

FORTRAN FOR THE '90s

Problem Solving for Scientists and Engineers

Dr. Stacey L. Edgar
State University of New York, Geneseo

COMPUTER SCIENCE PRESS
An imprint of W. H. Freeman and Company
New York

Library of Congress Cataloging-in-Publication Data

Edgar, Stacey L.

**Fortran for the '90s : problem solving for scientists and
engineers / Stacey L. Edgar.**

p. cm.

Includes bibliographical references and index.

ISBN 0-7167-8247-2

I. FORTRAN (Computer program language) I. Title.

QA76.73.F25E35 1992

005. 13'3—dc20

91-44475

CIP

Copyright © 1992 by W. H. Freeman and Company

**No part of this book may be reproduced by any mechanical,
photographic, or electronic process, or in the form of a
phonographic recording, nor may it be stored in a retrieval
system, transmitted, or otherwise copied for public or private
use, without written permission from the publisher.**

Printed in the United States of America

Computer Science Press

An imprint of W. H. Freeman and Company

41 Madison Avenue, New York, NY 10010

20 Beaumont Street, Oxford OX1 2NQ, England

1 2 3 4 5 6 7 8 9 0 VB 9 9 8 7 6 5 4 3 2

CONTENTS

PREFACE xxvii

INTRODUCTION: COMPUTERS AND PROGRAMMING 1

A BRIEF HISTORY OF COMPUTERS 2

HARDWARE AND SOFTWARE 4

COMPUTER STRUCTURE 5

BINARY REPRESENTATION AND STORAGE 6

COMPUTER LANGUAGES 9

 Machine Language 9

 Assembly Language 9

 High-Level Languages 11

 Beyond High-Level Languages 14

BRIEF HISTORY OF FORTRAN 14

SUMMARY OF IMPORTANT CONCEPTS INTRODUCED

 IN THIS CHAPTER 17

EXERCISES 17

SUGGESTED READINGS 19

1 APPROACHES TO PROBLEM SOLVING 21

 DEFINE THE PROBLEM 22

 CLARIFY THE INTENDED RESULT 23

 ORGANIZE THE APPROACH SYSTEMATICALLY

 (TOP-DOWN DESIGN) 24

 ATTENDING TO THE DETAILS (STEPWISE REFINEMENT) 25

 Algorithms 25

 Flowcharts 28

 Other Program Design Tools 32

Structure Charts 32

Nassi-Schneiderman Diagrams 33

Pseudocode 34

 LOOK FOR ANALOGIES 35

 PROGRAM TESTING 35

 APPLYING PROBLEM-SOLVING TECHNIQUES 36

 Problem Specification 36

 Clarifying Problem Specification 36

Developing the Algorithm	37
Testing	38
Refinement and Maintenance	39
The Problem Solved in a FORTRAN Program	39
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	41
EXERCISES	42
SUGGESTED READINGS	43

2 FORTRAN BUILDING BLOCKS 45

CONSTANTS AND VARIABLES 46

Integers	47
Reals (Floating-Point Variables)	48
Other Types of Constants	50
Variables	51
<i>Rules for Naming Variables</i>	51
Variable Types	52

ASSIGNMENT STATEMENTS 54

ARITHMETIC EXPRESSIONS, ORDER OF OPERATIONS 55

Integer Division/Truncation	60
Taking Remainders (A Useful Tool)	61
Mixed-Mode Arithmetic	62
Sequence of Exponentiations—(An Exception to a Rule)	64

SIMPLE (LIST-DIRECTED) INPUT/OUTPUT 66

CREATING FORTRAN ARITHMETIC EXPRESSIONS 70

OTHER SYNTACTIC RULES FOR WELL-FORMED INSTRUCTIONS 71

THE PHYSICAL LAYOUT OF A FORTRAN STATEMENT 72

Comments	74
----------	----

PROGRAM STATEMENT 74

STOP AND END STATEMENTS 75

DEFINED SYMBOLIC CONSTANTS (THE PARAMETER STATEMENT) 75

SIMPLE FORTRAN PROGRAM EXAMPLES 76

Gravitational Force (Two-Body Problem)	77
--	----

Installment Loan Payments	79
---------------------------	----

FORTRAN 90 FEATURES 81

SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER 82

EXERCISES 83

3 CONTROL STATEMENTS 91

MAKING DECISIONS 92

Logical IF Statement	92
----------------------	----

Transfer of Control—The Unconditional Branch	95
Block IF	100
IF/THEN/ELSE Structures	101
ELSEIF Structures	102
Arithmetic IF	105
Logical Connectives (AND, OR, NOT, EQV, and NEQV)	107
<i>DeMorgan's Laws</i>	110
<i>Order of Operations in Logical Expressions</i>	111
CONSTRUCTING SIMPLE REPETITIONS WITH THESE ELEMENTS	112
Loops That Repeat N Times	112
Loops That Repeat Until Some Condition Is Met (Posttest Loops)	114
Loops That Continue WHILE Some Condition Holds (Pretest Loops)	115
COUNTING	117
ACCUMULATION	119
EXCHANGING VALUES	120
SOME SIMPLE (BUT NOT TRIVIAL) PROGRAMS	121
Greatest Common Divisor	123
Revolving Charge Account	126
The Barbarian and the Roman Soldier	129
FORTRAN 90 FEATURES	130
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	130
EXERCISES	132

4 REPETITION STRUCTURES 139

THE DO LOOP	140
Range of a DO Loop	141
Prechecked Loops	144
The CONTINUE Statement	146
Restrictions on the Terminal Statement of a DO Loop	147
DO and END DO (Non-Standard; Included in Fortran 90)	147
Exits from DO Loops	148
NON-STANDARD REPETITION STRUCTURES	149
FORTRAN 90 DO STRUCTURES: VARIATIONS ON A THEME	152
NESTED LOOPS	154
PROPERLY NESTED CONTROL AND REPETITION STRUCTURES	155
EXAMPLE PROGRAMS	160
Prime Numbers	160
Bouncing Tennis Ball	161
Fibonacci Sequence	162

Optimum Wine Cask Dimensions	163
PREVIEW TO SUBROUTINES	165
FORTRAN 90 FEATURES	170
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	171
EXERCISES	172
5 DATA TYPES IN GENERAL	181
TYPE STATEMENTS	182
INTEGERS AND REALS	182
LOGICALS	184
CHARACTERS	186
Substrings	188
Concatenation	189
List-Directed I/O of CHARACTER data	189
Useful Character Functions: LEN, INDEX, CHAR, and ICHAR	190
COMPLEX VARIABLES	192
DOUBLE-PRECISION VARIABLES	196
IMPLICIT TYPE STATEMENT	198
DATA STATEMENT INITIALIZATION OF VARIABLES	199
USER-DEFINED DATA TYPES	201
FORTRAN 90 FEATURES	201
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	202
EXERCISES	203
6 FORMATTED INPUT/OUTPUT	211
WHY BOTHER?	212
PRINT OR WRITE (OUTPUT)	212
Carriage Control, Spacing (X)	212
Integers (I Format)	215
Reals	217
Characters (A Format)	220
Logical, Complex, and Double-Precision Values	220
Table of Various Outputs	221
Slash (/) in Format —Skipped Records	222
Example Program Simulating Gasoline Pump	222
Repeated Format Patterns	223
Recap	225
INPUT FORMATS	226
END= and ERR= Clauses	226
Input READ Descriptors	227
Slash (/) in Input Record	230
Repeated Formats on Input	231

Sample Program	231
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	233
EXERCISES	235
7 ARRAYS—SUBSCRIPTED VARIABLES	241
WHY ARRAYS?	242
ONE-DIMENSIONAL ARRAYS (LISTS, VECTORS)	242
Rules for Subscripts	244
Loops with Arrays	245
Lists and Vectors	248
I/O of One-Dimensional Arrays (and the Implied List)	250
The Visual Display of Data Points Program Revisited	252
Reference Arrays	253
Prime Numbers and Factorization	256
Fortran 90 Additions to Array Handling	259
USER-DEFINED TYPES	260
TWO-DIMENSIONAL ARRAYS	262
Uses	262
Nested Loops with Two-Dimensional Arrays	264
Input/Output of Two-Dimensional Arrays	265
Matrix Manipulations	267
ARRAYS OF HIGHER DIMENSION (UP TO 7)	268
ARRAY MANIPULATIONS	268
Finding Maxima and Minima	268
Searching (Linear)	269
Arrays Used for Counters	271
A Look at Fortran 90 Features for Two-Dimensional Arrays	272
SORTING	273
Bubble Sort	274
Shuttle/Interchange, or Selection, Sort	276
Efficiency Considerations	278
BINARY SEARCH	278
DATA STATEMENT INITIALIZATION OF ARRAYS	280
VECTOR MANIPULATION (AN OPTIONAL SECTION)	281
FORTRAN 90 FEATURES	286
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	286
EXERCISES	288
SUGGESTED READINGS	296
8 ERRORS—A FACT OF LIFE	299
SYNTAX ERRORS	300
EXECUTION-TIME ERRORS	303

LOGIC ERRORS	306
DEFENSIVE PROGRAMMING	309
ROUNDOFF ERRORS	310
Propagation of Errors	311
Cancellation Errors	312
SYMBOLIC DEBUGGERS	312
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	313
EXERCISES	313
SUGGESTED READINGS	320
9 SUBPROGRAMS	323
STATEMENT FUNCTIONS	324
Statement Function for a Hypotenuse	325
Restrictions on Statement Functions	326
Statement Functions for Temperature Conversions	326
"Helper" Statement Functions	327
USE OF SYSTEM FUNCTIONS	329
SUBROUTINES	329
Simple Subroutine Without Arguments	330
Adding Arguments to a Subroutine	331
Subroutines Which Provide Generality	333
Variable Dimensioning	335
COMMON (GLOBAL VARIABLES)	337
Blank COMMON	337
Labelled COMMON	338
BLOCK DATA Subprograms	339
Encoding Problem Using COMMON	340
Advantages and Disadvantages of COMMON	342
FUNCTIONS	344
Availability of Functions to the Rest of the Program	345
Comparison of Subroutines and Functions	345
Function to Test for Identical Matrices	346
SUBPROGRAMS AS ARGUMENTS (EXTERNAL, INTRINSIC)	347
ITERATIVE FUNCTIONS	349
USING SUBPROGRAMS EFFECTIVELY	351
Using SAVE	354
FORTRAN 90 FEATURES	354
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	355
EXERCISES	356

10 ADDITIONAL CONTROL STATEMENTS AND DATA MANIPULATION	365
THE ELSEIF STRUCTURE FOR CASE	366
THE COMPUTED GO TO	368
ASSIGNED GO TO	372
EQUIVALENCE	372
Using EQUIVALENCE to Equate Different Types of Values	374
Using EQUIVALENCE to Equate Arrays of Different Dimensions	375
Using EQUIVALENCE with COMMON	377
CHARACTER DATA AND TEXT HANDLING	379
PACKING SMALL VALUES	381
Packing True/False Values for Logical Manipulation	382
LARGE INTEGERS	383
A SUBTLE WAY TO INTERCHANGE VALUES USING EQUIVALENCE	387
FORTRAN 90 FEATURES	389
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	389
EXERCISES	390
11 FILE HANDLING AND OTHER ADVANCED INPUT/OUTPUT	399
OTHER FORMAT DESCRIPTORS	400
Integer Output Revisited	400
Numeric Signs	400
Scaling	401
BN and BZ Editing, Discussed Further	402
TABBING (T FORMAT)	403
Relative Tabbing	403
DIRECT AND SEQUENTIAL ACCESS FILES	404
OPEN Clause	405
ENDFILE Statement	407
BACKSPACE and REWIND	407
CLOSE	408
INQUIRE	409
A Simple Sequential-Access Program	410
A Simple Direct-Access Program	411
UNFORMATTED INPUT/OUTPUT	411
UPDATING A FILE	412

INTERNAL FILES (CORE-TO-CORE DATA TRANSFER)	414
WRITING TO PRINTER AND SCREEN IN THE SAME PROGRAM	416
VARIABLE FORMATTING	417
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	420
EXERCISES	421

12 VISUAL OUTPUT 433

THE PRINTER AS PLOTTER	434
Histograms	434
Plotting with Scaling	435
Plots with Incommensurate Axes	436
Biorhythms	439
Cartesian Graphs	442
SCIENTIFIC VISUALIZATION	448
FRACTALS	449
SCREEN GRAPHICS	451
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER	452
EXERCISES	452
SUGGESTED READINGS	460

13 SOME INTERESTING APPLICATIONS 463

SIMULATIONS AND MODELS	464
Why Model?	464
Random Number Generators	466
Scaling the Generator to Actual Distributions	468
Monte Carlo Techniques	473
Other Distributions	476
Generating Random Test Data	478
INTERACTIVE PROGRAMS	478
PATTERN RECOGNITION	479
IMPLEMENTING OTHER DATA STRUCTURES	483
Abstract Data Structures	483
User-Defined Types	484
Fortran 90 Derived Data Types	487
Records	488
Sets	489
Stacks	490
Queues	491
Linked Lists	493
FORTRAN 90 FEATURES	495

SUMMARY OF IMPORTANT CONCEPTS INTRODUCED
IN THIS CHAPTER 496
EXERCISES 497
SUGGESTED READINGS 508

14 NUMERIC METHODS 511

ROOT FINDING 512
DATA IMPROVEMENT 515
 Smoothing Data 515
 Enhancing Data 515
LEAST-SQUARES FITTING OF DATA 516
NUMERIC INTEGRATION 518
 Rectangular Quadrature 519
 Trapezoidal Rule 520
DIFFERENTIATION 522
SOLVING SYSTEMS OF SIMULTANEOUS LINEAR EQUATIONS 523
 Gaussian Elimination 524
 Determinants 527
FORTRAN 90 FEATURES 528
SUMMARY OF IMPORTANT CONCEPTS INTRODUCED
IN THIS CHAPTER 529
EXERCISES 529
SUGGESTED READINGS 536

15 SOME CLOSING REMARKS 539

STANDARDS FOR FORTRAN PROGRAMS 540
STRUCTURED PROGRAM DESIGN 541
WRITING WITH STYLE 544
 Input/Output Style Hints 548
DOCUMENTATION 549
 Internal Documentation 549
 External Documentation 550
SOFTWARE ENGINEERING 551
 Rebounding from User Data Entry Errors 553
 Program Testing and Reliability 554
 Structured Walkthroughs 555
 The Software Life Cycle 556
SUPERCOMPUTING AND FORTRAN 556
OPTIMIZATION 561
THE GRAND CHALLENGES IN SCIENCE AND ENGINEERING 565
SUGGESTED READINGS 566
SOURCES OF MATHEMATICAL SUBROUTINE PACKAGES 567

APPENDIX A	NUMBER SYSTEMS	569
APPENDIX B	FORTRAN 77 SYSTEM FUNCTIONS	577
APPENDIX C	CAPSULE SUMMARY OF FORTRAN 77 STATEMENTS	583
APPENDIX D	ASCII AND EBCDIC CODING SYSTEMS	593
APPENDIX E	FEATURES OF FORTRAN 90	598
ANSWERS TO SELECTED EXERCISES		611
CREDITS FOR CHAPTER OPENING PHOTOGRAPHS		652
INDEX		653

LIST OF PROBLEMS AND EXAMPLES

Problems (that is, exercises at the ends of chapters) are designated by the chapter number, followed by a dash, followed by the problem number (for example, 2-18). Examples (that is, illustrations in the text) are designated by the chapter number in parentheses followed by the page numbers (e.g., (3) pp. 91-92).

