHYR .GE. 28) etc.
```

developed there is seen to be nothing more than two logical expressions connected by a logical operator (.AND.) to form a single, more extensive logical expression.

It is fairly easy to see the priorities that are at work here: Since the overall expression must produce a final value of .TRUE. or .FALSE., its ingredients must be expressed as logical values that can be used to develop the final result. Consequently, each of the relational expressions will be evaluated first, thereby producing the logical values. Then those values can be processed by the logical operator for the final result. In general, then, *arithmetic operations will be performed (in a given expression) before relational operations, and relational operations will be performed before logical operations.* For example if X and Y are numerical variables, the expression associated with the following IF statement

```
IF (X**2 .GE. 2.8*(Y+24.0) .AND. SW1 .OR. SW2) etc.
```

will be evaluated as follows:

1. A numerical value is developed for X**2 and stored temporarily. (We shall call the temporary storage place V1.)
2. A numerical value is developed for 2.8*(Y+24.0) and stored in V2.
3. V1 and V2 are compared (using the .GE. operation) and the outcome (.TRUE. or .FALSE.) is stored in T1.
4. The AND operation is performed on T1 and SW1, with the result stored in T2.
5. The OR operation is performed on T2 and SW2, with the result stored in T3. It is T3's value, then, that determines how the IF statement will go.

**Example 19.1** To illustrate the use of simple logical expressions, we shall write a subroutine named SWITCH that operates as follows: SWITCH provides a general logical switch that compares two logical variables (the first two arguments) and sets the value of a third logical variable (the third argument) based on this comparison. The rules for the comparison are specified by the fourth argument, so that they can change (if desired) with each invocation.

When we compare two logical variables, there are only four possible combinations that can occur: Both values are .TRUE., both are .FALSE., the first is .TRUE. and the second is .FALSE. and vice versa. Consequently, the rules for such a two-way comparison can be expressed in terms of four logical values. These are conveniently organized as a four-element logical array, and that is exactly what the fourth argument is. Each of its elements specifies the value to be assigned to the third argument for a particular combination of values in the first two arguments. To illustrate, suppose the fourth argument is named RULE and there are values of .TRUE. in RULE(1) and RULE(2), .FALSE. in RULE(3), and .TRUE. in RULE(4).

1. RULE(1) defines the result (.TRUE.) to be assigned to the third argument if the first two arguments are .TRUE.;
2. RULE(2) says to assign .TRUE. to the third argument if the first argument is .TRUE. and the second is .FALSE.;
3. RULE(3) says to assign a value of .FALSE. to the third argument if the first argument is .FALSE. and the second is .TRUE.;
4. RULE(4) says to assign a value of .TRUE. to the third argument if both the first and second arguments are .FALSE..

Thus, if we specified values of .TRUE., .TRUE., .TRUE., and .FALSE. for RULE(1) through RULE(4), respectively, we would be duplicating the action of the OR operation.

The heart of the subroutine is a set of nested IF-THEN-ELSE constructions that determine the particular combination of values found in the first two arguments. This is seen in the statements given in Figure 19.1. An invocation of SWITCH could look like this:

```
i
.......
LOGICAL SW1, SW2, SW3, TEST(4)
.......
TEST(1) = .TRUE.
TEST(2) = .TRUE.
TEST(3) = .FALSE.
TEST(4) = .FALSE.
.......
  SW1, SW2 receive values (by computation or input/output)
CALL SWITCH (SW1, SW2, SW3, TEST)
.......
```

### 19.1.4 Logical Operations on Numerical Variables

When a logical variable is declared in HP FORTRAN 77, one or two words of storage are reserved, but only a single bit is used to represent the value (1=.TRUE., 0=.FALSE.). The settings of the other bits are immaterial and are beyond the programmer's direct

```
C*****************************************************************
C                              SWITCH                            *
C*****************************************************************
C  THIS SUBROUTINE PROVIDES THE FRAMEWORK FOR A GENERAL LOGICAL  *
C  SWITCH BY WHICH THE PROGRAMMER CAN IMPLEMENT ANY ONE OF 16    *
C  SETS OF RULES GOVERNING THE OUTCOME OF A LOGICAL COMPARISON   *
C  BETWEEN TWO (LOGICAL) VALUES. THERE ARE FOUR ARGUMENTS:       *
C   ARG1 __ARG2: THE LOGICAL VALUES TO BE COMPARED;              *
C   ARG3: THE LOGICAL VARIABLE RECEIVING THE RESULT;             *
C   ARG4: A 4-ELEMENT LOGICAL ARRAY DEFINING THE SET OF RULES TO *
C         BE USED FOR THIS INVOCATION:                           *
C     ARG4(1): THE VALUE TO BE ASSIGNED TO ARG3 IF ARG1 AND ARG2 *
C              BOTH ARE .TRUE.;                                  *
C     ARG4(2): THE VALUE TO BE ASSIGNED TO ARG3 IF ARG1 IS .TRUE.*
C              AND ARG2 IS .FALSE.;                              *
C     ARG4(3): THE VALUE TO BE ASSIGNED TO ARG3 IF ARG2 IS .TRUE.*
C              AND ARG1 IS .FALSE.;                              *
C     ARG4(4): THE VALUE TO BE ASSIGNED TO ARG3 IS ARG1 AND ARG2 *
C              BOTH ARE .FALSE..                                 *
C*****************************************************************
      SUBROUTINE     SWITCH (ARG1,ARG2,ARG3,ARG4)
      IMPLICIT       NONE
      INTEGER*2      I
      LOGICAL*2      ARG1,ARG2,ARG3,ARG4(4)

      IF (ARG1 .AND. ARG2) THEN
        I = 1
      ELSE IF (ARG1 .AND. (.NOT. ARG2)) THEN
             I = 2
           ELSE IF (ARG2 .AND. (.NOT. ARG1)) THEN
                  I = 3
                ELSE
                  I = 4
      END IF

      ARG3 = ARG4(I)
      RETURN
      END
```

**FIGURE 19.1** Statements for the Subroutine of Example 19.1.

control. Consequently, the use of logical variables for bit manipulation receives few awards for efficiency.

To overcome this difficulty, HP FORTRAN 77 allows the use of logical operations on integer variables. In this context the variables, though declared as INTEGER*2 or INTEGER*4, can be treated as strings of individual bits and subjected independently to the action of the logical operators. This facility is strengthened by several built-in functions that enable the HP FORTRAN 77 programmer to manipulate one or more individual bits in an integer variable without affecting the others. Details were examined in Chapter 5. The feature is mentioned here to complete the discussion of logical operations.

Logical values may be read and written just like any other FORTRAN data type. Both list-directed and edit-directed forms are acceptable, and logical values may be included in unformatted records by means of the regular techniques for preparing and processing such records (Chapter 17).

### 19.2.1 Logical Data Input

FORTRAN recognizes two basic forms for logical input values: We can use either the full symbols (.TRUE. or .FALSE.) or the more concise T or F.

**19.2.1.1 List-Directed Logical Input** Either of the forms given above can be used in list-directed input. For instance, if TEST is a four-element logical array, the statement