PHYSICS

Recoil velocity of a cannon/clown system	(2) pp.70-71
Gravitational force (two-body problem)	(2) pp. 77-78, 2-15&16; 4-23
Distance a body falls in "free fall" under gravity	(2) p. 58
Velocity of body sliding down a frictionless plane	(2) pp. 58-59
Height at which water from a hose hits a building (Roxanne)	2-7
Orbital angular momentum of a proton in an accelerator	2-18
Distance a proton travels in an accelerator	2-20
Eratosthenes' estimate of the radius of the earth	2-21
Maximum height for a projectile	2-24
Static friction, inclined plane	2-25
Path of an airplane in a crosswind	2-26
Motorcycle cop chasing a speeder	2-27
Temperature in a danger range	(3) p. 98
Testing measurements against an average	(3) pp. 98-99; 106-7
Barbarian shooting an arrow at a Roman soldier	(3) pp. 129-130
Weight-processing program	3-5
Newton's Second Law (conservation of momentum)	3-24
Maximum range of projectiles	3-25
Inelastic collisions (bullet striking wood on ice)	3-30
Distances of projectiles fired at different angles	4-14
Tunnels through earth between cities	4-21
Calculating the height of a rainbow	4-24
Line spectra (quantum mechanics)	4-26
Histogram of measurements	5-17; (7)p. 248; 7-21
Table of areas of squares and volumes of cubes	(6) pp. 216-217
Calculating the average tensile strength of materials	(6)p. 227
Visual display of data points from experiment	(6) pp. 231-232; (7) 252-253

Distance covered by an accelerating body	6-12
Scientific handbook tables of trigonometric functions	6-16
Ballistics table for "Big Bertha"	6-17
Weights—sum, average, standard deviation	(7) pp. 245-247
Counting ranges of experimental data	(7) pp. 248, 272
Sorting a 2-D array of projectile flight data	7-13
Calculating variation of measure from predicted values	8-21
Calculating time difference in seconds	(9) pp. 327-328
Statement functions to convert feet to meters, meters to feet	9-7
Statement functions to convert yards, feet, and inches to inches	9-8
Statement function to calculate maximum height of a projectile	9-9
Mapping distances between all pairs of points (maximum and minimum)	9-19
Center of gravity of a 2-Dimensional object	10-17
Plotting data using Tab	11-16
Temperature Plot Program	(12) pp. 437-438
Plotting trajectories of a projectile fired at different angles	12-9
Plotting the trajectory of a bomb dropped from a plane	12-10
Plotting a cycloid	12-11
Plotting trajectories of two planes heading toward one another	12-14

CHEMISTRY/CHEMICAL ENGINEERING

Amount of ammonia created	2-10; 3-11
Calories to raise an element's temperature a specific amount	2-11
Law of Dulong and Petit test	2-12; 3-20
Conversion: Fahrenheit to Celsius (and Kelvin); Centigrade to Fahrenheit (and Rankine)	2-14
Searching the Periodic Table	(7) p. 261
Statement functions for temperature conversion	(9) pp. 326-327
Creating sequential files of temperature measurements	(11) pp. 410-411
Radioactive decay plot	12-13
Operation function to combine two chemical compounds	13-41

STATISTICS

Averaging a set of measurements	(4) pp. 150-152
Analyzing scientific data (mean, st. dev., median, range, etc.)	(9) pp. 352-353
Pearson correlation	11-19
Scattergram	11-20; (12) pp. 447-448; 14-10
Statistics for poker	13-9
Probability of same birthday in a group of size N	13-30
Number of males 6' tall or taller	14-21

ENVIRONMENTAL ENGINEERING

Water quality (segmented nonlinear function)	3-26
Air Quality Standards	3-27
Pollution Standards Index and Health Effects	3-28; 7-27

COMPUTER SCIENCE

Binary representation of decimal integers, reals	Intro-7-9, 11-13
"Assembly language" simulated programs	(Intro) pp. 10-11; Intro-15, 16
"Machine epsilon" value determined experimentally	4-27
Simple logic design	5-20
Output printable characters in ASCII or EBCDIC	6-19
User-defined types	(7) pp.260-261
Linear search	(7) pp. 269-271
Bubble sort	(7) pp. 274-276; 7-12
Selection sort	(7) pp. 276-278
Shell sort	7-23
Insertion sort	13-33
Binary search	(7) pp. 278-280
Converting from base 10 to another base	7-7
Overflow and underflow	8-13
Effect of propagation of errors	8-20
"Packing" small values	(10) pp. 381-382
"Unpacking" small values	10-10
Two's complement of binary value	10-25
Adding a "check" digit	10-28
Hash coding record numbers for a direct-access file	11-10, 11
Ordering function for abstract data types	(13) pp. 485-486
"Push" and "pop" functions for stacks	(13) p. 490; 13-29
Queue model for plane arrivals and departures	(13) pp. 491-493
Linked-list version of bubble sort	(13) p. 495
Queue representation of a busy intersection	13-22
FORTRAN simulation of record structures	(15) p. 548
Rebounding from bad data (making a program "robust")	(15) pp. 553-554
Optimization of loop structures	(15) pp. 562-564

BIOLOGY

Structure of a phylogenetic tree	1-6
Linus Pauling's estimate of the effect of smoking cigarettes	2-17
Fibonacci Numbers	(4) pp. 162-163; 7-4; (Recursive)9-26
Exponential growth and decay	4-11
Ratio of surface area to volume for spherical creatures	4-18
Matching blood donors	6-18

MISCELLANEOUS

Area enclosed by a rope, in different shapes	(1) pp. 39-40
Structure of the chain of command in an organization	1-5
Installment loan payments	(2) pp. 78-81; 4-17; (9) p. 327
Conversion to military time	2-13
Honor-roll printout and count	(3) pp. 99-100

Printout of the song "99 Bottles of Beer on the Wall"	(3) p. 113
Revolving charge account	(3) pp. 126–128
Leap-year test	3–1
A more general leap-year test	9–4
Converting letter grades to numeric; GPA	3–2
Counting grades in the range 75–85	3–6
Graduated pay scale	3–17
Graduated income tax	3–18
Rating presidential candidates	(4) pp. 156–158
Interest earned, compounded in various ways	4–10
Doubling money every two years	4–12
Comparing job offers	4–13
Popcorn-popper production	4–15
Interest on the \$24 received by Indians for Manhattan Island	(6) pp. 217–218
Calculating the average GPA	6–6
Processing questionnaires	6–8
Curving grades	(7) p. 247
Number of days old a person is	7–18
Printing out a calendar	7–19, 20; 10–2
Weighted grades problem	7–25
Writing accurate paycheck amounts	8–17
Finding the date for Easter Sunday, given the year	9–5
Test for palindromes	9–11
Form letter	10–6
Monetary exchange rates	10–16
Number of days between two dates in the same year	10–20
Updating a file	(11) pp. 412–414; 11–9
Histogram of grade distribution	(12) pp. 434–435
Biorhythm plot	(12) pp. 439–442; 12–2
Drawing a clock face given the time	12–15, 16
How much is a billion dollars?	13–5
Writing paychecks	13–38
Spell-checker	13–39
Readability index (for DoD documents)	13–40

PSYCHOLOGY

Highest IQ in a set of data	(3) pp. 121–122
-----------------------------	-----------------

OENOLOGY

Optimum Wine Cask Dimensions	(4) pp. 163–165
------------------------------	-----------------

MATHEMATICS

Sums of integers	1–1c, 1d; (3) p. 119–120; (4) p. 142; 4–2, 3
Interest on investment	1–1g
Factorials	1–3; 3–4; 4–1; 9–24, (Recursive) 9–25

Largest value of a set of integers	1–4
Number of years equal to a billion seconds	(2) p. 84
Polar coordinates	(2) pp. 87–88
Fermat numbers	(2) p. 66
Table of squares, cubes, square roots, cube roots	2–6
Length of a microcentury	2–19
Smaller of two values	2–23
Count of # of integers whose digits sum to 9	(3) p. 118
Greatest Common Divisor	(3) pp. 123–126; (function) 9–31
Sum of the fractions in series $1/1 + 1/2 + 1/3 + \dots$	3–3; (4) pp. 148–150
Sum of series $1 + 1/2 + 1/4 + 1/8 + \dots$	3–8; 8–16; 12–1
Reversing the digits in an integer	3–7; 9–13
Sum of digits in an n-digit integer	(4) pp. 158–159
Product of integers	3–13
Sum of $1/200 + 1/199 + 1/198 + \dots + 1$	3–15
Sum of integers > 1000	3–16
Numerical centers	3–21, 22
Finding 3-digit integers = sum of their digits	4–7; 7–2
Perfect numbers	4–9; 7–17
Alternating series	4–16
Number of significant digits in an integer	4–22
Zeno's progression	(5)p. 197
Values of pi	5–10; 10–13
Complex values in polar and exponential form	5–8
Real and imaginary components of a product of complex values	5–14
Complex nth roots	5–15
Complex values as coordinates of a triangle; computations	5–16
Reference array for factorials to calculate combinations	(7) pp. 254–256
Multiplication table	(7) p. 264
Matrix multiplication	(7) pp. 267–268
Finding maximum value in a table	(7) p. 269
Finding a number whose sum of factors equals 888	7–8
Prime factors	7–15, 11–8
Goldbach's conjecture	7–16
Combinations, using arrays	7–24
Calculations with fractions	8–18
Statement function to calculate a hypotenuse	(9) pp. 325–326
Testing matrices for identity	(9) pp. 346–347
Absolute value function	9–2
Modulo function	9–3
Statement function to "round" a value	9–6
Testing two numbers for "amicability"	9–15
Recursive function for nth power of X	9–25
Least Common Multiples	9–32
Manipulating large integers (multiplication)	(10) pp. 383–386
Filling a Hilbert matrix	10–5
Adding/subtracting large integers	10–11

Factorials (large integers)	10–14
Printing out a large integer with inserted commas	10–21
Converting integers to Roman numerals, Roman numerals to integers	10–27
Large Fibonacci numbers	11–3
Plotting the program for a circle	(12) pp. 442–446
Density plot for function of two variables	12–3
Plotting an ellipse	12–4
Plotting a parabola and a straight line	12–5
Plotting a sine curve, filling in under the curve	12–8
Outputing Pascal’s triangle	12–23
Storing a lower triangular matrix	13–37

POPULATION DYNAMICS (DEMOGRAPHICS)

Rabbits and foxes	4–19
World population versus land surface	4–20
Population doubling times	13–1
<i>Limits to Growth</i> on food production	14–18
<i>Limits to Growth</i> global reserves depletion	14–19

AUDIO ENGINEERING

Identification of sound-intensity levels	(3) pp. 102–105
--	-----------------

CIVIL ENGINEERING

Fines for exceeding the speed limit	3–23
Measuring inaccessible distances	5–18

OPTICS

Ultraviolet range elements	(3) pp. 114–115
Index of refraction problems	9–37
Snell’s Law	9–38

CRYPTOLOGY

Prime numbers	(4) pp. 160–161; (5) pp. 185–186; (7) pp. 257–259
Prime numbers—the sieve of Eratosthenes	7–14
Various encoded message problems	7–10, 11
A coding problem (“SEND MORE MONEY”)	(9) pp. 340–342
Substitution cipher	10–22
Converting a phone number to “words”	10–23

ELECTRICAL ENGINEERING

Resistors in parallel	2–8; 9–1
Complex roots of a quadratic equation	(5) pp. 193–194; 5–9
Impedance for an electrical circuit	(5) pp. 194–196

Impedance and voltage for any RLC circuit	5–11
Resonance frequency for any RLC circuit	5–12
Impedances in series and parallel	5–13
Complex absolute values	5–19
Relaxation method, calculating electrostatic potential	7–26
Digital logic circuit analysis	11–21
Karnaugh map simplification for digital logic	11–22
Plotting a square wave and a sawtooth wave	12–12

LOGIC

DeMorgan's Laws	(3) p. 110; 5–1
Packing logical values	(10) pp. 382–383
Interchanging values using EQUIVALENCE	(10) pp. 387–389
Outputing truth tables	11–5, 6

SETS

Membership, union, and intersection	5–3, 4
Matching diseases and symptoms	(13) pp. 489–490
Searching for oil	13–12

TEXT MANIPULATION

Printing text in large block letters	2–5
Generating all substrings of a string	5–5
Generating all concatenations	5–6
Anagrams	5–7
Converting text from uppercase to lowercase	6–11
Outputing integers, 10 per line, with one space between	6–15
Creating a header in which numbers output columns	(9) pp. 330–332
Converting an n-character string to an integer	(10) pp. 380–381
Converting a character string to a real	10–8
Converting a text from lowercase to uppercase	10–9
Average word length in text	10–26
Converting an integer value to a character string (core-to-core)	11–12
Extracting the middle of a string	11–13
"Censor" a text file	11–14
Editing a text file to create gender-neutral terminology	11–15

NUMERICAL ANALYSIS

Finding the maximum of a function	(9) p. 348
An iterative function to calculate a square root	(9) pp. 349–350
An iterative function to calculate a cube root	9–22; 9–33
An iterative function to calculate pi	9–23
An iterative function to calculate sine(x)	9–27
An iterative function to calculate cosine(x)	9–28; 9–34
An iterative function to calculate e**x	9–29
Hyperbolic functions sinh and cosh	9–30

Printing out a fraction to a specified number of digits	10–12
Printing out pi to a specified number of digits	10–13
Storing and manipulating sparse matrices	10–24
Monte Carlo approximation of pi	(13) p. 473
Monte Carlo approximation of a function $f(x)$	(13) pp. 474–475
Root finding/bisection method	(14) pp. 512–514; 14–1
Root finding/Newton-Raphson method	(14) p. 514; 14–2
Rectangular quadrature area calculation	(14) pp. 519–520
Trapezoidal rule for area calculation	(14) p. 520
Gaussian elimination for a system of linear equations	(14) 524–527; 14–14
Finding two points where a function changes sign	14–1
Finding areas comparing different techniques	14–11, 16
Finding the area between two curves	14–12
Approximating the derivative	14–13
An iterative function to find the Nth root of a value	14–14
An efficient algorithm to calculate x^{**N}	14–17

GAME-PLAYING

Outputing a chessboard	6–3
Subroutine to set up a chess board	9–16
Does a group of five Queens “cover” a chess board?	9–17
Eight Queens Problem	9–18
Evaluating a tic-tac-toe board	9–35
Twelve Knights problem	9–36
Game of craps, simulation of wins and losses	10–4; 13–4, 11
“Guess the number”	(13) p. 472
Poker	13–8, 9
Hangman game	13–18

ASTRONOMY

Age as a fraction of the age of the universe	2–9
Time for light to travel to Earth from the Sun	2–22
Bode’s Law	4–25
Distance of a moving galaxy from ours at different times	6–9
Finding the day of the week for any date	10–1, 18
Converting the day of the year into the month/day and the astrological sign	10–3
Converting the day of the year into the month/day and the day of week	10–19
Pictorial representation of an eclipse of the Sun	12–12
“Once in a Blue Moon”	13–35

METEOROLOGY

Histogram of data from a weather-monitoring station	12–1
---	------

PATTERN RECOGNITION

Calculating error in matching template to unknown pattern	(13) 480–481
Normalizing a pattern	(13) p. 481
Finding the center of gravity of a pattern	13–19
Pattern “signature”	13–20

SIMULATION

Simulating a gasoline pump	(6) pp. 222–223; 6–13
Slot machine simulation	12–22
Mid-square random number generator	13–2
Linear congruential random number generator	13–3
Porthos and Aramis fight a duel	13–6
Shuffling and dealing cards	13–7
Doves and tigers in the Coliseum	13–10
Simulation of English (information theory)	13–12
Simulation of the expected heights of volunteers	13–13
Generating random test data	13–14, 15, 16, 17
“Game of Life”	13–23
Battleship chasing an enemy ship	13–24
Organism eating the hull of a Klingon ship	13–25
Random walk/ Brownian motion	13–26
Self-avoiding random walk	13–27
Creating a pointillist “painting”	13–28
Generating normally distributed random numbers	13–31
“Poisson” probabilities	13–32

FRACTALS

Mandelbrot Set	(12) pp. 450–451; 12–19
Julia Sets	12–20, 21
Recursive triangle pattern	13–4

IMAGING TECHNOLOGY

Rotating a visual object through angle A	12–6
Rotating a rectangle, superimposing	2–7
“Windowing” a graphical image	12–17
Smoothing data (remote sensing application)	14–3
Enhancing data	14–4
Reducing a matrix of sensed data	14–5
Removing noise from sensed data	14–6
“Stretching” brightness values in a visual array	14–7
Convolution (filtering data using a weighted mask)	14–8
Enhancing contrast in sensed data	14–9

GEOGRAPHY

Directed graphs/distances on a map 13–36

ARCHITECTURE

“Golden Mean” ratio (aesthetically pleasing shape) 4–29

MUSICOLOGY

Piano keyboard/frequencies of sounds from keys 4–30

MANUFACTURING ENGINEERING

Equipment inventory statistics 7–1, 3

Linear Programming 14–23

SOCIAL ENGINEERING

Utilitarian decision-making 7–25

ARCHAEOLOGY

Radioactive carbon 14 dating of Standard of Ur 14–20

ECOLOGY

Allometric relationships/heart rate and body mass 14–22

SPORTS SCIENCE

Baseball pitcher’s throw 2–28

Basketball tryout roster (3) pp. 122–123

Conservation of momentum for two skaters 3–29

Bouncing tennis ball (4) pp. 160–161; 4–8

Bobsled (conservation of energy) 4–28

Scoring an Olympics gymnastic event 10–15

PREFACE

The object of this textbook is to provide science and engineering students with a modern introduction to programming in the major scientific language, FORTRAN. After its creation in the 1950s, FORTRAN developed many different dialects as it was continually modified. The current American National Standards Institute (ANSI) FORTRAN standard is ANSI X3.9-1978, widely known as FORTRAN 77. This book presents a thorough discussion of FORTRAN 77 and provides an introduction to the next standard of the language, Fortran 90, currently under development.