```
READ (1,*) TEST
```

will work with an input line such as

```
T,F,F,T
```

or

```
.TRUE., F, .FALSE., T
```

or any other such combination. Note that blanks have no effect.

**19.2.1.2 Edit-Directed Logical Input** The same forms can be used for edit-directed input. Regardless of the form used, the data are described by the L-format:

L*w*

where *w* is an integer indicating the number of input columns. The simplest way to represent edit-directed logical input is to use T or F, in which case the format description is L1. Other forms also are legal, but they are clumsier, and efforts should be made to avoid them. They are shown here for the record, but their use is discouraged:

1. The T or F may be *preceded* by one or more blanks. Of course, when this is done, the value of *w* in the format description must match the length of the field. (FORTRAN will look through the field from left to right until it finds the T or F.)
2. The T or F may be *preceded* by a decimal point, and the decimal point, in turn, may be preceded by one or more blanks.
3. The constants .TRUE. and .FALSE. may appear as legal logical values, *preceded* by any number of blanks. Of course, if this form is used, a value of .TRUE. will have to be preceded by one blank so that it occupies the same number of columns as the .FALSE. value.
4. Other characters may appear as part of the field counted in the *w*. These can be any characters at all, since they will be ignored. However, in order for this to work, such extraneous characters must appear *after* the actual logical value. (This is a particularly awkward option, leading to all sorts of confusion; accordingly, there has to be a compelling reason for using it.)

These alternatives are illustrated in Table 19.3.

### 19.2.2 Output of Logical Data

FORTRAN offers less of a choice for the display of logical values than it does for their presentation as input. However, the available features are convenient, clear, and easy to use.

**Table 19.3** Edit-Directed Forms for Logical Input Values.

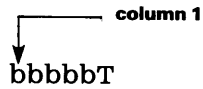
| *Input Value* | *Format Specification* | *Result* |
|---|---|---|
| T | L1 | OK |
| Tbb | L3 | OK |
| bbF | L3 | OK |
| b.F | L3 | OK |
| .bT | L3 | illegal; if there is a decimal, it must be just before the value. |
| .FALSE. | L7 | OK |
| b.TRUE | L7 | OK |
| bbb.FALSE. | L10 | OK |
| T9J | L3 | OK |
| 9JT | L3 | illegal; blanks or a decimal are the only preceding characters allowed. |
| bbb.FALSE.8bHT | L14 | OK |

**19.2.2.1 List-Directed Output of Logical Values** Logical values specified for output in list-directed ormat will appear as T or F, preceded by a specific number of blanks. As is true with numberic or character data, the number of blanks will be set inside the particular FORTRAN system and, therefore, beyond the programmer's direct control.

**19.2.2.2 Edit-Directed Output of Logical Values** The L-specification is used for output in much the same as was described for edit-directed logical input (Section 19.2.1.2). The basic difference is that FORTRAN uses only T or F for output. Thus, if logical variable SW1 has a value of .TRUE. the statements

```
    WRITE (*,16) SW1
16  FORMAT (1X,5X,L1)
```

produce a line of print that says

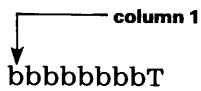



```
bbbbbT
```

If the value of *w* in the format specification is greater than 1, FORTRAN will pad to the left with *w*−1 blanks. For instance, if we change statement 16 above so that it says

```
16  FORMAT (1X,5X,L4)
```

there will be three additional blanks before the T is printed:




```
bbbbbbbbT
```

**19.3 SUMMARY** Logical variables (whose only possible values are .TRUE. or .FALSE.) may be named, declared, and initialized using the same rules that apply to other FORTRAN variables. Values are assigned to logical variables using the regular assignment statement. The *logical expression* to the right of the assignment operator (=) may consist of the following:

1. A logical constant or variable;

2. Logical constants or variables combined by the logical operators . AND. , . OR. , . EQV. , . NEQV. , and . NOT. ;
3. Pairs of arithmetic or character expressions combined by one of FORTRAN's relational operators . EQ. , . NE. , . LT. , . LE. , . GE. , or . GT. ;
4. Any grouping of logical expressions described above, connected by a logical operator.
5. Integer constants and/or variables connected by logical operators.

Any legitimate logical expression may be used as the test for an IF statement.

Logical data may be read or written either in list-directed or edit-directed format. The L-specification (L*w*) is used to edit logical data. Input values may appear either as . TRUE. , . FALSE. , T, or F. Output values (in either format) always will appear as T or F, preceded by zero or more blanks.

## PROBLEMS

1. Declare a 6 × 8 logical array named SETS and initialize the odd-numbered rows to . TRUE. and the even-numbered rows to . FALSE. .

2. Write a truth table for each of the following expressions or statements. Assume that all of the variables have been declared as LOGICAL, but do not assume that all of the expressions or statements are written correctly:
   (a) L1 . OR. . TRUE.
   (b) L2 . OR. . TRUE. . AND. L2
   (c) L3 = L1 . OR. . NOT. L2
   (d) L3 = L1 . AND. (. NOT. L2 . AND. . NOT. L3)
   (e) L1 . EQV. (L2 . AND. L3)
   (f) L4 = L4 . NEQV. L1
   (g) L3 = . NOT. L1 . AND. . EVQ. L2
   (h) L4 = L1 . AND. L2 . EQV. . NOT. (L3 . AND. L1)

3. Indicate what the final value will be for each of the following independent statements. Assume that all variables are declared as LOGICAL, and that they have the following values just before the statement is executed:

   L1, L3, and L5 are . TRUE. ; the rest are . FALSE. .

   (a) L3 = L3 . AND. . TRUE.
   (b) L5 = L1 . AND. L2 . AND. L3 . AND. L4
   (c) L4 = L4 . EQV. L1 . NEQV. L2 . OR. L3 . AND. . NOT. L5
   (d) L5 = (L1 . NEQV. L2) . OR. . NOT. (L2 . EQV. L3) . EQV. (L3 . AND. L4)

4. Indicate what the final value will be for each of the following independent statements. Assume that all variables whose names begin with L are declared as LOGICAL, and all the others are numerical. The variables have the following values just before each statement is executed:

   L1, L3, and L5 are . FALSE. ; the other logical variables are . TRUE.
   X = −70.5  Y = 11.2  S = 8.0  T = 6.5

   (a) L5 = L5 . AND. X . GE. Y
   (b) L5 = . TRUE. . AND. . NOT. (Y . LT. 17.1)
   (c) L3 = L2 . AND. X+Y
   (d) L4 = L2 . EQV. X+Y . LE. S**2
   (e) L3 = L1 . AND. L2 . NEQV. S+T . GE. ABS (X) /Y
   (f) L2 = L2 . OR. . NOT. (X*S . EQ. X**2/T . AND. L12

5. Rewrite the subroutine in Example 19.1 as a function.

6. Rewrite either the subroutine in Example 19.1 or the function specified in Problem 5 so that it processes a one-dimensional logical array.

7. Write a main program for the subprogram in Problem 6 so that the result is a program that reads and compares a collection of pairs of logical arrays. Values for each input pair of 30-element arrays are

recorded on a single line, with the first array's values in columns 1–30 and the second array's values in columns 41–70. The first pair of logical arrays is preceded by a special line that contains the four rules for the subprogram in columns 1–4.

8. Modify the program in Problem 7 so that the rules can be changed whenever desired. That is, a set of rules submitted for a particular pair of arrays will apply to all subsequent arrays until a new set is submitted.

9. Write a function named ANY whose single argument is a logical array. If any of the values in the argument is . TRUE. , the function returns a value of . TRUE. ; otherwise, it returns a value of . FALSE. .

10. Rewrite the subprogram in Problem 9 as a subroutine.

11. Write a function named ALL whose single argument is a logical array. If all of the values in the argument are . TRUE. , the function returns a value of . TRUE. ; otherwise, it returns a value of . FALSE. .

12. Rewrite the subprogram in Problem 11 as a subroutine.

13. Write a function named SOME that is invoked as follows:

    SOME (*arg1*,*arg2*)

    where *arg1* is a logical array and *arg2* is an integer value. This function includes the operations of the functions ANY and ALL in Problems 9 and 11. Specifically, SOME uses the value in *arg2* to determine the conditions under which it will return a value of . TRUE. : If there are at least *arg2* values of . TRUE. among *arg1*'s elements, SOME will return a value of . TRUE. ; otherwise, it will return . FALSE. .

14. Write the subprogram in Problem 13 as a subroutine.

15. The Department of Phrygian Studies at the University of East Clavicle would like to have a program that grades the final examination in its advanced course. This examination consists of fifty true-false questions. Each student's answers are recorded on a line containing the student's name in columns 1–20, student i.d. number in columns 25–30, and the fifty answers (T or F) in columns 31–80. The first student's data are preceded by a line in which the correct answers are given in columns 21–70. Each correct answer counts for two points. Your program should print a line for each student showing his or her i.d. number, name, answers, and total score. After all the data have been processed, the program is to print (on a separate page) the highest score, the number of students achieving that score, the lowest score, the number of students achieving that score, the average score, and the total number of students taking the test. Assume that every student answers every question.

16. Now, East Clavicle's Department of Ionian Studies wants a test scorer too. However, their system is a little different: They also give an examination of fifty true-false questions. However, instead of just crediting each correct answer with two points, they also want to deduct one point for every question not answered. This means that it is possible for some knucklehead to achieve (if that is the word) a negative score. Correct answers still are worth two points apiece, and missing answers are blank. This program is to produce exactly the same output specified for Problem 15.

17. Extend the processing in Problem 16 so that each line shows everything it did before, along with the number of questions missed, and the number of questions omitted.

18. Extend the processing in Problem 17 so that the summary shows everything it did before. In addition, it is to include a table containing one line of print for each question showing the question number, the number of students answering that question correctly, the number answering it incorrectly, and the number leaving it out.

# Appendix A
# *Complex Data*

FORTRAN provides a complete set of facilities for recognizing and processing complex numbers (i.e., numbers with real and imaginary components).

## A.1 DECLARATION OF COMPLEX VARIABLES

A complex constant consists of two numerical values: A real component and an imaginary component. These are expressed by enclosing them in parentheses and separating them by a comma. The real component always is first. Several examples follow:

| *FORTRAN Complex Constant* | *Value Represented* |
|---|---|
| (3.0, 4.5) | 3.0 + 4.5i |
| (−6.7, 2.0) | −6.7 + 2.0i |
| (8, −8) | 8 − 8i |
| (22.1, 0.0) | 22.1 + 0i |
| (0.0, −7.8) | 0.0 − 7.8i |

Complex variables are named like any other FORTRAN variables and are declard with the COMPLEX*8 or COMPLEX*16 data type. for example:

```
COMPLEX*8       X, NCL(8)
COMPLEX*16      GHOST(2, 4)
```

COMPLEX*8 reserves eight bytes (four words on HP's 16-bit computer or two words on HP's 32-bit computer). COMPLEX*16 reserves 16 bytes. For either declaration, the allocated storage is divided equally between the real and imaginary components.

Complex variables may be initialized with the DATA statement. Using the declarations given above, we can write something like

```
 DATA   X, NCL, GHOST/ (1.1, 1.0), 8*(2.5, 0.0),
1       4*(1.0, -1.0), 4*(-1.0, 1.0) /
```

## A.2 COMPUTATIONS WITH COMPLEX NUMBERS

Complex arithmetic in FORTRAN is conducted according to the following rules. (CV1 and CV2 are complex values consisting, respectively, of RL1+IL1i and RL2+IL2i):

| *Operation* | *Result Produced by FORTRAN* |
|---|---|
| CV1 + CV2 | (RL1+RL2, IL1+IL2) |
| CV1 − CV2 | (RL1−RL2, IL1−IL2) |
| CV1 * CV2 | (RL1*RL2−IL1*IL2, RL1*IL2+IL1*RL2) |
| CV1 / CV2 | ( (RL1*RL2+IL1*IL2) / (RL2**2+IL2**2), (RL2*IL1−IL2*RL1) / (RL2**2+IL2**2) ) |

A real value (say, RVAL) multiplied by a complex value (say, (RL1, IL1)) produces the complex value (RVAL*RL1, RVAL*IL1).

Complex values may appear in arithmetic expressions either with other complex values and/or with combinations of other numeric values. Since complex numbers are special, their appearance in an arithmetic expression will compel FORTRAN to convert everything in that expression to complex. for example, if CV1 is declared as COMPLEX, the assignment

```
CV1 = (2.0,3.0) + 4 + 6.5
```

will be processed by converting the real and integer values so that final evaluation will be performed as if the values were

```
(2.0,3.0) + (4.0,0.0) + (6.5,0.0)
```

and the result stored in CV1 will be (12.5,3.0), the equivalent of 12.5+3.0i. The only restriction is that complex values cannot appear in any expression in which a double precision value appears.

## A.3 BUILT-IN FUNCTIONS AND COMPLEX DATA

A specific group of FORTRAN's built-in functions can be applied to complex arguments. In addition, the CMPLEX function can be used to convert a single numerical argument (integer, real, or double precision) to a complex value in which the real portion will be taken from the argument's (converted) value and the imaginary portion will be 0.0. Alternatively, COMPLX can be used to process two numerical arguments, in which case they must both be of the same type. When this form is used, a complex number is formed with the real value taken from the first argument and the imaginary value taken from the second argument. For instance,