This text is oriented toward a first course in scientific programming in FORTRAN, and it can also be used in a FORTRAN-as-a-second-language course. Since it is the practical scientific applications that will keep FORTRAN the major scientific language for years to come, these aspects are featured, preparing students for the reality of solving problems efficiently. There are other, more advanced FORTRAN texts, which deal with the more sophisticated features of the language and large-scale problems; this text is designed to be a good foundation for such advanced work.

Because FORTRAN's strength is its utility in scientific problem solving, this text emphasizes applications. A wide range of fields is covered in a variety of programming problems—from physics to music, chemistry to sports. The main goal of the text is to give students the working knowledge of FORTRAN that they will need to solve problems creatively in their schoolwork and in their careers. The book thoroughly covers all of the basics of the language and includes special emphasis on scientific features such as vectors, matrices, and complex numbers. It then branches out into exciting application areas such as simulations and models, pattern recognition, numeric integration, queues, scientific visualization, and fractals.

This book has undergone extensive class testing at SUNY-Geneseo, making the book more accessible to students and more effective in its scope and approach.



GOALS

This text has five primary goals.

1. To provide clear explanations of concepts through an accessible writing style and the extensive use of examples and exercises
2. To provide a thorough introduction to the wide range of FORTRAN applications by supplying programming problems in a variety of disciplines, giving students a compendium of FORTRAN tools and techniques
3. To emphasize problem solving in a practical software engineering environment
4. To provide a “bridge” between FORTRAN 77 and Fortran 90 by introducing key features of the new standard as they apply to the topics discussed
5. To give comprehensive coverage to important topics not often found in introductory FORTRAN texts such as scientific visualization and fractals

In summary, the text strives to be a clear exposition of all the essential tools of the FORTRAN language, an adventure into their use in many different problems, and a guide to developing programming expertise through example and practice.



SPECIAL FEATURES

To reach its goals, the book provides several special features.

Chapter Openers: The paragraphs on the first page of each chapter serve two purposes: to briefly describe the contents of the chapter and to motivate the students to study the chapter by describing the skills and knowledge to be gained.

Chapter Summaries: Each chapter ends with an informative, concise summary of the important concepts introduced.

Worked Problems: The “Answers to Selected Exercises” section at the back of the book contains fully worked programming answers for approximately one quarter of the problems in the book.

Linkage with Fortran 90 Throughout: Fortran 90 is not just covered in an appendix. Fortran 90 features are introduced in nearly every chapter and directly applied to the topic being covered.

Emphasis on Problem-Solving Techniques: Chapter 1 thoroughly describes the art of problem solving and clearly explains how each step of

the process works. Examples and problems throughout the book further emphasize the importance of using problem-solving techniques logically in a variety of applications.

Early Introduction to Subroutines: In Chapter 4, "Repetition Structures," subroutines are previewed in a manner that beginning students can comprehend. In Chapter 9, subroutines are explored in depth.

Special Chapter on Errors: Because errors can be such a frustrating problem, especially for beginners, Chapter 8 is devoted exclusively to dealing with them.

Coverage of Visual Output: Chapter 12 discusses the creation of visual output with FORTRAN, including coverage of histograms, scientific visualization, fractals, and screen graphics.



SUPPLEMENTS

Instructor's Manual

The Instructor's Manual that accompanies the text contains several valuable aides to the instructor.

Teaching suggestions based on the use of this material in several semesters of class testing

Suggested course structure also based on previous use of this material

Additional examples of a variety of problems on topics throughout the text

Answers to problems not given in the back of the text, to allow the instructor greater flexibility in assigning homework

Exam questions based on the material

Programming Aptitude Test, similar to those given to prospective employees in industry, which has proved an interesting and informative exercise for students

Data Disks

Diskettes containing all the programs in the text and solutions to some exercises are also available from the publisher.

◆ ACKNOWLEDGMENTS

I would like to thank two reviewers who contributed valuable suggestions for the improvement of this book: Charles Redecker of the University of Washington and Nan C. Schaller of the Rochester Institute of Technology. I would also like to thank my editor, Nola Hague, for her efforts in seeing this project through to completion. Thanks to Larry Marcus for his work in finding some excellent photographs. And, lastly, I'd like to extend a special thank you to all the students I've taught over the years. Without them, this book never would have been possible.

Stacey L. Edgar

INTRODUCTION



COMPUTERS AND PROGRAMMING

Scientists, engineers, and many other people use computers to solve problems. In this book you will learn to use a fairly sophisticated computer language, FORTRAN 77, and preview the added capabilities of its next version, Fortran 90. Many important science and engineering projects of the last four decades depended on FORTRAN. This introduction will provide an overview of FORTRAN's place in the computer revolution.

With the exception of the space shuttle's final descent into the earth's atmosphere, computers pilot the spacecraft and monitor all onboard systems.

“‘Where shall I begin, please your Majesty?’ he asked. ‘Begin at the beginning,’ the King said, gravely, ‘and go on till you come to the end; then stop.’”

- Lewis Carroll, Alice in Wonderland



A BRIEF HISTORY OF COMPUTERS

The science of *computing* is as old as that of logic and mathematics, requiring only an appropriate symbolism and a mind to manipulate these symbols according to rules. Yet it is clear that the use of certain tools can make the mental manipulations easier and more efficient. Such tools have included wax tablet and stylus (or papyrus and quill, or, more recently, paper and pencil), but they have also included calculating devices such as the abacus, Leibniz' or Pascal's calculating machines, and today's computers and hand calculators. Such computing aids generally have been developed to free the human mind from the more menial calculating tasks, to allow it the flexibility to work on more interesting and complex aspects of the world.

The development of calculating aids was relatively slow after the abacus, first mentioned in Herodotus about 450 B.C., but probably dating back in China to at least the sixth century B.C. In the seventeenth century, Leibniz and Pascal began building physical devices that performed arithmetic calculations mechanically, an activity that was to culminate in today's high-speed general-purpose computers. The most significant figure in this development was Charles Babbage, an Englishman who, in the early nineteenth century, developed the plans for an Analytical Engine, the precursor of today's digital computers. The Analytical Engine contained a “store” for values and a “mill” for their manipulation. Ada, Lady Lovelace, the only daughter of the poet Lord Byron, was a mathematician who had studied under Augustus De Morgan and took a great interest in Babbage's work. She wrote programs for the machine, thereby becoming the first programmer, and also wrote letters and articles to the London papers supporting Babbage's work. However, there was not enough money to complete his work, so Babbage died bitter and frustrated, his work largely neglected for over a hundred years.

Two government workers in Buffalo, New York, Herman Hollerith and John Billings, came up with the notion of using a punched-card mechanism to store and process information efficiently. The 1880 census was still not completely processed by the time it came to take the 1890 census. Since the population of the country was continually expanding, clearly something had to be done to keep up to date. Hollerith and Billings developed the idea of putting the census data on cardboard cards by creating patterns of punched holes to represent the information and then using a mechanical sorting device

to select on particular patterns of holes. With this innovation, the 1890 census was completed within one month after all the data had been collected. Hollerith left the census office to found his own company, the American Tabulating Machine Company. By 1914, a young man named Thomas J. Watson had joined the firm, and by 1924 he took over, renaming it IBM.

The Second World War created the demand for ballistics tables to aid in positioning artillery pieces and in making bombing raids effective. Such complex tables took a long time to calculate, taking into account many variables such as wind velocity, temperature, angle of the weapon, or speed of the bomber, weight of the shells, and so forth. The calculation of such a trajectory required about 750 multiplications, and it took about 20 hours for a human to calculate a 60-second trajectory. For this reason, scientists designed a calculating machine (computer) named ENIAC (Electronic Numerical Integrator and Calculator) which was "hard-wired" to do just one job—calculate trajectories for projectiles under different conditions. Information on a particular projectile, its target and distance, and other conditions such as weather (wind velocity, air density) were "input" to the machine, which always performed the same set of operations on whatever data it was fed. It then "output" the required table of details for the trajectory.

In 30 seconds, the ENIAC could compute a trajectory that would have taken a human 20 hours to complete. The machine took up 1500 square feet of floor space and weighed 30 tons. It was made up of 19,000 vacuum tubes and 1500 telephone relays, and had a high failure rate. It performed as an "idiot savant," highly skilled at one job and unable to do any others. Work on such machines involved many of the foremost professors at the Princeton Institute for Advanced Studies—scholars such as Oswald Veblen (father of economist Thorstein Veblen), Albert Einstein, Hermann Weyl, Eugene Wigner, and John von Neumann.

Work had been done earlier on a similar vacuum-tube machine at Iowa State University, by John V. Atanasoff, to solve systems of linear equations. Atanasoff thus claims priority in the development of the first digital computer in this century. He won a court suit in 1974 against the patent filed by the developers of ENIAC, since they had been using many of his ideas.

The brilliant mathematician von Neumann who, with Oscar Morgenstern, had developed mathematical game theory (to be used extensively in economics), noted the weakness of having a huge machine like ENIAC which could only perform one task. He thus developed the concept of a "stored-program" computer, in which the set of operations to be performed (the "program") could be input as well as the data. One should probably say it was "rediscovered," since this was the principle Babbage employed in his Analytical Engine, using a *mill* and a *store*. This concept gave rise to the first general-purpose stored-program computer, the EDVAC (Electronic Discrete Variable Automatic Computer), in 1949–1952. There were other parallel developments in computers in this country and in Europe, and the computer age was underway.

 **HARDWARE AND SOFTWARE**

When talking about computers, it is convenient, and fashionable, to make a distinction between their hardware—the physical components such as wires, microchips, and the like—and their software—the programs, compilers, operating systems, and other such logical constructs that control the hardware and make it perform useful and interesting functions. Some liken the distinction between hardware and software to that between body and mind, or physiology and psychology. Computer hardware specialists are those who concern themselves with making the machine faster, smaller, and more reliable, as well as developing and using the component parts that will make this possible. Software specialists are interested in developing powerful new computer languages, more efficient systems to control computer operations, or elegant programs to accomplish some new task, solve some new problem, or analyze huge amounts of data.

The history of hardware, already discussed briefly, illustrates the development from the abacus through various primitive calculating devices to today's computers and robots. The history of software is really that of logic, mathematics, and practical approaches to problem solving. A proof in logic, the construction of the proof of a mathematical theorem, or the rules for building a house or baking a cake are all examples of solutions to problems. These solutions are systematic, rule-abiding, and lead from a problem state to a goal—the desired proof or problem resolution. The existence of the hardware of a computer, properly constructed, leads to the possibility that many such proofs, mathematical operations, and decision-based problem solutions can be performed by a machine instead of a human. Once the human expresses the required solution steps in a way compatible with the computer's organization, the steps can be carried out automatically without further human intervention.

So far, computers have not been used to build houses (though they have certainly been used extensively by architects to help in building design) or to bake cakes (though the household robot of the next decade may do just that), but they have been used to perform many repetitive tasks that are too time-consuming or too complex for a human to accomplish. For example, a computer was essential to the proof of a mathematical hypothesis—the Four-Color Theorem—that had eluded all human mathematicians for 124 years. The University of Illinois mathematicians (Haken and Appel) who proved the theorem admitted that it would have been impossible without the aid of the computer—a total of 1200 hours of computer time was enlisted in supporting the proof. They even remarked that the computer had developed more clever strategies than it had been given by them through the programmer.

Our purpose in this book is to give the reader the ability to communicate problem solutions to the computer in a language (FORTRAN) that can be readily translated into something the computer "understands," and then let

the machine do all the hard work of actually implementing the solution steps to arrive at an answer. Once this skill is developed, it can be applied to facilitate solution of a wide variety of problems. Supercomputers today solve complex problems in nuclear physics, airplane design, quantum chemistry, meteorology, oceanography, and the like, and they are primarily FORTRAN-driven machines.



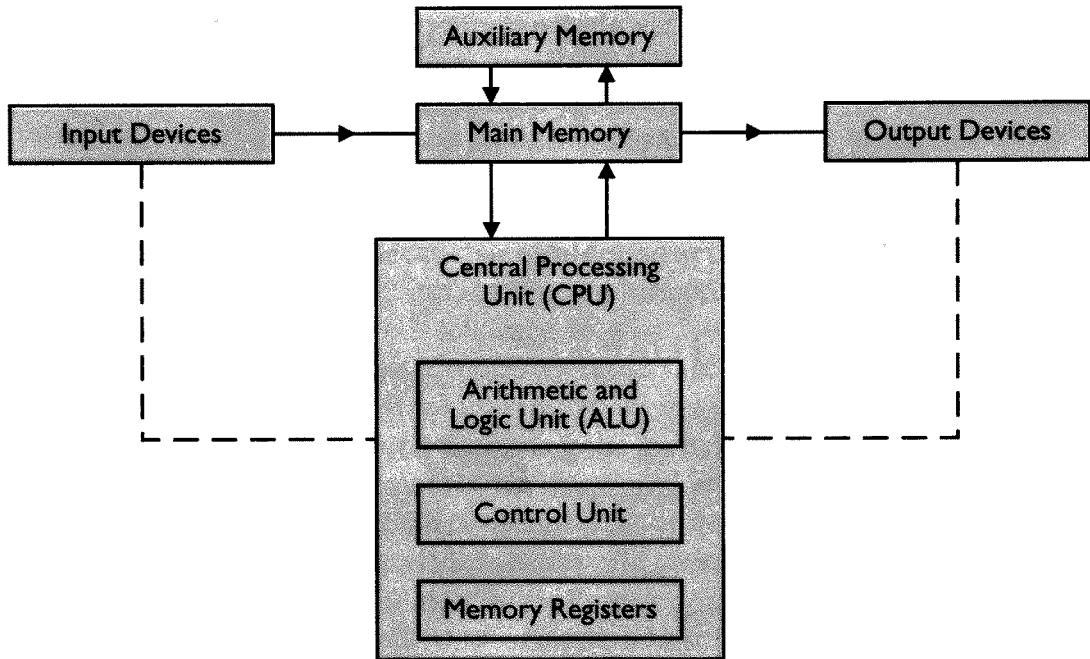
COMPUTER STRUCTURE

The computer has a Central Processing Unit (CPU), which is the "brain" of the machine, controlling all of its functions. A CPU has three basic components—an Arithmetic and Logic Unit (ALU), in which all of the high-level computations are carried out; a Control Unit, which synchronizes all of the operations executed by the computer; and a small set of Memory registers, for holding the values currently being operated on by the ALU. The CPU controls Input and Output devices, and sends results to output devices as a result of executing the appropriate commands.

Input devices may include a terminal keyboard, punched card reader, teletype, phone line, magnetic tape or disk, "mouse," or mass storage device. Output devices may include a terminal screen, high-speed printer, plotter, card punch, magnetic tape or disk, or mass storage device. There is an extensive main memory in which the computer program and all of the values it operates on or computes are stored, and there may be supplementary auxiliary memory for additional storage capacity. A schematic diagram of the simple computer organization might be as follows on the next page (where solid lines indicate flow of information and dashed lines indicate flow of control).

The Central Processing Unit will execute a program that is resident in memory, one simple operation at a time. It will "fetch" values stored in locations in the main memory, perform the appropriate arithmetic or logical (comparison) operations on the values, and "store" results into other locations of memory. If input is required by the program, the control unit will activate access to the appropriate input device, and when output of the results of the program is required, it will be sent to the appropriate output device (usually terminal screen, magnetic disk, or high-speed printer). This simplified model can become much more complex if many terminals are networked together through one CPU, or several CPUs are networked together, or if we have parallel processing carried on by several ALUs.

The computer *memory* is usually organized into fixed-size locations, or "words." Each word is made up of a number of *binary* (two-state) devices, and the number of these, or the length of the word, determines how large a numeric value may be stored, the length of a character string that may be



stored, or the precision (number of significant digits) of a real number that may be stored. Words on microcomputers are generally 8 or 16, sometimes 32, *bits* (binary digits—0 or 1) long. On larger computers, words generally range from 32 to 64 bits (with 32, 48, 60, and 64 bits being the most common word lengths).



BINARY REPRESENTATION AND STORAGE

The representation of values on a physical computer is not in the *decimal* (ten-valued) form we are used to, but in *binary* form. For a number of the media used for input and output of computer information, the binary states are simple to describe. For paper tape (which was used on earlier systems to save programs) or punched cards, at each location there will either be a hole or no hole. For magnetic tapes, disks, drums, or mass storage devices, at a particular point there will either be a magnetized spot or no spot. A switch is either open or closed; a wire either carries current or no current. A light bulb or a vacuum tube, as in the ENIAC, is either on or off.

We have seen the binary nature of media that may be used to bring information into and out of the computer. Internal to the machine itself, values are also represented by groups of binary devices. In the ENIAC, vacuum tubes were used. Soon they gave way to magnetic “cores,” tiny iron ferrite doughnuts about the size of the head of a pin, which could be polarized in one of two directions, clockwise or counterclockwise—another binary device.

These cores were used for memory storage so long (until the mid- 1970s) that memory came to be referred to as "core." The magnetic cores were largely replaced by semiconductor memory, composed of transistors. The binary states are determined by two different voltage levels in the transistor. More recently, integrated circuits have been employed, giving rise to VLSI (Very Large Scale Integrated circuit) technology, and occasionally "bubble memories" have been used as well.

A single binary device would not allow us to store very many interesting values—just 1 or 0, representing the two states of the device. However, if we group several bits together, we can represent a wide variety of values with such bit patterns. We can represent integer numbers in base two (binary) in a manner analogous to that we use for decimal values—a positional notation in which each digit represents a multiplier times a power of the base. In base 10, the number 365 represents the value 3×100 (or ten squared) + 6×10 (or ten to the first power) + 5×1 (or ten to the zero power). Similarly, in base two (binary), the bit string 1101 represents 1×8 (i.e., 2^3) + 1×4 (2^2) + 0×2 (2^1) + 1×1 (2^0), which corresponds to the decimal value 13. In a similar, but more complicated, manner, real numbers can be represented as binary multipliers times a base to some power (also stored in binary). It is simple to represent signs of numbers—a 1 may mean minus and a 0 plus (when set as the *leftmost* bit of the storage word).

Character values can also be represented in binary. It is merely a matter of having a unique binary string (called a "byte") for each symbol you want to represent. How long the bit strings for each symbol must be depends on how many symbols you want to be able to represent. Common character codes are 6-bit codes (BCD—Binary Coded Decimal, or Display Code), which can code 64 symbols; 7-bit codes (ASCII—American Standard Code for Information Interchange), which can code up to 128 symbols (2 to the 7th power); and 8-bit codes (EBCDIC—Extended Binary Coded Decimal Interchange Code), which could represent up to 256 characters. This gives you a very brief overview of how values may be represented internally in the computer, or in patterns of magnetic spots written out to tape or disk.

A different binary representation is used on a punched card. Though the use of such cards is becoming rare these days, this means of using binary patterns to represent values is of at least historical interest. A punched card has twelve "rows" (12-row, 11-row, and rows 0 through 9) and 80 "columns." Each column is used for punching the pattern representing a particular digit or other character. A pattern of holes distinguishes each character, but since there are twelve rows, and holes may be placed in any combination of these rows, we only need one or two punched holes to represent each character uniquely. Each digit is represented by a hole in the appropriate row ('1' by a hole in row 1, and so on); the other characters are represented by a pair of holes in their column—for example, an "A" is represented by a 12-punch (in the top, or 12th, row) and a 1-punch, and so on.

The details of such bit representations of values need not concern us at this point, though an experienced programmer will eventually become familiar with them. It is relevant to our purpose at this stage to know that such values can be represented in the computer and manipulated by programs whose instructions will also be stored in binary. The *word size* on the particular computer you are using will determine how large an integer value you may store, the range of real values that can be represented, and the number of characters that can be packed into one word. For example, since an integer must be represented as a binary pattern including a sign, one bit must be used for the sign. On a machine that has a 32-bit word this leaves 31 bits for the magnitude, and thus the largest value that can be represented is:

$$2^{31} - 1 = 2147483647$$

On a 60-bit machine (such as a CDC), the largest integer is:

$$2^{59} - 1 = 576460752303423487$$

An attempt to calculate a result *larger* than these limitations would result in an error called an “overflow.”

Similarly, the word size on the computer influences how large a real value may be stored (determined by how many bits are used to represent the exponent) and how many digits of precision the value has (determined by how many bits are used to represent the multiplier). As a simple analogy, imagine a ledger with only six columns for storing numbers. If we were to store a signed integer in the ledger, the largest value we could store would be

$$10^5 - 1 = 99999$$

If we also used the ledger to store real numbers, with fractional parts, and if we decided to use the leftmost 4 columns for the multiplier (with the decimal point understood to be at the left of the multiplier), with the remaining 2 columns for the exponent (assuming that both multiplier and exponent are positive), the largest positive real value we could store would be

$$\begin{array}{r} (.) \\ \swarrow \\ 9999|99 = .9999 \times 10^{99} \end{array}$$

Thus the limit on the size of positive real numbers that could be stored would be on the order of 10 to the 99th power, and they would be precise to no more than four significant digits.

As an example, the following patterns indicate how integer, real, and character values would appear stored in a 32-bit word on a machine such as a VAX or IBM. Because the characters are stored in 8-bit “bytes”—unique bit patterns for each symbol to be represented—up to 4 characters can be stored in one word. Details explaining the “why” of this are covered in Appendix A.



COMPUTER LANGUAGES

We have seen how values can be represented in the computer, but now we must concern ourselves with how they can be operated on and tested to make the computer more than just an oversized hand calculator. We must be able to instruct the computer to perform various tests and operations, such as addition, as well as to input or output information. The problem of how we can do this is addressed by computer languages.

Machine Language

The computer can be instructed to perform arithmetic operations or logical comparisons, or to perform input or output, but it must be instructed to do so in its *own language*, that is, in binary. The appropriate binary devices must be set to the unique values which represent each command. Thus early programmers had to learn *machine language*, that is, the representation of the commands to add, subtract, store, load, and so on, as binary strings. Numeric values also had to be put into their binary representation to be used by the machine.

Programming was a painstaking and frustrating experience, very far removed from the human level of thinking through a problem. Each step in a solution had to be expressed in terms of the very simple commands available on the computer, and so a step which was seemingly simple from a human perspective became many machine language steps.

Assembly Language

For the programmer, probably the biggest difficulty with writing and correcting a machine-language program was to get an overview of what the code was actually *doing*, since strings of binary values all tend to look much the same to a human, and the meanings of “add,” “store,” and so forth would not jump out from the morass of 0s and 1s. Thus the next evolutionary step in programming was to allow the programmer to write sequences of commands in a shorthand notation that was much more meaningful at the human level. *Assembly*

language commands were generally expressed as short (often three-letter) *mnemonic* instructions such as ADD, SUB, MUL, STO, and LOA. Generally, numbers could also be expressed in their decimal form, and names could be used for the locations or memory registers involved in an operation to simplify references to them in terms of their binary addresses.

A solution still had to be expressed in terms of the very simple operations available on the machine, so most assembly language instructions mapped one-to-one onto machine language instructions. However, there were a few *macro* expressions—used to replace a common sequence of instructions such as those for Input/Output—that translated into several machine language instructions. A programmer still had to think of a problem solution more at the machine level than the human level, in terms of the machine's simple available operations; but at least meaningful names could be given to the operations, and values looked like familiar human-notation values.

Of course, the computer could not understand strings such as ADD, STO, 65, and 3.14159, so there had to be an intermediate *translator* program, which converted these instructions and values into equivalent binary strings—such a program is called an *assembler*. Assembly languages are still available on almost all machines, and programmers use them when they want to get closer to the machine's basic capabilities, or use it more efficiently, than a high-level language such as FORTRAN or COBOL or Pascal may allow. However, assembly language programming is still difficult and taxing, and is generally only used for special applications.

Carefully examine the following simple “programs,” the first two in a hypothetical machine language and assembly language, and the third in FORTRAN, to accomplish a job. A comparison of the programs may help you appreciate the simplicity available in a high-level language such as FORTRAN.

The problem is to add together four integer numbers—3, 5, 6, and 8—which have already been stored in four locations on the machine. In machine language, we will have to refer to the *addresses* (numbered positions) of the locations in which they are stored (represented in binary), but in assembly language and FORTRAN, we will be able to refer to the locations by symbolic

Name	Address	Contents of Location
I	001	000011
J	010	000101
K	011	000110
L	100	001000
N	1100	000000 (initially)

names we have given them: I, J, K, and L. We want to store the sum in a fifth location, numbered 12 (binary 1100) and called N.

Let us assume that our machine language has an instruction that will add together the contents of any two locations and store the result in a third location. The machine-language instruction for this add command is the six-bit string 101011; it is to be followed by the (binary) addresses of the two locations whose contents are to be added together, and then the address of the location into which the result is to be stored. Let us also assume that all addresses in our hypothetical machine are six bits long (which allows the addressing of 64 different locations). The assembly language code for the addition instruction will be ADD, and it will be followed by the symbolic names of the three addresses involved. Each instruction we write must also occupy a location in the machine, and have an address. In the assembly language, we may give the location a symbolic name if we want to refer to it.

Thus, to write our program to add together the four stored values indicated, we will write binary instructions and their parallels in assembly language. We will first add I and J together and store the result in N; then we will add K to N and finally L to N. The programs are as follows:

101	101011000001000010001100	ADD I, J, N
110	101011000011001100001100	ADD K, N, N
111	101011000100001100001100	ADD L, N, N

You can readily see the advantage, from the human viewpoint, of the assembly language program. In FORTRAN, we can simply write

$$N = I + J + K + L$$

to accomplish the same task.

High-Level Languages

As computers became more widely used, the demand for easier accessibility to their capabilities increased greatly. Not everyone wanted to become a machine- or assembly-language programmer, but many still wanted to be able to use and control the computer themselves, without having to rely on someone else. This demand gave rise to the proliferation of a multitude of *high-level languages* that were closer to human representations of problems and their solutions and were designed for various purposes. There began, and still continues today, a veritable "Tower of Babel" of languages to be used on the computer, but only a few of the early ones, or even the more recent, have survived the test of time. FORTRAN was the earliest high-level language to be developed that is still in wide use today (though in modified and improved form). It was followed quickly by LISP, a list-processing, recursive language

used in artificial intelligence applications, and then by COBOL, still the primary language used for business applications.

There still is a multitude of languages, some very special-purpose, such as APT (for Automatically Programmed Tools) to control machining operations. The most significant of the more recent languages developed have been the following: BASIC, a teaching language developed at Dartmouth by Kemeny and Kurtz, and available on almost all microcomputers today; Pascal, a language designed to encourage well-structured programming, and largely adopted as the *lingua franca* of the computer science academic community; Modula-2, a possible successor to Pascal in this capacity; Ada (named after Lady Lovelace), supported by the Department of Defense for programming control of "embedded systems" such as those used for missile guidance; C, developed at Bell Labs, which is a scientific language with similarities to FORTRAN and Pascal, but which also implements many assembly-language capabilities; and "object-oriented languages" (discussed later) such as C++.

Even with the many languages available, FORTRAN is still the primary language used by the scientific community, and will continue to be so for many years to come. Prominent members of the computing community have remarked that the scientific language of the next century will be some version of Fortran.

A high-level language incorporates commands that are closer to human notation, such as algebraic equations or simple English. These allow the programmer to express a program for solving a problem in a notation closer to that generally used in problem solving in the human realm. The importance of a good notation to problem solving has long been recognized, as the adoption of Leibniz' notation facilitating the use of the calculus will attest. Of course, algebraic-like expressions such as $A = B + C$, or English-like instructions such as PRINT, are not meaningful to the computer. A translator program, generally called a *compiler*, is needed to translate such high-level language instructions into binary. Sometimes an *interpreter*, which translates the program one line at a time into executable machine code, is used instead; but for FORTRAN compilers are generally used, which translate the program as a whole.

The compiler is generally supplied by the manufacturer of the particular computer in use, and it will have its own special features and means of implementing high-level language commands in machine language. The FORTRAN program is referred to as the *source code*, and the binary instructions generated by the compiler are called the *object code*. In order to *execute* the object code, it must be *loaded* (by a program called the *loader*) into computer memory, and then *linked* (by the *linker*) with any system functions or utilities that are required for the full operation of the program. It is desirable for all compilers for a specific widely-used high-level language to conform to some standard (sometimes set by ANSI, the American National Standards Institute), so that all compilers for the language will have a definite set of the same

features, implemented in the same way from the programmer's point of view. This consistency is a great advantage in a language, and FORTRAN was the first language to conform to such a standard.

Such standardization of a language allows for program *portability*, the ability to use a program written in the language and following the standard on any machine with a compiler for that language. Thus a programmer should be able to write a FORTRAN program that conforms to the standard on an IBM machine, but then take it around the country and run it on any machine with a standard FORTRAN compiler. This feature of a language is very desirable, since programs are not intended to run on just one class of computers, and the techniques and regulations you learn when becoming acquainted with a language should not just be applicable to one machine or group of machines. The portability (standardization), efficiency, and simplicity of FORTRAN have all contributed to its longevity as the major scientific language.

For a long time, programs were created and saved on paper tape or punched cards, then put into the computer, compiled, loaded, linked, and executed, all as steps in one process which, once the program was submitted, was beyond any control by the programmer. This process was called *batch* mode. If there were any syntax errors in the program detected by the compiler, the rest of the process was terminated, and the programmer had to wait for the run to come back before making any corrections and resubmitting. This was a slow and painful process.

On more modern systems, most programs are developed at a terminal which is connected to a large machine which is "time-shared" by many users, or on a microcomputer over which the user has exclusive control. A program is developed and submitted for compilation, and if there are any errors, the programmer receives immediate notification of the problems at the terminal. Thus any errors can be corrected and the program resubmitted for a new compilation. The whole operation is much more efficient than was batch mode, and greatly enhances the program development process. Furthermore, when the program executes, information can be entered for the program to process, or to control its various operations, while it executes. This is called *interactive* mode. In batch mode, all input data had to be completely prepared ahead of time and submitted with the program, and there was little possibility of programmer control of the program during its execution.

FORTRAN was the first high-level language to be standardized, and this portability contributed greatly to its early success and wide adoption. It is very desirable to be able to take your programs and talents and transport them elsewhere, if the need should arise. Thus, in this book, the ANSI Standard features of FORTRAN 77 will be primarily emphasized. Only occasional reference may be made to additional features which may be available on some FORTRAN compilers, so that you are aware of their existence and utility, should they be available on the machine you are using. However, developing programming habits that conform to the ANSI Standard will be encouraged throughout.

Beyond High-Level Languages

These days, many people use computers who do not use assembly language or any of the high-level languages. They make use of *software packages* that someone else has written in a high-level language or in assembly language; all the user has to do is learn the proper set of simple commands that will make the program work. Examples of such packages are word processors, spreadsheet programs, database manipulation packages, and statistics packages such as SPSS (for the Social Sciences). It is very handy for a person in today's society to know how to use this prepared software, but it is also very important to know how to use a language like FORTRAN, which will allow you to write a program to solve a problem for which no package exists.

BRIEF HISTORY OF FORTRAN

Since the generally-available FORTRAN language has now been around for over 30 years (longer than any other high-level language), its history deserves examination. Such a perspective on the language and the evolution of its features will make the reader more familiar with the language and its major intent, that of solving scientific problems simply and effectively. A historical perspective on the language is especially useful if programs written under earlier versions are encountered and must be dealt with, or compilers incorporating an earlier version than FORTRAN 77 are all that is available in a particular circumstance. There are many FORTRAN production programs in existence that were written years ago, but are still in active use and will need to be maintained.