```
CMPLX (1,-2)
```

produces a complex value equivalent to 1.0–2.0i.

Two other functions are designed specifically for use with a single complex argument:

1. The AIMAG function delivers the imaginary part of a complex value, expressed as a real value. For instance, AIMAG(3.0,–4.0) delivers the real value –4.0.
2. The CONJG function delivers the conjugate of a complex value. Thus, CONJG(–7.0,8.0) is (–7.0,–8.0).

**Table A.1** Use of Built-in Functions with Complex Arguments
(Assume CV is a complex variable with value A + B*i*)

| Function | Value delivered by the Function |
|---|---|
| ABS(CV) | $\sqrt{A^2 + B^2}$ |
| SQRT(CV) | $\sqrt{A + Bi}$ |
| COS(CV) | $\cos(A \cosh B) - i \sin(A \sinh B)$ |
| EXP(CV) | $e^A(\cos B + i \sin B)$ |
| LOG(CV) | $\frac{1}{2}\log(A^2 + B^2) + i \tan^{-1}(B/A)$ |
| SIN(CV) | $\sin(A \cosh B) + i \cos(A \sinh B)$ |

Table A.1 summarizes the characteristics of the other built-in functions applicable to complex values.

## A.4 INPUT/OUTPUT OF COMPLEX DATA

Complex data can be read either in list-directed or edit-directed form. A list-directed complex value must appear as (real,imaginary). Both the parentheses and the separating comma must be present. Edit-directed complex values may appear as two consecutive numeric values of any type, described by two consecutive format specifications which may differ from each other. FORTRAN treats the first value as the real part and the second value as the imaginary part. For instance, suppose CV1 is declared as COMPLEX and we use the statement

```
READ(*,'(3X,F3.1,F2.0)') CV1
```

to read the following line:

```
bbb34587223BJ
```

FORTRAN will read and store a value of (34.5,87.0) in CV1. The F, E, D, or G specifiers can be used for edit-directed complex data.

Output facilities are consistent with those described for input.

# Appendix B
# *Representation of Characters in HP Systems*

## B.1 THE HP CHARACTER SET

HP systems are designed to recognize the ASCII code (American Standard Code for Information Interchange) for character data. In the ASCII coding system, a combination of seven 1s and 0s is used to represent each character. (Each ASCII character is stored in an eight-bit byte, but the eighth bit is handled automatically and does not affect consistent interpretation of the information.)

Under certain conditions, it is possible for a computer to "look" at a character in storage as a string of 1s and 0s rather than as a letter, or punctuation mark, or some other character. When this happens, each string can be treated (by the computer) as a number. In other words, each type of character has its own *internal numerical representation.* The order of these internal numerical values (for a given coding system) is called the *collating sequence.*

Since ASCII uses a string of seven 1s and 0s to represent each character, there are 128 possible types. Any of the combinations are "characters" in the sense that the computer uses them, and can tell them apart from the others. However, they are invisible. (When printed, they appear as blanks, but they really are not.) Computers use them for various internal control purposes. Table B.1 shows the visible ASCII characters along with one invisible character, i.e., the one and only genuine blank.

## B.2 THE FORTRAN CHARACTER SET

Table B.2 shows the ASCII representations for those characters that can be used in writing an HP FORTRAN 77 program. This does not mean that the other characters (i.e., the ones shown in Table B.1) are unavailable. Any character that can be represented can be used as *data* in a FORTRAN program. However, only the members of the FORTRAN character set can be used for the program statements themselves.

## B.3 SPECIAL FUNCTIONS FOR THE COLLATING SEQUENCE

FORTRAN 77 provides four functions that compare two character strings with regard to their relative positions in the ASCII collating sequence. Each of these functions requires two arguments (both character strings) and returns a value of . TRUE. if the outcome of the comparison is true, and . FALSE. if the outcome is false. These are summarized below using CH1 and CH2 to represent two character strings.

| *Function* | *Outcome* |
|---|---|
| LLT (CH1, CH2) | LLT returns . TRUE. if CH1 is lexically less than CH2 in the ASCII collating sequence. |
| LLE (CH1, CH2) | LLE returns . TRUE. if CH1 is lexically less than or equal to CH2 in ASCII. |
| LGE (CH1, CH2) | LGE returns . TRUE. if CH1 is lexically greater than or equal to CH2 in ASCII. |
| LGT (CH1, CH2) | LGT returns . TRUE. if CH1 is lexically greater than CH2 in ASCII. |

**Examples:**

1. LLT ('AB', 'AW') returns a value of .TRUE..
2. LLT ('AB', 'A2') returns a value of .FALSE..
3. LLE ('AB', 'AB') returns a value of .TRUE..
4. LLE ('AB', 'A2') returns a value of .FALSE..
5. LGE ('AB', 'A2') returns a value of .TRUE..
6. LGE ('AB', 'AC') returns a value of .FALSE..
7. LGT ('AB', 'A2') returns a value of .TRUE..
8. LGT ('AB', 'AE') returns a value of .FALSE..

**Table B.1** Visible ASCII Characters

| Character | Code | Character | Code | Character | Code |
|---|---|---|---|---|---|
| blank | 0100000 | @ | 1000000 | ` | 1100000 |
| ! | 0100001 | A | 1000001 | a | 1100001 |
| " | 0100010 | B | 1000010 | b | 1100010 |
| # | 0100011 | C | 1000011 | c | 1100011 |
| $ | 0100100 | D | 1000100 | d | 1100100 |
| % | 0100101 | E | 1000101 | e | 1100101 |
| & | 0100110 | F | 1000110 | f | 1100110 |
| ' | 0100111 | G | 1000111 | g | 1100111 |
| ( | 0101000 | H | 1001000 | h | 1101000 |
| ) | 0101001 | I | 1001001 | i | 1101001 |
| * | 1010101 | J | 1001010 | j | 1101010 |
| + | 0101011 | K | 1001011 | k | 1101011 |
| , | 0101100 | L | 1001100 | l | 1101100 |
| - | 0101101 | M | 1001101 | m | 1101101 |
| . | 0101110 | N | 1001110 | n | 1101110 |
| / | 0101111 | O | 1001111 | o | 1101111 |
| 0 | 0110000 | P | 1010000 | p | 1110000 |
| 1 | 0110001 | Q | 1010001 | q | 1110001 |
| 2 | 0110010 | R | 1010010 | r | 1110010 |
| 3 | 0110011 | S | 1010011 | s | 1110011 |
| 4 | 0110100 | T | 1010100 | t | 1110100 |
| 5 | 0110101 | U | 1010101 | u | 1110101 |
| 6 | 0110110 | V | 1010110 | v | 1110110 |
| 7 | 0110111 | W | 1010111 | w | 1110111 |
| 8 | 0111000 | X | 1011000 | x | 1111000 |
| 9 | 0111001 | Y | 1011001 | y | 1111001 |
| : | 0111010 | Z | 1011010 | z | 1111010 |
| ; | 0111011 | [ | 1011011 | { | 1111011 |
| < | 0111100 | \ | 1011100 | ¦ | 1111100 |
| = | 0111101 | ] | 1011101 | } | 1111101 |
| > | 0111110 | ^ | 1011110 | ~ | 1111110 |
| ? | 0111111 | _ | 1011111 | | |

**Table B.2** FORTRAN 77 Character Set

| Character | EBCDIC Code | ASCII Code | Character | EBCDIC Code | ASCII Code |
|---|---|---|---|---|---|
| blank | 01000000 | 0100000 | 0 | 11110000 | 0110000 |
| A | 11000001 | 1000001 | 1 | 11110001 | 0110001 |
| B | 11000010 | 1000010 | 2 | 11110010 | 0110010 |
| C | 11000011 | 1000011 | 3 | 11110011 | 0110011 |
| D | 11000100 | 1000100 | 4 | 11110100 | 0110100 |
| E | 11000101 | 1000101 | 5 | 11110101 | 0110101 |
| F | 11000110 | 1000110 | 6 | 11110110 | 0110110 |
| G | 11000111 | 1000111 | 7 | 11110111 | 0110111 |
| H | 11001000 | 1001000 | 8 | 11111000 | 0111000 |
| I | 11001001 | 1001001 | 9 | 11111001 | 0111001 |
| J | 11010001 | 1001010 | + | 01001110 | 0101011 |
| K | 11010010 | 1001011 | - | 01100000 | 0101101 |
| L | 11010011 | 1001100 | * | 01011100 | 0101010 |
| M | 11010100 | 1001101 | / | 01100001 | 0101111 |
| N | 11010101 | 1001110 | = | 01101110 | 0111101 |
| O | 11010110 | 1001111 | ( | 01001101 | 0101000 |
| P | 11010111 | 1010000 | ) | 01011101 | 0101001 |
| Q | 11011000 | 1010001 | , | 01101011 | 0101100 |
| R | 11011001 | 1010010 | . | 01001011 | 0101110 |
| S | 11100010 | 1010011 | $ | 01011011 | 0100100 |
| T | 11100011 | 1010100 | ' | 01111101 | 0100111 |
| U | 11100100 | 1010101 | : | 01111010 | 0110101 |
| V | 11100101 | 1010110 | | | |
| W | 11100110 | 1010111 | | | |
| X | 11100111 | 1011000 | | | |
| Y | 11101000 | 1011001 | | | |
| Z | 11101001 | 1011010 | | | |

# Appendix C *Additional FORTRAN Features*

Among its other aspects, the FORTRAN 77 standard language is required to recognize and accept existing features from all earlier versions. This makes it possible for successful programs written in these versions to be submitted to (and processed by) a standard FORTRAN 77 compiler without change. Consequently, these features are of historic interest and are outlined here. In addition, this appendix presents other FORTRAN 77 features whose use is sufficiently specialized to warrant their exclusion from the body of the text.

## C.1 DECLARATIONS: THE IMPLICIT STATEMENT

FORTRAN's defaults can be changed temporarily (for a program) unit so that variables (or function names) beginning with specified letters automatically will be assigned a specified data type. The general form is

IMPLICIT *datatype* (*letter*, *letter*, etc.)

The *datatype* specification may be either DOUBLE PRECISION, COMPLEX, REAL, INTEGER, LOGICAL, or CHARACTER***length*. If *length* is omitted, FORTRAN uses

The following additional rules:

(1) The IMPLICIT statement, if used, must appear before other declarations.
(2) Multiple sets of specifications may appear in a single IMPLICIT statement.
(3) The same letter may not appear in more than one specification.
(4) A range of letters may be specified using the form

*startingletter* – *endingletter*

The two limits thus specified must be in alphabetical order. Thus, the declaration