The development of the first FORTRAN compiler began in 1954 at IBM with a team under the direction of John Backus. A working version was made available in 1957. It was called FORTRAN to stand for "FORmula TRANslator," implying that it was to be used to translate scientific or algebraic formulas into a form the computer could process. As with other languages whose names are acronyms derived from some phrase, the convention has been to write the name of the language all in capital letters; we will adopt this convention here. Similarly, COBOL (for COnmon Business-Oriented Language), developed in 1960-1962, is written all in capitals, but Pascal, developed in the mid-1970s by Nicklaus Wirth, is not written in capitals, since it is named after the French mathematician and philosopher, Blaise Pascal.

The first version of FORTRAN was written primarily for a particular IBM computer (the 704), and many of the language's peculiarities (such as the length of variable names and the Arithmetic IF) are derived from the nature of that machine. As soon as the first version was made public, the group at IBM was working on a revised and improved version and, in the next year,

FORTRAN II was released. That version added the Logical IF (we will discuss this in Chapter 3) and, very importantly, subroutines and functions (Chapter 9), as well as improving many other features of the language. One of the great advantages of FORTRAN is that its versions have generally been “upwardly compatible,” so that the previous version is included as a subset of the new version. In this way, programs written under the earlier version do not have to be revised to run under a new compiler. The newer versions have simply added new features.

A FORTRAN III version was developed “in-house” in IBM, which included allowing assembly language instructions to be intermixed with FORTRAN statements, but it was too tied to a particular machine model and its assembly language, and was never released to the general public. FORTRAN IV was released in 1962 and enjoyed a long and productive life. It introduced logical types (to hold True and False values) and the ability to pass subprograms as arguments to other subprograms. Unfortunately, because many different dialects of FORTRAN IV were developed by different manufacturers, programs that ran on one machine might not run on another. The American Standards Association (ASA) immediately became interested in seeing that FORTRAN was standardized, so that it would run equally well on any machine with a FORTRAN compiler.

IBM briefly had versions of a FORTRAN V and a FORTRAN VI, attempts to include both scientific and business applications in the same language, but these versions did not really become available to any great extent outside of the company. FORTRAN VI was transformed by IBM into a new language, which they marketed under the name PL/I, keeping FORTRAN IV primarily scientific.

By 1966, ASA developed a standard for FORTRAN compilers, generally referred to as FORTRAN 66. This version remained in active use until the introduction of the FORTRAN 77 Standard in 1978, and even after that. So many years had been spent using FORTRAN IV/66 compilers that there was a reluctance to change to the new Standard. Thus the new version was introduced gradually; FORTRAN 66 compilers remained available for a long time on many large-scale machines, and some of the FORTRAN compilers available for microcomputers used essentially FORTRAN 66 features. Thus, we will note throughout this text the features that were new in FORTRAN 77, just in case you deal with older programs or compilers. All of the jobs that can be done in FORTRAN 77 can be accomplished in FORTRAN 66 (and most of them even in FORTRAN II); it is just that they can be done more elegantly in FORTRAN 77. In 1969, ASA changed its name to the American National Standards Institute (ANSI), and from then on, versions of FORTRAN and of other standardized languages are referred to as “ANSI Standard.”

The FORTRAN 77 Standard, released in 1978, added important *structured programming* constructs such as the IF/THEN, the IF/THEN/ELSE, and the ELSEIF (Chapter 3) to the existing FORTRAN. These structured constructs

allow the programmer to write code in a uniformly accepted form that can easily be integrated into large-scale programs. It also standardized the handling of character data, and introduced the use of symbolic constants. Arrays under FORTRAN 66 were restricted to three dimensions, but under FORTRAN 77 they may have as many as seven. Except for the handling of character data, upward compatibility was still maintained, and programs written under FORTRAN 66 could still run under the new FORTRAN 77 compilers. DO loops are handled somewhat differently in FORTRAN 77 (see Chapter 4).

There has been for some time now an effort under way to develop a new version of FORTRAN, an effort which began in 1978, as soon as the FORTRAN 77 version was released. The new version was initially referred to as Fortran 8x, but more recently (as the development phase went into 1990) as Fortran 90. Its developers have preferred to capitalize only the first letter of the language name. We will adopt this convention when referring to the new version of the language, but keep the familiar FORTRAN for earlier versions, up through FORTRAN 77. The new version will add many interesting and powerful features to the language, while attempting to maintain upward compatibility as much as possible. Where appropriate, throughout the text we will mention new features that will be added when Fortran 90 compilers become available, so that you will be ready to use them.

The new Fortran 90 Standard also designates certain FORTRAN features as "obsolete," indicating that they are no longer necessary given the newer features available, or that their use encourages bad or dangerous programming practices. These features will still be available under the Fortran 90 compilers, but they are targeted for possible removal in the subsequent Fortran version which, at the rate these things seem to progress, would probably not be completed until at least the year 2000.

In this text, we will discuss the historical development of the language features in FORTRAN, indicating why certain new features were added to improve the program design capabilities available to the programmer. For example, any program written with the newer Block IF statements, or even the Logical IFs, could have been accomplished using only the original Arithmetic IF and GO TO statements; but the newer programs are much easier to read, maintain, and incorporate into new, larger projects. In our discussion, the general superiority of the newer structures over the old ones will become apparent. This will help you understand why some of these older features are now considered outmoded by the modern programming community.

This book will emphasize the ANSI Standard features of FORTRAN 77 and encourage you to develop programming habits which adhere to the Standard. This will mean that your FORTRAN programs, and your skills, will generally be transferrable to any computer environment that includes FORTRAN. This has been one of the great strengths of FORTRAN—its general availability and ease of portability to different installations. This, and its long life, account for the continuing popularity of this language in the scientific community, as well

as the existence of a huge library of existing, functioning FORTRAN programs. We will occasionally mention non-Standard features that, if available on your system, may be useful to you, but you should not expect to be able to carry over their use or any programs that use them to other computer installations. We will also continue to look ahead to the new features and changes to be expected in Fortran 90, so that the transition will be easy for you. The new vector and matrix operations available in Fortran 90 will be of particular interest to scientific users of supercomputers.

SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

In this Introduction, the following new terms and names relating to computers and programming in FORTRAN have been discussed:

Babbage, Ada (Lady Lovelace), Hollerith, ENIAC, "stored-program" computer, EDVAC, hardware, software, CPU, ALU, memory, core, bit, byte, ASCII, EBCDIC, machine language, assembler, compiler, source code, object code, load, link, batch mode, interactive mode, portability, Backus, ANSI, and Fortran 8x/90.

EXERCISES

Note: Certain problems marked with a "bullet" (circle) have their answers provided at the end of the book.

1. Who was Charles Babbage, and what is his importance to the history of computers?
2. Who was Herman Hollerith? You will want to remember him when representations of character strings are discussed.
3. Describe the ENIAC and its importance.
- 4. What was the advance embodied in the EDVAC?
5. Classify each of the following appropriately as hardware or software: card reader/punch, video display terminal, compiler, operating system, FORTRAN program, line printer, integrated circuit, mathematical proof, tape drive, disk drive, data.
6. What are the component parts of a CPU?

7. What is the binary representation of the decimal integer 18? of -18? [Recall that we mentioned that one bit is used for the sign of the number. This is the *leftmost* bit, and is a 0 (zero) for positive values, 1 (one) for negative.] Indicate how it would be stored on a machine with an 8-bit, 16-bit, and a 32-bit word.
- 8. Another way of representing negative values is in *one's complement* form. The one's complement of a bit string is simply a sort of inverse "mirror image" of the string, with all of the 1's changed to 0's, and all of the 0's changed to 1's. Indicate how the value -25 would be represented in one's complement form on a machine with an 8-bit word; on one with a 16-bit word.
 - 9. A third method of representing negative values (and useful in one efficient way that subtraction is performed on the computer) is the *two's complement* form. The two's complement of a value is simply the one's complement (see question 8) plus 1. (Note that in binary addition, $1 + 1 = 10_2$, with a sum digit of 0 and a carry digit of 1.) Show how the values -18, -25, and • -36 would be stored in two's complement form in a 16-bit word.
 - 10. What is the largest (unsigned) integer value that can be stored in a 7-column decimal ledger? In an 8-bit binary word?
 - 11. Many supercomputers use a 64-bit word. If one bit is used for the sign of an integer, what is the largest integer value that can be stored on a supercomputer? (You can check your result against this value to be found in Chapter 5.)
 - 12. Imagine a hypothetical machine that has a 16-bit word. If the bit pattern (numbered 0 - 15, from right to left) for storing reals on this machine is divided up in the following way:
- leftmost bit (#15) is the sign of the number
 next bit (#14) is the sign of the exponent (a power of 2)
 next six bits (#13 – #8) are used for exponent (of 2)
 last eight bits (#7 – #0) are used for the multiplier (also called the mantissa), with a binary point understood to be to the *left* of the multiplier, and with the convention that the *leftmost* bit of the multiplier must be a 1 (to assure the greatest possible precision)
- then determine what the largest and the smallest real values are that can be stored on this machine. A value is interpreted as:

$$\pm 0.\text{multiplier} \times 2^{\pm\text{exponent}}$$

13. Redo problem 12 assuming that the understood binary point is at the *right* of the string representing the multiplier.

- 14.** Distinguish clearly among machine language, assembly language, and a high-level language such as FORTRAN.
- **15.** Write an “assembly language program” (using our simple language model in the text) to store the result of this algebraic equation in location Y, assuming that values have already been stored in locations A, B, C, D, and X. Assume that you have assembly instructions SUB, MUL, and DIV, which operate the same way as ADD. The algebraic equation is:

$$y = ab - c/d + x$$

(In FORTRAN, it can be simply written as: $Y = A*B - C/D + X$)

16. Write an “assembly language program” that will calculate the average of four values stored in locations A, B, C, and D, and store the result in location Y.

(In FORTRAN, this can be done simply by: $Y = (A + B + C + D)/4$)

17. Distinguish between source code and object code, between batch mode and interactive mode. Who originally developed FORTRAN, and what are its various versions?



SUGGESTED READINGS

American National Standard Programming Language FORTRAN X3.9-1978. New York: American National Standards Institute, 1978.

American National Standards Institute (X3J3, the Fortran Technical Committee), *Fortran 90: X3J3/S8.118*. May 1991.

Appel, Kenneth, and Wolfgang Haken. “The Solution of the Four-Color-Map Problem.” *Scientific American* (October 1977), 108-121.

FORTRAN’s Twenty-Fifth Anniversary. Special Issue. Annals of the History of Computing 6, no. 1 (January 1984).

Haken, Wolfgang. “An Attempt to Understand the Four-Color Problem.” *Journal of Graph Theory* 1 (1977), 193-206.

Horowitz, Ellis. *Programming Languages: A Grand Tour*. New York: Computer Science Press, 1983.

Lazou, Christopher. *Supercomputers and their Use*. Oxford: Clarendon Press, 1986.

Matsen, F. A., and T. Tajima. *Supercomputers: Algorithms, Architectures, and Scientific Computation*. Austin, Texas: University of Texas Press, 1986.

Sammett, Jean E. *Programming Languages: History and Fundamentals*. Englewood Cliffs, N. J.: Prentice-Hall, 1969.

Wexelblat, Richard L. (ed.) *History of Programming Languages*. New York: Academic Press, 1981.

CHAPTER I



APPROACHES TO PROBLEM SOLVING

A good problem solver needs a sharp mind, a knowledge of the subject area, a bag of “tricks” (tools), the ability to choose the *right* tool, and a well-organized approach. Most interesting problems do not have simple solutions. One effective strategy is to “begin at the beginning” or “to start at the top and work down.” Such a *top-down* approach advises you to look at the big picture first and work out the details later. Problem-solving design tools are introduced in this chapter that will help you to work out the details in FORTRAN.

Han Solo attempts to repair
the grounded Millenium
Falcon in *The Empire Strikes
Back*.

"All progress is precarious, and the solution of one problem brings us face to face with another problem."

- Martin Luther King, Jr., Strength to Love

In the course of your life, you will meet a great many problems: deciding on a career, which job offer to accept, whether to marry or live a carefree single life, finding a place to live, perhaps even learning how to cook a gourmet meal. In the course of your experience in programming, you will also encounter a wide variety of more clearly defined problems, which you also need to solve. In both of these situations, it is critically important that you approach the problem before you with a clear head, with all of the relevant information available to you, and in a logical and systematic manner.



DEFINE THE PROBLEM

Some problems you deal with are of your own creation, and some are given to you from some outside source, such as your instructor or your boss. In any of these cases, the first thing you must do is to make sure that the problem is clearly stated, that there are no ambiguities about what is available or what is expected, and that the problem is possible to solve within reasonable parameters. If you do not begin with a careful analysis of these aspects of the problem, the whole endeavor is in serious trouble.

If your problem is to "cook a gourmet meal," then there are many questions you must ask at the outset: who, what, where, when, and why, to follow the journalist's five Ws. *Who* (and how many) are you inviting to dinner? *What* will you serve? *Where* will the production be staged (and thus what facilities will you have available for preparations)? *When* will the gala event take place? And finally, *why* are you doing this? Once these questions have been answered, others may arise. Are there any special restrictions on your problem, such as foods you cannot serve because of allergies, religious beliefs, or moral convictions? All such questions regarding the problem must be resolved and clarified first; then you can begin the solution.

If, instead, you have a problem that requires a computer solution, you must be just as careful to get a clear statement of the problem, resolve all ambiguities, and answer all questions that relate to the nature of the problem. If your instructor asks you to write a sorting program to rearrange data into a particular order, you must begin by asking a number of questions. Is the information to be sorted character data, to be put into alphabetical order, or is it numeric data? If it is numeric, are the values whole numbers or reals, and

is it to be sorted into ascending or descending order? Then you need to know something about the source of the data—will it be on punched cards, or on a magnetic tape, or on disk, or must it be entered from the terminal? Since the input details of your program will differ depending on the answers to these questions, you need to establish these conditions clearly at the beginning. You also should ask questions about the nature of the data itself, such as how much is there, how will you know when you have finished reading it, and is it already in partially sorted order?

It is likely that more questions will occur to you as the problem-solving process begins, and it is important to get answers to these questions as soon as possible. But it is best to ask as many questions as you can at the very beginning, so that you are proceeding “on the right track.” Otherwise, if there is a serious misconception at the beginning of the process, then much work may be wasted before it is corrected. Imagine—in the case of preparing the gourmet dinner—if you thought it was to be for 100 people, when it was really only for 10!

What is required here is a clear *specification* of the problem, laid out in a systematic, unambiguous manner. All inputs to the process, and all conditions that are to be met, should be clearly spelled out. Another important aspect of such a specification is that the desired nature and form of the output is understood.



CLARIFY THE INTENDED RESULT

To understand the starting conditions is important, but even more important is to have a clear idea of what *result* you are striving to attain. All too often considerable effort goes into solving the *wrong* problem. Be sure that you know what you want, or what your customer wants, and that you are clear on the required form of the result. To use a common-sense analogy, your customer might wish to put a competitor out of business. One possible way to achieve this general result would be to so greatly improve your customer’s product and/or advertising that the competitor could no longer keep up. Another possible way would be to blow up the competitor’s business. In such a case, perhaps considerations other than pure profit should determine the form of the solution.

Sketch a rough outline of the form the result(s) will take, assess it for adequacy, and show it to your customer to see if it will be satisfactory. Make a similar sketch of the form of any input to the process, to be sure that you adequately understand its nature and format. If you are creating the input data (if any) yourself, then you have control over its form, and you only need to be consistent. However, often the information to be processed already exists,

and then you must be aware of the kind of storage medium (tape, disk, punched cards), and the data's format, since your program must accommodate these.

This overview is necessarily general, since it is meant to apply to computer problem solving in general. As we get into specific problems, we will discuss solving complex problems in detail, employing the considerations we have outlined here.



ORGANIZE THE APPROACH SYSTEMATICALLY (TOP-DOWN DESIGN)

Up to this point, we have treated the need for careful analysis of the problem's starting conditions and input data (if any), and the results/output desired—that is, the *goal state of the process*. Between the starting conditions and the end state comes most of the interesting work in solving the problem. You have begun by considering the process as a "black box," whose function is understood by the relationship between its initial state and its goal state (or between its input and its output). Now the details of the contents of the black box have to be attended to. An analogy might be taking your car to the mechanic; when you take it in, it does not run well, but you take it out as a finely tuned machine. The really interesting process is what goes on in between these two states, but you do not need to know the details of how it is accomplished (the contents of the "black box"). However, a time might come when you will need to do the mechanic's job yourself (you have no money, perhaps). At that time, you will have to attend to the details.

The best way to organize finding the solution to a large, complex problem is to break it down into smaller, manageable problems. This is often referred to as *top-down design*, and some refer to it as a "divide-and-conquer" technique. A simplistic analogy is the problem of moving a two-cord load of wood into the house so that you can burn it in your woodstove to keep warm in the winter. It seems impossible to move the entire load at once, but if you divide it into smaller loads and make several trips, you can accomplish the job easily. Of course, the subdivision of most large problems into smaller problems is not so simple or so linear. Some of the smaller problems may still be too big, and may need to be subdivided; tests and repetitions may have to be carried out; and so on. Actually, our "simple" wood-moving process involved a repetition—we divided the big job into small uniform jobs, and then repeated the small job until the load was moved into the house. This is a sort of "repeat-until" logic we will encounter again as we begin to solve problems in FORTRAN.

Once the large problem has been divided into smaller problems, each of the smaller problems has to be solved. This procedure is referred to as *stepwise*

refinement. But before we jump into solving each of the smaller problems, we should be sure that our overall organization of the problem is correct. To do this, it is beneficial to back off and get a perspective on the whole problem and its organization. There are various techniques that can be helpful in looking at the problem organization and in expressing the details of problem solution. We will examine a variety of such techniques and then adopt one or two for use in this text. If you find that you prefer one of the other techniques available, you are certainly free to adopt the method that best suits you. What is important is to work toward good problem-solving design.

ATTENDING TO THE DETAILS (STEPWISE REFINEMENT)

We have already talked about the need to organize your thoughts, to make clear the desired relationship between the starting conditions (or input) and the goal (or the desired results, or output), and then to approach the solution as a connected group of logical steps or subproblems. At some point, the process of subdividing comes to a halt, and we have reached the “atomic” level of steps that can be accomplished in one operation or in a simple procedure. We will be examining the simple atomic operations that are available to us in FORTRAN (similar operations are available in other high-level programming languages). Our ultimate goal will be to reach this level, use the operations correctly, and from them compose the building blocks for the solution of the problem at hand.

Algorithms

An *algorithm* is a finite sequence of instructions, or steps, that comes to a conclusion or result in a finite amount of time. The word “algorithm” comes from the name of a ninth century Arabic mathematician, Al-Khowarizmi; it was also through the translation of his work on algebra that Arabic numerals became known in the West. The development of a good, efficient, working algorithm is the very crux of effective problem solving. Because there may be many possible paths from our initial conditions to the desired goal state, it is important to choose the best path or algorithm. Some paths may lead nowhere, or to the wrong result, or tie us up forever repeating the same step(s); these paths are to be avoided. The development of a good algorithm takes clear thinking, as well as practice dealing with similar problems. It is a skill that will develop and improve the more you practice it. There are some useful tools that can help you design and improve your algorithms, tools such as flowcharts and other diagrammatic representations, or “pseudocode.”

A simple example of an algorithm is a recipe (an excursion into “culinary science”). Usually, a recipe consists of a simple sequence of steps leading to a desired result, the product to be eaten or drunk. A simple recipe for making cocoa is as follows:

COCOA

Ingredients (or “input”):

$\frac{1}{4}$ cup sugar
 $\frac{1}{4}$ cup cocoa
 $1\frac{1}{2}$ cups water
6 cups milk

Instructions:

1. Combine the sugar, cocoa, and water in a pan.
2. Heat mixture in pan over low heat until it boils.
3. Slowly add the milk to the pan; heat until it just comes to a boil.
4. Serve. (serves 6 in mugs)

The “output” is obvious—six delicious servings of hot cocoa. The set of instructions forms a simple *sequence*; it is easy to follow, and it leads to the correct result. You can let your imagination roam and imagine some “bad” solutions to this problem, perhaps by changing the ingredients, performing different steps, and so on.

An algorithm involves a *finite* set of instructions. The cocoa recipe is clearly such a finite set, and it would be impossible to provide you with an example of an infinite set of instructions. However, there is another interesting condition for a good algorithm—it must come to a conclusion in a finite amount of time. We can provide you with an example which looks like an algorithm, but does not satisfy this condition. Examine the following set of instructions given to a prisoner working on the road gang for purposes of “rehabilitation”:

ROAD GANG

Ingredients (“input”): shovel, a stretch of ground

Instructions:

1. Dig a hole in the ground one foot deep with the shovel.

2. Fill the hole back up.
3. Repeat.

This is not a good algorithm, because it includes the insidious instruction *repeat*, which says to go back to the beginning and do it all over again, and there is no way out! It is an *infinite loop* and is undesirable, in life or in a computer program. In fact, you will have to be on the lookout to avoid writing infinite loops in your programs and, even then, you will surely write many of them in your programming career.

Yet we have added an important dimension to our algorithms, the ability to *repeat*; the problem is that we repeat forever. If we have no ability to repeat, then all we can do is write simple sequences of instructions like the cocoa recipe, which only do something once, and severely limit our capabilities. Lives (and problems) are much more complicated than that. What we would like is the ability to repeat, but a definite number of times, or until some specific condition is met. Thus examine the following two algorithms: the first, to write out the numbers from 1 to 100; the second, to put money into your bank account until you have saved \$500 (to buy a stereo).

**WRITE OUT NUMBERS
FROM 1 TO 100**

1. Begin with the number 1
2. Print the number
3. Add 1 to the number
4. If new number is less than or equal to 100, then repeat from 2
5. Stop

**SAVE MONEY FOR
STEREO**

1. Start with balance of 0
2. Put money in bank
3. Add money to balance
4. If balance is less than 500, repeat from 2
5. Buy stereo

Both of these examples involve finite repetitions (unless, of course, you find it impossible to save money in the second example), and create useful algorithms. The first example, to write out the numbers from 1 to 100, is an example of a loop executed a definite number of times, that is, 100 times. The second example is one that is repeated until a certain condition (that the balance be at least 500 dollars) is met, but we do not know ahead of time how many times the loop must be repeated to satisfy the completion condition.

Both of these repetitive algorithms can be implemented if we have the ability to *test* a condition, whether the number is less than or equal to 100, or whether the balance is less than 500. Thus if we can add the *repetition* and the *selection*, or *test*, to our basic *sequential* operation of instructions, we will be able

to build powerful algorithms. We will then need to find FORTRAN statements that will allow us to implement these steps.

Before we move on to the building blocks and control operations available in FORTRAN, it is important to be sure that the algorithms we build using these tools will be effective. Thus it is good to look at the overall plan, and the details of the plan, before we begin building, just as in building a house. In building a house, we would use a blueprint to direct our efforts, and have a set of building materials available. In building and executing our algorithms, we need to have a perspective on the “blueprint” to be sure it will work, and to make sure we have all of the materials (or the “input”) available before we begin implementation. There are several useful tools and techniques that are helpful in implementing and evaluating our algorithms, and we will discuss them next.

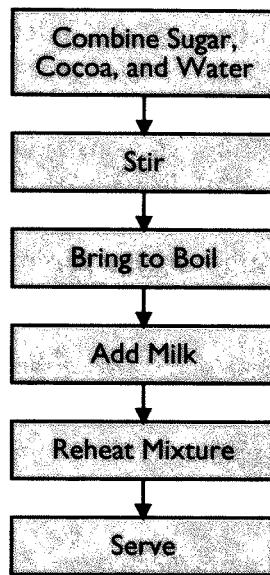
Flowcharts

To organize your thinking in a logical manner, a “picture” or diagram is often helpful. We use a blueprint in building a house, instead of trying to do it all from some mental picture, or instinct, or merely pages of instructions. Similarly, it helps us envision the logical “flow” of an algorithm we are developing by drawing a diagram of the steps and their connections in the algorithm. One such diagramming technique is called a “flowchart,” and has been around just about as long as computers have been available to solve problems. It has the advantage of being familiar to many people, because of its longevity. We will present this technique, and then discuss several alternatives, so that you can choose the one that best suits you for your particular program design activities.

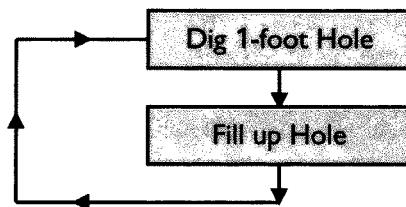
We have already seen from the simple algorithms we examined that we need to be able to represent simple operations, the sequence in which operations are to be performed, and the ability to repeat and test. We will begin with the simplest components for our flowchart, those of simple operations and sequential flow of such operations.



We can put any simple operation in a rectangular box, and we can use the directional flow lines to indicate the order in which such operations are executed. This will allow us to represent the cocoa recipe and other simple sequences of instructions as a group of connected operation boxes:

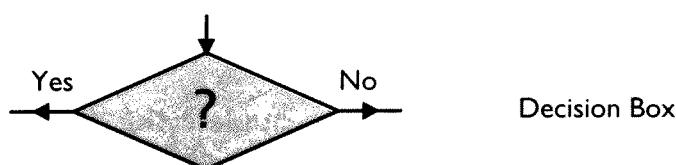


We can even draw a flowchart diagram for our infinite loop, the prisoner on the road gang:



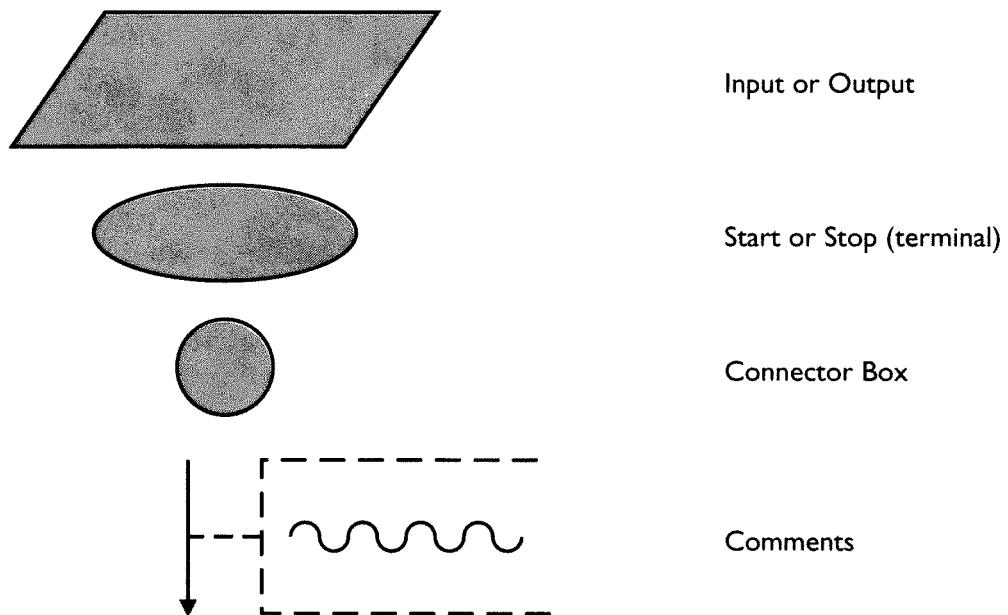
Notice that the “repeat” instruction has been implemented as a flow line that takes you back to the beginning of the sequence of instructions. In this way, we can draw an infinite loop, but we need more tools to allow us to create finite repetitive loops. We need the ability to *test* a value, and then perform an operation based on the results of such a test.

A *selection*, or *test*, operation in a flowchart is represented by a diamond figure. A flow line leads into the diamond, and contained in the figure itself is a question representing the test, shown by a question mark (?). Usually it is a “yes-or-no” type of question, and different paths out of the figure are designated depending upon the answer to the question. Such a flowchart figure is called a *decision box*:



The ability to test (to ask a question) and select the flow of logic depending on the result of the test (that is, make a *decision*), is perhaps the single most important and powerful capability we can have in our design language. This capability is exhibited by a number of constructs available in FORTRAN.

There are several other useful flowchart figures that will help design algorithms and represent them in pictorial fashion. These symbols are indicated here, and allow us to bring information in and out, indicate the beginning and end of the flowchart, continue a flowchart, and make comments:



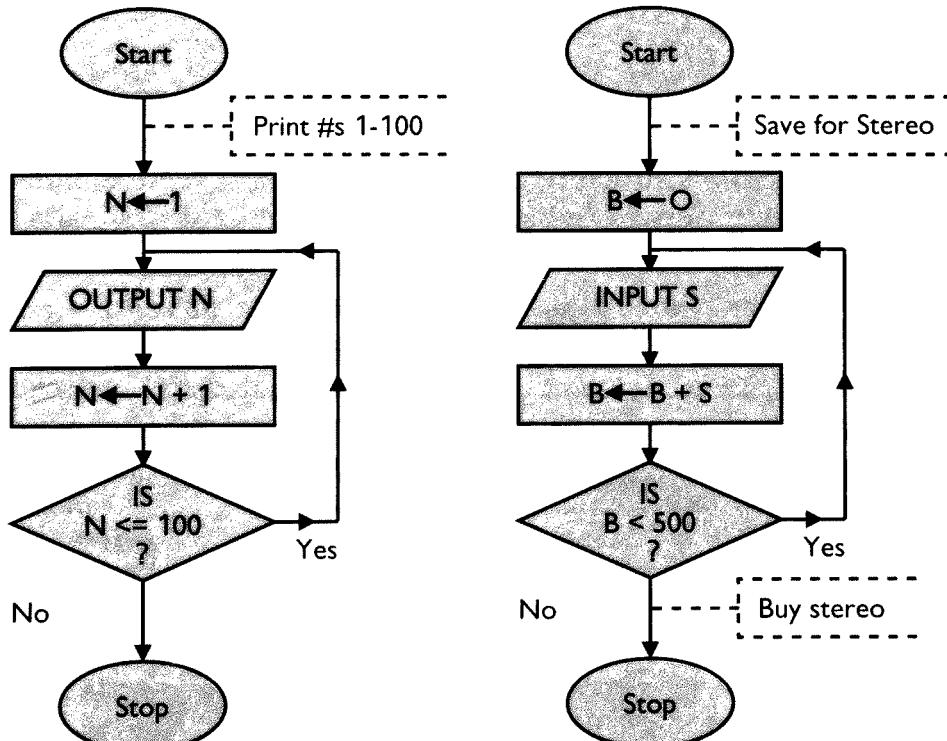
- The parallelogram representing either input or output must thus contain the appropriate word, INPUT or OUTPUT, to designate its operation.
- The terminal symbol (an ellipse), should contain the word START or BEGIN for the beginning of the program logic, or STOP or END for its termination.
- The round connector box is used where flow lines come together, or occasionally when you run out of space on one page and need to continue your flowchart on another page; in the latter case, a number (such as 1) is put in the connector box at the bottom of a page, and another connector box containing the same number is put at the top of the next page, so that if the two connector boxes were overlapped and stapled together, you would get a long, continuous flowchart.
- The comment symbol allows you to annotate the diagram with useful remarks or descriptions that do not affect the action.

Given this set of flowchart symbols, we can now draw diagrams of the number printing algorithm and the one to save money to buy a stereo. At the moment, we are working with the simplest set of building blocks necessary to write programs.

We need to add one more notation to our repertoire, however. When we have a value that changes during the progress of our algorithm, such as the number to be output or the balance being accumulated, we give it a name, refer to it as a *variable* (since its value will *vary*), and whenever it gets a new value, we will indicate the assignment of that value by an arrow pointing toward the name of the variable where it is stored: $A \leftarrow 6$ indicates the giving of a value of 6 to the variable named A. You might want to think of it as a special piece of paper on which you can only write one value at a time. When it is given a new value, the old value is erased.

One comment is appropriate at this juncture. We are currently using our "procedure boxes," the rectangles, to contain simple atomic instructions such as: "give a value to a variable," or "add numbers together," or "dig a hole." At the beginning we are writing simple programs made up of simple operations. For a more complex problem later on, a procedure box could be used to represent an entire detailed procedure, which we would have to develop as part of the "stepwise refinement" process.