```
IMPLICIT CHARACTER*8 (C, D, E, F, G, H) , LOGICAL (M, N, O)
```

is equivalent to the declaration

```
IMPLICIT CHARACTER*8 (C – H) , LOGICAL (M – O)
```

Either version specifies that all names (in that program unit) beginning with the letters C through H automatically are associated with eight-character strings, and all names beginning with M through O are associated with logical variables.

We have followed the practice throughout the text of using the IMPLICIT NONE declaration to make sure that no defaults are in force.

## C.2 CONTROL STATEMENTS

### C.2.1 The Assigned GO TO Statement

In addition to the unconditional and computed GO TO statements, FORTRAN recognizes the *assigned* GO TO statement. This is another way of specifying a choice of destinations

based on some prior definitions. There are two forms:

GO TO *destination*

or

GO TO *destination*, (*label*, *label*, etc.)

In both forms, *destination* is the name of an integer variable. Though it looks like an integer variable and is declared like an integer variable, *destination* is special:

1. The only value that such a variable may have is that of a statement number used somewhere in the same program unit.
2. The only way such a variable may receive its value is by means of the ASSIGN statement:

   ASSIGN *label* TO *destination*
3. Aside from a declaration statement, this variable can appear only in an ASSIGN statement or in an assigned GO TO statement.

In the second form, the parenthesized list consists of statement numbers used somewhere in the same program unit. There is no limit to the number of labels included in that list; it is not even necessary for all the labels to be different. However, if such a list is used, the value of label ASSIGNed to *destination* must be one of the values given in the list.

### C.2.2 The Arithmetic IF Statement

This is the original form of FORTRAN's IF statement:

IF (*arithmetic expression*) *label1*, *label2*, *label3*

FORTRAN evaluates the arithmetic expression inside the parentheses. If the value is less than zero, the next statement executed is the one with label *label1*. If the value is zero, the next statement executed is the one with label *label2*. If the value is greater than zero, the next statement executed is the one with label *label3*. The three alternative destinations do not all have to be different. Thus,

```
IF (X - Y) 12, 12, 14
```

provides a comparison of X against Y: If X is less than or equal to Y, the program will go to statement 12; otherwise, it will go to statement 14.

### C.2.3 The PAUSE Statement

The PAUSE statement has the form

PAUSE *message*

where *message* is either an unsigned integer constant up to five digits long, or a character string constant. For example,

```
PAUSE 12345
```

or

```
PAUSE 'THE LOOP IS COMPLETE'
```

These statements are intended to help in testing programs. As such, they may be placed anywhere in a program. (Each PAUSE statement should have a unique *message*.) When the executing program comes to a particular PAUSE statement, it displays the associated message on the standard output unit.

C.3 ALTERNATE RETURN ADDRESSES FOR A SUBROUTINE

Use of this feature should be avoided. However, it is there and it will be described.

The ordinary RETURN statement, i.e.,

```
RETURN
```

causes execution to shift from a subprogram to the invoking program unit just after the point of invocation. It is possible to set up a subroutine (not a function) so that there are several specified return points. When this feature is used, the RETURN statement looks like this:

```
RETURN integer expression
```

The value of *integer expression* indicates which of the choices is to be used for that particular return. For instance, if there are four choices from which to select, *integer expression* must have a value of 1, 2, 3, or 4. The choices themselves are defined as statement labels included in the argument list when the subroutine is called. Each statement label thus specified, of course, must correspond to a label attached to an executable statement somewhere in the invoking program unit. Such arguments are specified by giving the statement number, preceded by an asterisk. For instance, the statement

```
CALL SUBR (*12, *14, *8, XSQR, SUMY, NUMZ)
```

invokes a subroutine named SUBR with six arguments: The first three indicate that the subroutine will be able to return to statement 12, statement 14, or statement 8. (The other three arguments are whatever they are.) Correspondingly, if a RETURN statement inside SUBR says