Notice that *repetition* is effected in the flowchart by a *branch* to some earlier statement, indicating to repeat from that point. Such a branch can be accomplished in FORTRAN by a GO TO statement (to be discussed) or by

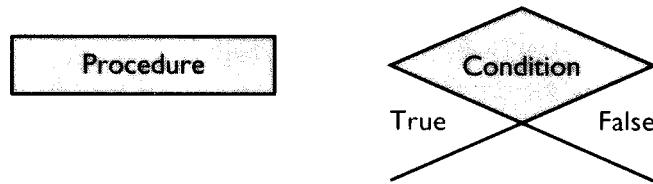


certain IF statements, but you will see that the language has incorporated more elegant ways to express selection and repetition. These newer constructs allow for a more *structured* (that is, neatly and logically organized) use of the language, so that the formerly essential GO TO can largely be eliminated, thus avoiding programs that seem to jump all over the place. Appropriate flowchart symbols for these constructs will be introduced as they are discussed in the text.

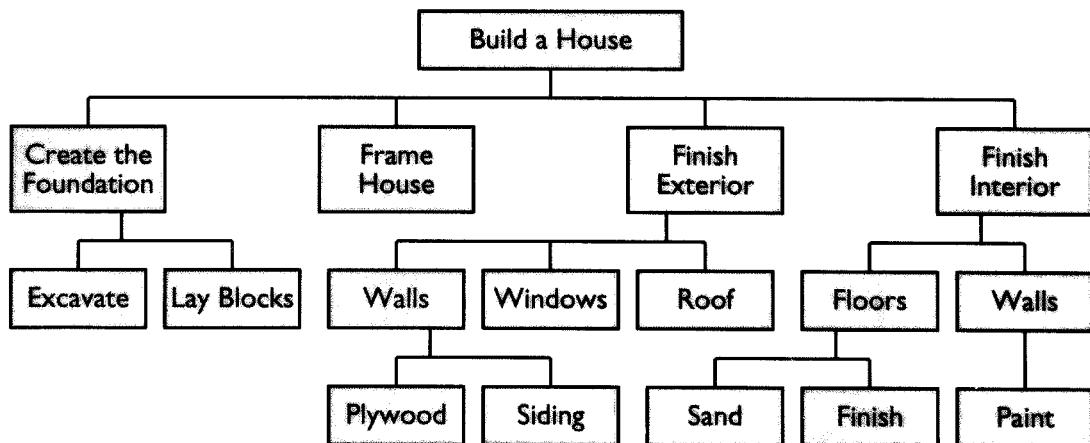
Other Program Design Tools

Structure Charts

Structure charts or *hierarchy charts* (known as HIPO diagrams—for Hierarchy of Input, Processing, and Output) can be used in organizing a problem solution. Such charts put simpler procedures in rectangular boxes, choice alternatives in two boxes following a diamond containing the test condition, and repetitions as a version of a choice condition (sometimes simplified to indicate just the condition followed by a box containing the operation to be followed as long as the condition holds true). We summarize these general forms here:

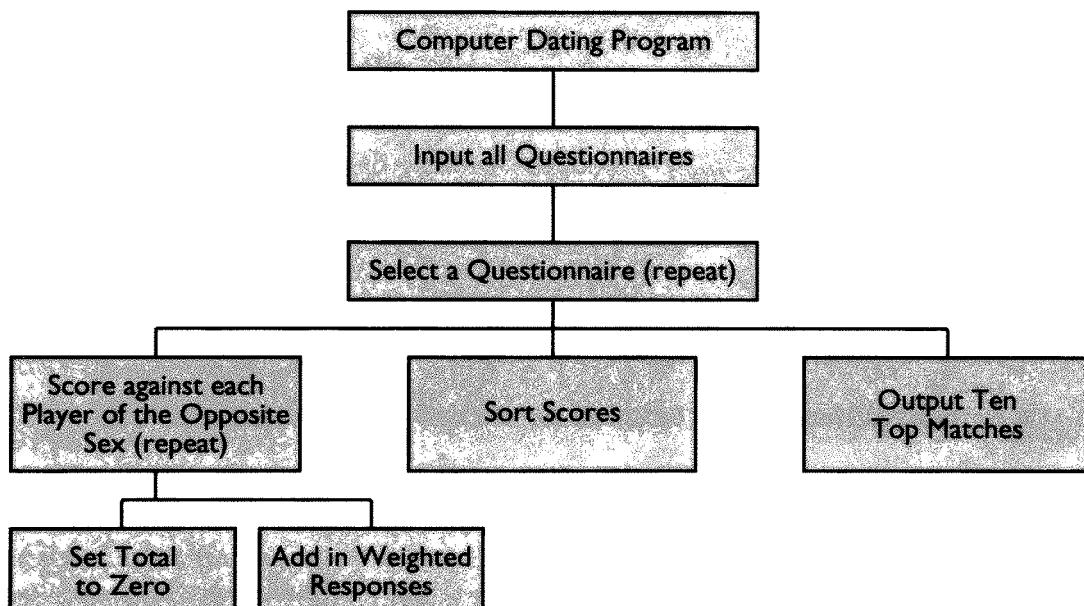


Generally, however, structure charts are used for an overview of the problem structure indicating how the main problem is broken down into smaller subproblems. A structure chart is not a bad first representation of a complex problem, to serve as a block diagram of the structural parts deter-



mined in a top-down design. If several procedures appear on the same line in a structure chart, they are executed from left to right. As a simple example, here is a structure chart for building a house.

As a more complex example, we could use a structure chart to represent the overall design of a computer-dating program, in which we would input questionnaires answered by a number of males and females, compare the questionnaire of each “player” against that of all of those players of the opposite sex and determine a *score* for each match, then sort the scores into descending order and output the names of the top ten best matches, according to their scores. An overview of this program could be given by a structure chart.



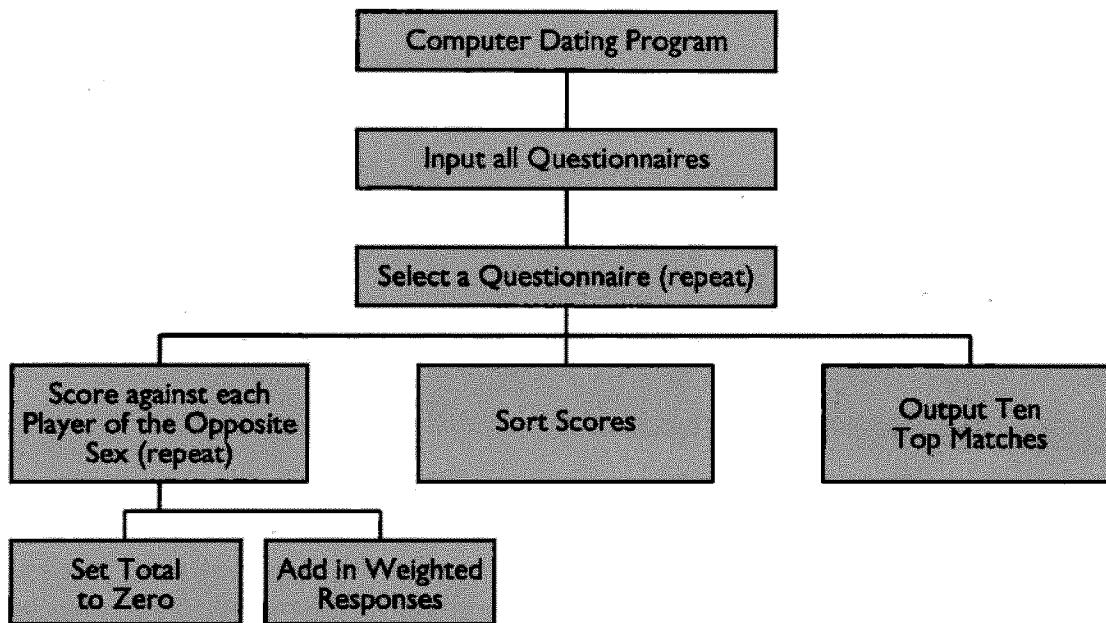
The details of implementation of such a problem involve certain repetition and decision structures that are more clearly represented by flowcharts or other methods. Thus the structure chart is valuable as a “modular” perspective on the organization of the overall problem, but other methods serve better for outlining the details.

Nassi-Schneiderman Diagrams

Another method of picturing algorithms that has been adopted is the Nassi-Schneiderman (N-S) diagram. These diagrams are all rectangular boxes, whether they represent sequential procedures, selection (test), or repetition. The advantage of such structures is that they can be easily *nested*, that is, drawn inside one another, to represent procedures within procedures, or repetitions within larger repetitions. The general formats of these diagrams are as follows:

mined in a top-down design. If several procedures appear on the same line in a structure chart, they are executed from left to right. As a simple example, here is a structure chart for building a house.

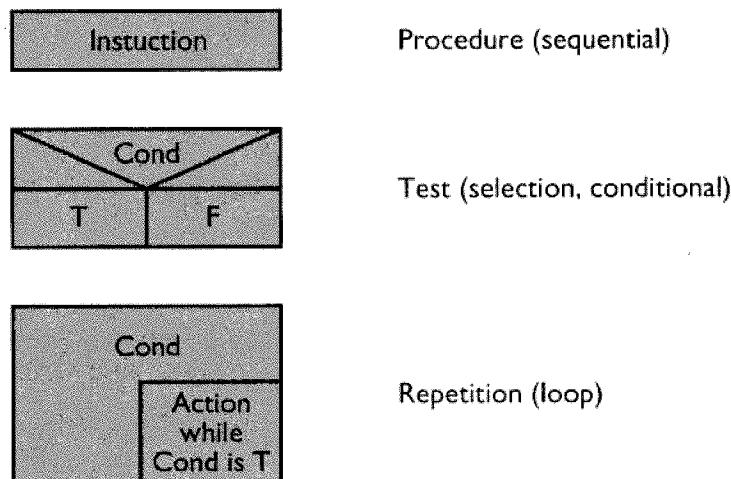
As a more complex example, we could use a structure chart to represent the overall design of a computer-dating program, in which we would input questionnaires answered by a number of males and females, compare the questionnaire of each "player" against that of all of those players of the opposite sex and determine a *score* for each match, then sort the scores into descending order and output the names of the top ten best matches, according to their scores. An overview of this program could be given by a structure chart.



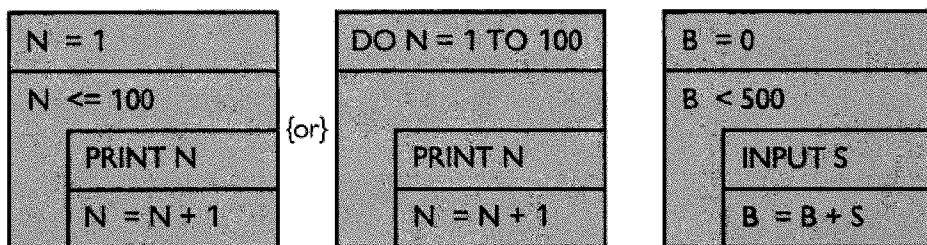
The details of implementation of such a problem involve certain repetition and decision structures that are more clearly represented by flowcharts or other methods. Thus the structure chart is valuable as a "modular" perspective on the organization of the overall problem, but other methods serve better for outlining the details.

Nassi-Schneiderman Diagrams

Another method of picturing algorithms that has been adopted is the Nassi-Schneiderman (N-S) diagram. These diagrams are all rectangular boxes, whether they represent sequential procedures, selection (test), or repetition. The advantage of such structures is that they can be easily *nested*, that is, drawn inside one another, to represent procedures within procedures, or repetitions within larger repetitions. The general formats of these diagrams are as follows:



Different techniques may be more appropriate for different problems. We can look at N-S representations for our earlier flowchart problems:



Pseudocode

Another approach to systematic program development that is used is *pseudocode*, putting instructions and control statements in an English-like formalized language. Because there is no standard form for pseudocode, it becomes simply an abbreviated version of the natural language description of the problem steps, in a form that parallels constructs available in the programming language. It is generally a mixture of English, mathematical expressions, and statements very similar to programming-language instructions. Simple instructions or procedures are described in abbreviated form. Selections, or tests, are generally expressed in an "if/then/else" form, and repetition is generally expressed in a "repeat action until," "do action for," or "while condition do" form. While some programmers consider this a valuable approach to describing the problem steps, others believe that it does not really add anything not already in the full English description, and that some pseduocode structures are so close to the actual implementation in a language such as FORTRAN or Pascal that there really is no point in making this pseudocode description an intermediate step. Let us examine some pseudocode representations of our two looping problems:

Initialize N to 1	Initialize B to 0
While N <= 100 do	While B <= 500 do
Output N	Input S
Increment N by 1	B = B + S

We will occasionally outline a problem solution in a brief form that might be considered pseudocode, but generally we will try to have a clear English description of the algorithm and then move directly to its implementation in a diagram such as a flowchart, or to its representation in FORTRAN.



LOOK FOR ANALOGIES

Finding good algorithms and actually carrying them out, in the physical world or in a computer program, can be a difficult task. It seems that we get better at doing such things the more we do them: "practice makes better." This is not some mysterious ability that suddenly is bestowed; clearly, there is something you absorb from the earlier successes that helps you in new problems. You develop a few tools that continue to be useful in many of the problem applications you face; you also develop certain techniques, or approaches, to solving problems.

You discover that the top-down approach works well for complex problems that initially look unmanageable, because it is not difficult to solve a number of small problems into which you have divided the big problem. You learn that certain approaches work best when you are faced with testing a number of alternatives, and you pick up many other "tricks of the trade" as you go along. But even more than that, you find that having solved a number of problems helps you solve different problems. You find similarities in the new problem to ones you have already solved. By finding such *analogies*, you are able to see through the difficulties of a new, seemingly foreign, problem. The ability to see such similarities, to recognize certain patterns, is one you will gradually develop over time, and that will prove to be very helpful to you.



PROGRAM TESTING

Once you have designed your algorithm and perhaps laid it out in a diagrammatic form, you will want to actually implement it. You can do this either by carrying out the instructions yourself (or giving them to someone else), or by writing a program to instruct the computer to do the work. In any case, you

will want to test your results to see that they are correct. One way to do this is to choose some simple values to run through your instructions or program, values for which it is easy for you to determine what the correct answer will be. You then compare the correct answer to that given by your set of instructions or program. If the answers differ, you have more work to do. By this approach, you can determine if your procedure or program is incorrect, but it does not demonstrate conclusively that the program is flawless, since that would require exhaustive testing.

There is an evolving science of *proving* programs correct, by treating them as a set of logical statements whose connections and conclusion can be logically demonstrated. However, this subject is beyond the scope of an introductory text. For the interested reader, references are given at the end of the chapter.



APPLYING PROBLEM-SOLVING TECHNIQUES

Now that we have discussed several approaches to organizing problem solutions, let us take a particular (not too difficult) problem, and run it through the various stages we have suggested.

Problem Specification

Your boss (or your instructor) presents you with the following task: given a particular length of (nonstretchable) rope, set up a table indicating the maximum area that can be enclosed if that rope is used to form various geometric figures. Your boss wants you to examine the areas if the rope is configured to enclose a square, a pentagon, a hexagon, a septagon (seven sides), an octagon, a decagon (ten sides), and a circle.

Clarifying Problem Specification

The purpose of this exercise is not disclosed to you when you are given the task description. You suspect that it may be to determine the optimum figure that can be made with the rope, so as to enclose the largest possible area. Being bright, you realize that if this is the intent of the exercise, the work involved in such a project can be skipped, since nature shows us that the circle is the optimum figure (or the sphere the optimum three-dimensional surface). You approach your boss with this “solution,” but you are told that this is not the point of the exercise. Instead, a company you are under contract to wants a table of the possible areas of the different figures, given a certain length of rope.

So you go "back to the drawing board." You then realize that you do not know how long the rope is supposed to be, and your boss tells you that this information is not currently available, and that the program should be written so that it will do the calculations for *any* length of rope that is specified. Thus, you will allow this value to be "input," and you will store it in a variable.

Next you need to decide what the output of your work will look like. Since the specs referred to a table, you will design your output in tabular form. However, just a list of numbers you have calculated will be meaningless to anyone but you, so the output should be properly labelled. You determine that the output will look something like the following (where the blanks _____ indicate places the program will fill in values):

AREAS ENCLOSED BY LENGTH OF MATERIAL _____ INCHES	
SQUARE	SQUARE INCHES
PENTAGON	" "
HEXAGON	" "
SEPTAGON	" "
OCTAGON	" "
DECAGON	" "
CIRCLE	" "

You then discuss this output "template" with your boss, to be sure it is acceptable. For example, will the rope length actually be specified in inches? If not, you will have to adjust. Is it acceptable that you used dittos after the first SQUARE INCHES entry, or should you spell out the label each time? (It is decided not to use the dittos.) You will also have to check out how much precision is wanted in the output—that is, to how many significant digits should it be specified, since that will affect the spacing in your table.