```
RETURN 2
```

it means that, *for this invocation*, the program will return to statement 14 of the invoking program unit. Note that it is the choice, and not the statement number, that is specified in the RETURN statement. For this particular subroutine, the integer (or integer expression) given with the RETURN statement must have a value of 1, 2, or 3.

In order for this to work, the subroutine must include information in its dummy argument list to show that the choices are available. This is done by showing an asterisk for each choice. In this example, since the subroutine is designed to offer three choices, the initial statement would say

```
SUBROUTINE SUBR (*, *, *, DVAL, EVAL, NUMBER)
```

The first three items in the dummy list, then, correspond to the three statement labels given as the first three arguments in the CALL statement.

Subroutines equipped with capabilities for alternate return addresses still may include ordinary RETURN statements.

# *Index*


ABS Built-in Function, 107, 120
ACOS Built-in Function, 118, 121
ACOSH Built-in Function, 118, 121
AIMAG Built-in Function, 512
AINT Built-in Function, 111, 120
Algorithms
  definition of, 16
  description of 21–23
  properties of, 17–19
An American Tradition, 101
ANINT Built-in Function, 112, 120
Arguments for Subprograms, 310–321, 412–414
Arithmetic Expressions
  built-in functions in, 105
  construction of, 79
  operations in, 80
  parentheses in, 82
  priority of operations in, 90
Arrays
  as arguments for subprograms, 308, 310, 317–321
  bounds of, 141
  declaration of, 137, 141
  dimensionality of, 135
  elements of, as arguments for subprograms, 306
  identification of elements in, 140, 143
  in common storage, 483
  initialization of, 142–143
  input/output of, 138, 145, 151
  loops for processing, 258–259
  of character data, 164, 316
  one-dimensional, 135
  organization of, 135, 140
  size limits of, 137
  sorting of values in, 259–263
  storage of, 138, 140
  two-dimensional, 136, 258–259
ASCII Character Set, 515
ASIN Built-in Function, 118, 121
ASINH Built-in Function, 118, 121
Assigned GO TO Statement, 517
Assignment Statement, 39, 47, 165
ATAN Built-in Function, 118, 121
ATANH Built-in Function, 118, 121
ATAN2 Built-in Function, 118, 121

BACKSPACE Statement, 349
Bit-handling Operations, 124–129
Blank Common Storage, 483
Blanks
  in FORTRAN statements, 51
  in input data, 376,440
  in output displays, 398
BLOCK DATA Subprogram, 491
BTEST Built-in Function, 129
Built-in Functions
  ABS, 107, 120
  ACOS, 118, 121
  ACOSH, 118, 121
  AIMAG, 512
  AINT, 111, 120
  as arguments for subprograms, 325
  ASIN, 118, 121
  ASINH, 118, 121
  ATAN, 118, 121
  ATANH, 118, 121
  ATAN2, 118, 121
  BTEST, 129
  CHAR, 181
  CMPLX, 512
  CONJG, 512
  COS, 118, 121
  COSH, 118, 121
  DBLE, 107, 120
  DIM, 108, 120
  DPROD, 108, 120  EXP, 117, 121
  IBCLR, 127
  IBSET, 126
  ICHAR, 181
  INDEX, 173
  INT, 107, 120
  ISHFT, 128
  ISHFTC, 128
  LGE, 170, 514
  LGT, 170, 514
  LLE, 170, 514
  LLT, 170, 514
  LOG, 117, 120
  $LOG_{10}$, 117, 120
  MAX, 113, 120
  MIN, 113, 120
  MOD, 108, 120
  NINT, 111, 120
  REAL, 107, 120
  SIGN, 107, 120
  SIN, 118, 121
  SIGN, 118, 121
  SQRT, 117, 120
  TAN, 118, 121
  TANH, 118, 121

CALL Statement, 279, 322
Canine Fund, 355
Carriage Control, Mechanisms for, 398
CASE Construction, 209–216
Central Processor, Operation of, 1
CHAR Built-in Function, 181
Character Data
  arrays of, 164
  as arguments for subprograms, 307
  assignment of values to, 165
  concatenation of, 167
  constant values for, 66
  conversion of, 181
  declaration of variables for, 160–164
  format description of, 393–396, 403–405
  searching of, 174
  substrings of, 160, 394–395, 404
CHARACTER Declaration, 39, 70

Character Set for HP FORTRAN 77 Programs, 51, 53, 515
Clearing Individual Bit Values, 127
CLOSE Statement, 352
CMPLX Built-in Function, 512
Coconuts, 134
Collating Sequence, 168, 515
Comments in FORTRAN Programs, 51
COMMON Declaration, 483–490
Common Storage
  blank, 483
  declaration of, 483–490
  initialization of values in, 490–492
  misuse of, 494–495
  named, 488–490
  organization of, 485–488
Compilers, Characteristics of
  characteristics of, 8–9
  directives to, 52
Computed GO TO Statement, 214
Computers
  input/output components of, 4–5
  main storage in, 3–4
  major components of, 1
  secondary storage for, 6
Concatenation of Character Strings, 167
CONJG Built-in Function, 512
Constants
  as arguments for subprograms, 304
  as output values, 380, 405
  character, 66
  complex, 511
  decimal, 62
  double precision, 65
  floating point, 65
  hexadecimal, 63
  Hollerith, 66
  logical, 66
  names for, 67
  numerical, 62–66
  octal, 62
Conversion
  of character data, 181
  of Numerical Data, 87–90, 106
COS Built-in Function, 118, 121
COSH Built-in Function, 118, 121
Cyclic Operations. *See* Loops

DATA Declaration, 72–73, 142–143, 490–491, 511
DBLE Built-in Function, 107, 120
Decisions
  based on character string values, 168
  based on individual bit values, 125, 129, 504–505
  based on multiple tests, 193–200
  HP FORTRAN 77 statements for, 42, 62–74
  in structured programs, 22
  N-S description of, 23
  nesting of, 204–209
  pseudocode description of, 23
  with complex actions, 201
  with multiple choices, 209–216
DIM Built-in Function, 108, 120
DIMENSION Declaration, 144
Direct Files, 469–476
DO Statement, 43–44, 48–49, 151, 238–243, 256–259
DO-UNTIL Construction, 234
DO-WHILE Construction, 231–234
DO WHILE Statement, 231
DOUBLE PRECISION Declaration, 69
Double Precision Numbers, 65, 69, 392
DPROD Built-in Function, 108, 120
Drivers for Subprogram Development, 276, 287
Dummy Arguments for Subprograms, 280, 310, 317, 321
Dutch Settlers, Real Estate Transactions With, 101

Edit-Directed Data
  format descriptions for, 388, 394, 401–427, 434–439, 507–508, 513
  interpretation of, 337, 387, 389–393
  runtime descriptions for, 446–448
  techniques for formatting, 406–411, 434–439
  variable format descriptions for, 448–456
ELSE IF Statement, 210
END Declaration, 37
ENDFILE Statement, 349
EQUIVALENCE Declaration, 71–72
EXP Built-in Function, 107, 121
Expressions
  as arguments for subprograms, 305
  construction of, 79, 167, 502, 505
  operations in, 80, 167
  parentheses in, 82, 91, 167, 502
  priorities of arithmetic operations in, 90, 502
  with array elements, 145
External Files, 341
EXTERNAL Statement, 322

Files
  and records, 336
  and units, 341
  backspacing, 345
  closing, 344
  creation of, 343
  deletion of, 343
  direct, 469–476
  examination of, 352–355
  external and internal, 341, 448–456
  opening of, 344, 461, 476
  rewinding of, 345
  unformatted, 460–469
Floating Point Numbers, Storage of, 65
  *See Also* Real Numbers
Format Descriptions
  at runtime, 442–448
  for character data, 393–396, 403–405
  for integer data, 391, 401–402
  for numbers, 391–393, 401–405, 424–437
  for real data, 388–392, 401–403
  techniques for, 406–411, 438–439
  variable, 448–456
Formatted Data, 335
Functions
  built-in, 105
  definitions for, 280
  dummy arguments for, 280
  generic, 120
  invocation of, 119, 284
  names for, 281
  structure of, 277

GRAN Built-in Function, 119
Generic Functions, 120
Ghosts, 131
GO TO Statement, 232–233, 517
Gurus, 117

Hexadecimal Numbers, 63, 392
High Level Languages
  characteristics of, 6–10
  compilers for, 8–9
Hollerith Constants, 66
HP Processors
  characterization of, 3–4
  internal representation of data in, 65, 121

IBCLR Built-in Function, 127
IBSET Built-in Function, 126
ICHAR Built-in Function, 181

IF Block, 42, 201, 233
IF Statement
  basic forms for, 42, 517
  use of, for loop construction, 233
  use of, with character data, 168
  use of, with individual bit values, 125, 504–505
IF-THEN-ELSE Construction, 22
Imaginary Numbers. See Complex Numbers
IMPLICIT Declaration, 71–516
INDEX Built-in Function, 173
Indians, Real Estate Transactions With, 101
Initialization of Variables, 72–73, 142–143
Input
  devices for submitting, 4–6
  edit-directed, 337, 387, 394
  from direct files, 472
  from internal files, 450–456
  list-directed, 337, 374–377
  mechanisms for reading, 40
  of complex data, 513
  of logical data, 507–508
  of unformatted data, 461–469
Input/Output Devices, 4–6
INQUIRE Statement, 352
INT Built-in Function, 106, 120
INTEGER Declaration, 37, 68–69
Integers
  conversion to, 106
  declaration of, 37, 68–69
  expressions with, 86
  format descriptions of, 391, 401–402
  forms for, 62
  internal representation of, 121
  logical operations on, 122–129
Interactive Programs, Examples of, 8, 10
Internal Files, 341, 448–456
INTRINSIC Declaration, 325
Invocation
  nested, 325
  of functions, 119, 284
  of subroutines, 278, 288, 305
IRANP Built-in Function, 119
ISHFT Built-in Function, 128
ISHFTC Built-in Function, 128

Labels for FORTRAN Statements, 50
LGE Built-in Function, 170, 514
LGT Built-in Function, 170, 514
List-Directed Data
  forms for, 374, 377, 507
  interpretation of, 339, 375
Literal Output, 380, 405
LLE Built-in Function, 170, 514
LLT Built-in Function, 170, 504
LOG Built-in Function, 117, 120
$LOG_{10}$, Built-in Function, 117, 120
Logical Constants, 66, 499
LOGICAL Declaration, 70, 499
Logical Operations
  on integer values, 122–129, 505–507
  for manipulating individual bits, 124–126
  properties of, 499–503
Loops
  as structural components in programs, 23
  automatic counters in, 238–244
  control mechanisms for, 43, 231–234, 244–245, 251–255
  DO-UNTIL constructions for, 234
  DO-WHILE constructions for, 43, 231–232
  for counting events, 238–243
  for input/output of arrays, 151, 378
  indexes for, 238
  N-S description of, 23–27
  nested, 256–259
  programming techniques with 243–244
  pseudocode description of, 23
  structure of 42–47, 231–234, 243, 249
  with the block IF statement, 233

Machine Language, 3, 7
Main Storage, Organization of, 3
Masking, Techniques for, 124–129
MAX Built-in Function, 113, 120
Memory. *See* Main Storage
MIN Built-in Function, 113, 120
MOD Built-in Function, 108, 120
MVBITS Subroutine, 127

N-S Diagrams for Describing Algorithms, 21–23, 30
Named Common Storage, 488–490
Names
  for FORTRAN programs, 37
  for variables, 67–68, 71
NINT Built-in Function, 111, 120
Numbers
  complex, 511–513
  conversion among different types of, 87–90, 106–107
  conversion to characters, 181
  decimal, 62
  double precision, 69
  expressions involving, 79
  floating point, 65
  format description of, 388–394, 401–403, 434–437
  forms for 62–65
  hexadecimal, 63, 392
  in logical expressions, 505–507
  octal, 62, 391
  operations on, 80
  precision of, 65
  rounding of, 111–112
  scaling of, 438–439
  scientific notation for, 63

O'Casey's Ghost, 131
Object Programs
  characteristics of, 9
  from source programs, 12
Octal Numbers, 62, 391
OPEN Statement, 350, 461, 473
Operating Systems as Program Environments, 10–12
Operations in Arithmetic Expressions, 80–91
Output
  devices for producing, 4–6
  edit-directed, 389–393, 401–403
  list-directed, 377–378
  mechanisms for writing, 40
  of complex data, 513
  of logical data, 507–508
  of unformatted data, 461–469
  to direct files, 473
  to internal files, 450–452
Overprinting, 399

PARAMETER Declaration, 67
Parameters. *See* Dummy Arguments
Parentheses, Uses of, 82, 91
PAUSE Statement, 517
Precision of Real Numbers, 65
PRINT Statement, 40, 341, 344, 348, 381
PROGRAM Declaration, 37
Programming Languages
  and operating systems, 10–12
  characteristics of, 6–10
  compilers for 8–9
Programs
  components in, 51

Programs *(Continued)*
operation of, 40
organization of, 36, 272
systematic preparation of, 15–16, 275
Pseudocode for Describing Algorithms, 21–23, 30

Random Files. *See* Direct Files
Random Numbers, Generation of, 117–119
READ Statement, 40, 151, 342, 344, 346, 378–379, 461, 473
REAL Built-in Function, 107, 120
REAL Declaration, 69
Real Numbers
conversion to, 107
expressions with, 85
format descriptions for, 388–392, 402–403, 434–439
forms for, 63
precision of, 65
rounding of, 111
truncation of, 111
Records
and files, 336
ENDFILE for, 344
for direct files, 471
formatted, 337
input/output of, 344, 396
list-directed data in, 375
unformatted, 337, 460–469
Redeye Airlines, 384
REWIND Statement, 350
Rounding of Numerical Values, 111–112
Runtime Format Descriptions, 442–448

SAVE Statement, 492–493
Scaling of Numerical Values, 438–439
Scientific Notation, 63–64
Searching Techniques with Character Data, 174
Secondary Storage Devices, 6
Sequence Numbers for FORTRAN Statements, 50
Setting Individual Bit Values, 124, 126
Shifting Operations in HP FORTRAN 77, 128
SIGN Built-in Function, 108, 120
SIN Built-in Function, 118, 121
SINH Built-in Function, 118, 121
Sorting Techniques, 259
Source Programs
characterization of, 7
object programs from, 12
SQRT Built-in Function, 117, 120
SSEED Subroutine, 118
Standard Character Warehouse, 395
Statement Functions, 278
Statements
as high-level language components, 7
blanks in, 51
format for, 47, 50–52
labels for, 50
sequence numbers for, 50
Structured Programming
as part of a larger process, 15–16
characteristics of, 19–20, 30
Structured Programs
basic components of, 20–28
representation of, 21–28
Subprograms
as arguments, 321
common storage for, 481–490
definitions for, 280–284
drivers for developing, 276, 287
for setting up common storage, 490–491
invocation of, 283–289
operating principles of, 272–275, 291–294
saving data from, 492–493
types of, 277
Subroutines
alternate returns from, 518
definitions for, 283
invocation of, 278, 288
structure of, 278
Substrings, 160, 164, 166, 174, 394–395, 404

TAN Built-in Function, 118, 121
TANH Built-in Function, 118, 121
Testing Individual Bit Values, 125, 129
Tradition, American, 101
Tramp Steamers, 134
Truncation of Real Values, 111

Unformatted Files, 460–469
Units and Files, 341, 347
URAN Built-in Function, 118

Variable Format Descriptions, 448–456
Variables
in common storage, 485–490
initialization of, 72–73, 485
local, 492
names for, 67–68, 71–72

WHILE-DO Construction
N-S representation of, 26
pseudocode representation of, 27
WRITE Statement, 342, 344, 347, 380–381, 468, 473

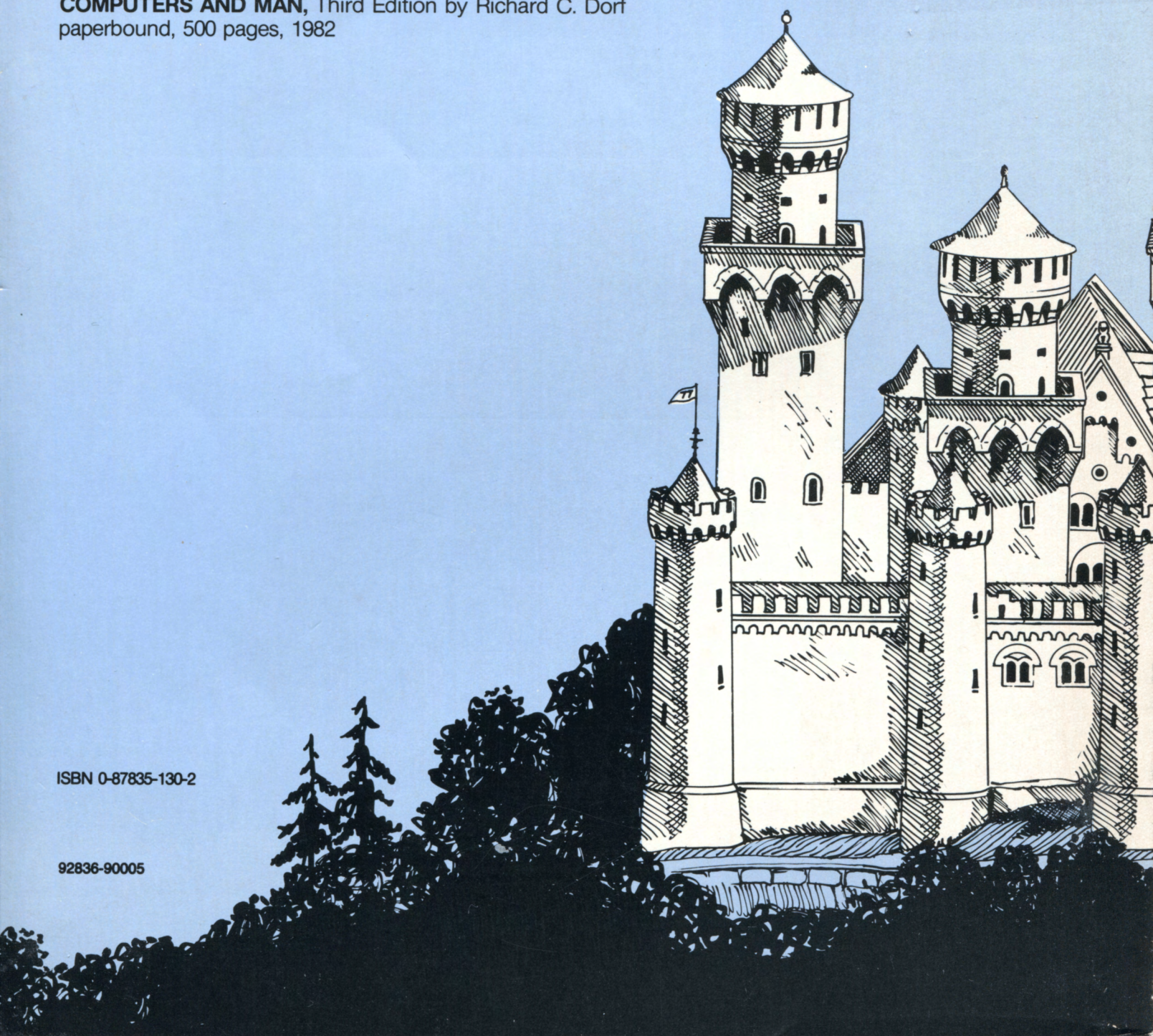
Professor Pollack emphasizes the important interrelation between **FORTRAN 77** and **structured programming.** The features of the **FORTRAN 77** language are carefully illustrated through the use of numerous well-structured examples. In this way the characteristics of the language and principles and techniques of structured programming reinforce one another. The author's **top-down approach** provides the student a proven methodology for effective design and implementation of programs.

The HP Fortran 77 compiler fully implements the American National Standards Institute X3.9-1978 standard (ANSI 77) for FORTRAN. It has many extensions to provide a more structured approach to program development and more flexibility in computing for scientific applications. As part of its extensions, HP FORTRAN fully implements the MIL-STD-1753 Military Standard FORTRAN.

Also available from Boyd & Fraser.

**PASCAL PROGRAMMING: A SPIRAL APPROACH** by Walter S. Brainerd, Charles H. Goldberg and Jonathan L. Gross
paperbound, 584 pages, 1982

**COMPUTERS AND MAN,** Third Edition by Richard C. Dorf
paperbound, 500 pages, 1982



ISBN 0-87835-130-2

92836-90005