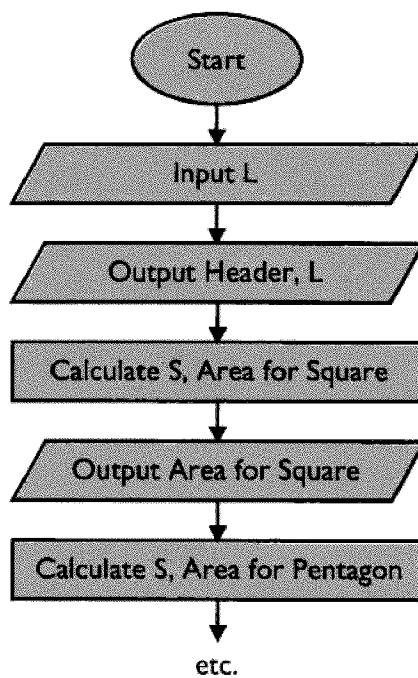
Developing the Algorithm

Once you are clear on the problem specification, you need to get down to the details of how you will do the calculations (whether you intend to do them by hand or with a computer program). The great advantage of writing a computer program for this is that, if the customer wants to "run" tables for very many rope lengths, all you will have to do is input the length to the program, and

the computer will do the work. If you have merely worked out an algorithm to do the calculations by hand, *you* will have to do the computations and write out the results for each different length specified.

You need to do some research to figure out how you will determine these various areas. You know that if you use the rope to form a square, each side of the square will be $L/4$ (if L is the rope length) inches long. Similarly, the pentagon will have sides of $L/5$, the hexagon of $L/6$, and so on. Of course, the circle is a different matter. L will be the *circumference* of the circle, and you know that the circumference of a circle of diameter D is πD , so D is L/π . Now you must rely on your memory, or (perhaps better) refer to a math text that will tell you how to calculate areas. You know that the area of a square of side S is S^2 ; but what about the areas of the pentagon, the hexagon, and so on? Best look them up and incorporate the formulas into your program design. You recall that the area of a circle of radius R is πR^2 , so you figure out how to calculate the area if you know the diameter (which is twice the radius).

All of this information constitutes your simple "algorithm," which will merely direct how the calculations are to be done. You hardly even need a flowchart or other diagram for this, but it might begin something like this:



Testing

Once the program (or set of instructions) is finished, you should *test* it on some simple length for which you can easily do the hand calculations, to be sure your results are correct.

Refinement and Maintenance

After you have run the program and verified the results, you may want to look for ways in which it might be improved. For example, might the customer wish to run through several different lengths at one time, instead of just one? If so, you would have to modify your work to allow for repetitions.

The Problem Solved in a FORTRAN Program

Even though you have not yet been introduced to all of the details of FORTRAN syntax, it may be informative here to include a FORTRAN program which will solve the “rope” problem we just described. This will at least show you how the problem-solving techniques can be translated into FORTRAN, and that a FORTRAN program is not all that formidable, but actually reads easily.

The problem specification has already been discussed, and its details clarified with the project manager. The “template” we designed is acceptable for the output, and we know that the program is to work for any (positive, nonzero) length of rope that is specified (as input). When we get into actually writing the FORTRAN program, the question arises, will the length of the rope be specified as an integer or a real value; we decide it will be a real, since few ropes are actually cut into exact integer lengths. In order to make the output look as described in the template, we determine we will have to use formatted output (discussed in Chapter 6), since this will allow us to control the spacing, so that everything lines up nicely, and also to control the number of places which are output to the right of the decimal place for rope length and areas (we decide on two places).

In developing the algorithm, we find a formula for the area of a polygon (in the *Handbook of Chemistry and Physics*) which says that the area of a regular polygon of n sides each of length l is:

$$\frac{1}{4} nl^2 \cot \frac{180^\circ}{n}$$

We discover that FORTRAN has a built-in function TAN which will calculate a tangent of an angle (expressed in radians); this function is discussed in Chapter 9 and appears in Appendix B). We recall that the cotangent of an angle is equal to 1 over the tangent (i.e., $\cot x = 1/\tan x$), so we will be able to use this to calculate cotangents. We know that there are 2π radians in 360° , so the 180° in the numerator of the angle can be expressed as π radians. Since the formula will work in general for any number of sides n and length of sides l , we write our own statement function (Chapter 9 again) that will calculate the area given any n and l , so we do not have to write the formula out in all its detail every time. We are ready to begin our program.

```
***** AREAS CIRCUMSCRIBED BY A ROPE OF LENGTH L *****
REAL L, LEN, AREA, PI, R, SQUARE, CIRCLE
PARAMETER (PI = 3.14159)
AREA (N, L) = 0.25*N*L**2 * 1.0/( TAN(PI/N) )
*** INPUT ROPE LENGTH AS A REAL, WITH A DECIMAL POINT ***
READ*, LEN
PRINT 77, LEN
77 FORMAT('1',14X, 'AREAS ENCLOSED BY LENGTH OF MATERIAL',
& F10.2, ' INCHES')
*** BEGIN BY CALCULATING THE AREA OF A SQUARE MADE BY THE ROPE ***
L = LEN/4.0
SQUARE = L*L
PRINT 99, 'SQUARE ',SQUARE
99 FORMAT('0', 20X, A8, 10X, F12.2, ' SQUARE INCHES')
L = LEN/5.0
PRINT 99, 'PENTAGON', AREA (5, L)
L = LEN/6.0
PRINT 99, 'HEXAGON ', AREA (6, L)
PRINT 99, 'SEPTAGON', AREA (7, LEN/7.0 )
PRINT 99, 'OCTAGON ', AREA (8, LEN/8.0 )
PRINT 99, 'DECAGON ', AREA (10, LEN/10.0 )
*** NOW DO THE CIRCLE, FIRST BY CALCULATING THE RADIUS R ***
R = LEN/(2*PI)
CIRCLE = PI*R*R
PRINT 99, 'CIRCLE ', CIRCLE
STOP
END
```

The output from this program, for an input length of 100.0 inches, looks as follows:

```
AREAS ENCLOSED BY LENGTH OF MATERIAL 100.00 INCHES

SQUARE      625.00 SQUARE INCHES
PENTAGON    688.19 SQUARE INCHES
HEXAGON     721.69 SQUARE INCHES
SEPTAGON    741.62 SQUARE INCHES
OCTAGON     754.44 SQUARE INCHES
DECAGON     769.42 SQUARE INCHES
CIRCLE      795.78 SQUARE INCHES
```

The program was tested on several rope lengths, and the results for 100.0 inches were "hand-checked" (that is, the calculations were done by hand to check the computer's results).



SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

This chapter has been devoted primarily to approaches to solving problems. These are general suggestions, for all sorts of problems, not just those on a computer. You must first *define the problem* clearly, removing any vagueness or ambiguities. A problem generally involves getting from one situation (the initial state) to another (the goal state), and the solution describes the steps needed in between. To do this adequately, the conditions of both the initial state and the goal state must be clearly defined (that is, you must know where you are coming from and where you are trying to go).

You must *organize the approach systematically*, so that you are not just randomly striking out in all directions. Generally, any complex problem is too difficult to handle as a whole (that is, to see the solution all at once), and you need to break it up into smaller, manageable problems (this approach is called *top-down design*). You then solve each of the smaller problems (this approach is called *step-wise refinement*), and *synthesize* these solutions into one whole.

An *algorithm* is a sequence of instructions that comes to a result in a finite amount of time. Some examples of algorithms are recipes, instructions on how to build something, and any well-constructed computer program. In addition to *sequential operations*, an algorithm may include *repetition* (as long as it is limited, not infinite) and *selection* (that is, the ability to *test* conditions, and make decisions based on those tests).

A *flowchart* is an important problem design tool that shows—in a diagrammatic fashion—the flow of the action in the problem solution. It is basically composed of *procedures* connected together in sequence by “flow lines” (which can also indicate the order of repetitions), and *decision boxes* (points at which tests are made). It can also include *input*, *output*, and *comments*, as well as indicators for the beginning and end of the process. Other problem design tools discussed in the chapter were *structure charts*, *Nassi-Schneiderman diagrams*, and *pseudocode*.

Once you have begun solving problems, you will find that your previous experience is helpful. New problems often have similarities to problems you have already solved, and if you are on the alert for such *analogies*, it will make handling the new problems much easier. In this book, you will learn how to develop a set of reusable tools that will apply to many new problem situations.

A problem solution (or program) must be *tested* once it is developed. This involves applying it to many different sets of starting conditions, to be sure it can stand up to the variations (that is, whether it is *robust*), and you must *hand-check* the results to be sure they are correct. Just because a computer produces an answer does not automatically make that answer correct. Once a problem solution has been developed, whether it is a machine to perform a task or a computer program, it must be *Maintained*. That is, during the lifetime of the

tool, it must be kept free of errors, it must be repaired, and new features may have to be added to it to adapt to changing problem conditions.



EXERCISES

1. Briefly describe, in English or a pseudocode, the steps required to solve the following problems. Then draw a flowchart or an N-S diagram that outlines your solution.
 - a. Outline your morning routine (get up, breakfast, work or classes, etc.). You may assume that many common-sense operations are understood, but at least think about the details you would have to provide to a friendly alien to explain these actions.
 - b. Outline a plan for selecting your course schedule for next semester. Take into account that some courses may be closed. Note that this will surely involve some tests and repetitions.
 - c. Add the integers from 1 to 100, and output the sum.
 - d. Add the even integers from 200 through 400, and output the sum. Use the clear analogy with your solution to question c.
 - e. Calculate the gross pay for a worker who works any number of hours (input the number of hours), if the pay scale is \$4 an hour for regular time (up to 40 hours), \$6 an hour for overtime (hours between 40 and 60 hours), and \$8 an hour for double time (hours over 60). Output the gross pay amount. Think carefully!
 - f. Calculate the amount of tax a person must pay under a new tax law for any amount of taxable income (input INCOME), if the tax tables are as follows:

Up to \$15,000	No tax
\$15,000 to \$25,000	10%
\$25,000 to \$50,000	20%
Over \$50,000	30%

Notice the analogy of this problem to question e, except for the addition of a zero-tax bracket. Both involved graduated scales.

- g. If you get paid 0.5% on your initial investment of \$1000 every month, and you let the interest go in as additional investment (and earn interest on it), how many months will it take for your total investment to reach \$1500? This

problem involves a repetition structure (a loop) and a *counter* to keep track of the months involved. If you have difficulty with this, be patient. We will discuss accumulation and counters soon.

2. Try to think of as many everyday instances as you can in which making an analogy to some previously learned skill helped you with a new problem or ability. The more you practice the skill of finding similarities and patterns, the greater it will become.
- 3. Draw a flowchart to find the factorial ($n!$) of any positive integer n input, where the factorial is defined as the product of the integers from 1 through n . Output n , “ $! =$ ”, and the product.
4. Draw an N-S diagram to outline the procedure for finding and printing out the *largest* value in an input set of 100 integer values; think about an effective technique to accomplish this.
5. Draw a structure chart that represents the “chain of command,” or organizational structure, at your college or the company for which you work.
6. Draw a structure chart of the phylogenetic tree.



SUGGESTED READINGS

Anderson, Robert B. *Proving Programs Correct*. New York: John Wiley & Sons, 1979.

Arbib, Michael A., and Saul Alagic. *The Design of Well-Structured and Correct Programs*. New York: Springer-Verlag, 1978.

Deutsch, Michael S. *Software Verification and Validation*. Englewood Cliffs, N. J.: Prentice-Hall, 1978.

Dijkstra, E. W. *A Discipline of Programming*. Englewood Cliffs, N. J.: Prentice-Hall, 1976.

Mason, John, Leone Burton, and Kaye Stacey. *Thinking Mathematically*. Reading, Mass.: Addison-Wesley, 1982.

Mili, Ali. *An Introduction to Formal Program Verification*. New York: Van Nostrand Reinhold, 1985.

Nassi, I., and B. Schneiderman. “Flowchart Techniques for Structured Programming.” *SIGPLAN Notices* (August 1973), 12-26.

Polya, G. *How to Solve It*. New York: Oxford University Press, 1957.

Van Tassel, D. *Program Style, Design, Efficiency, Debugging, and Testing*. 2nd Edition. Englewood Cliffs, N.J.: Prentice-Hall, 1977.

Yeh, R. T. *Current Trends in Programming Methodology. Volume II: Program Testing and Validation*. Englewood Cliffs, N. J.: Prentice-Hall, 1977.

CHAPTER 2



FORTRAN BUILDING BLOCKS

You and the computer should become a problem-solving team. In order to communicate, however, you must learn the computer's language. Once the basic elements of FORTRAN—numbers, character strings, and "logical" values—have been mastered, we can move on to the FORTRAN alphabet, performing arithmetic operations, the use of variables, and how to input and output information. After learning FORTRAN's building blocks, you can begin to construct programs.

The complexity of a skyscraper under construction increasingly demands that architects and engineers use sophisticated computer programs to help them plan and execute their work.

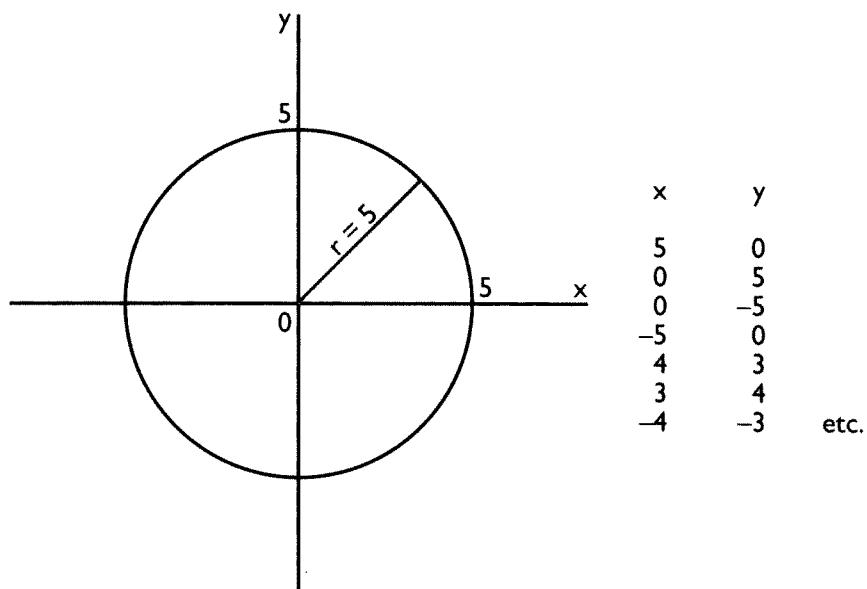
"Think of the tools in a tool-box: there is a hammer, pliers, a saw, a screw-driver, a rule, a glue-pot, glue, nails, and screws. The functions of words are as diverse as the functions of these objects."

- Ludwig Wittgenstein, Philosophical Investigations

In order to communicate with the computer—to get it to do the jobs you want and solve the problems you want to solve—you need to learn the various elements (constants and variables of various kinds) that the machine will operate on, and the instructions you can issue in FORTRAN to cause operations on these entities. You will discover that you can give the machine a variety of kinds of values to work on (within broad limits), and have it perform sequential operations, alter the sequence when appropriate, perform tests (selection), and execute repetitions. With these tools at your command, and a logical, systematic approach to analyzing problems, you will be ready to handle a wide variety of interesting problems.

◆ CONSTANTS AND VARIABLES

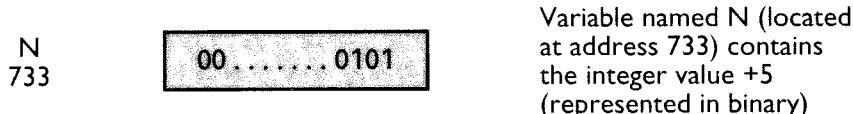
The computer can store different numeric and nonnumeric values, and do calculations and comparisons with them. A value may be used directly, or it may be stored in a place in the machine for later examination, manipulation, or alteration. Such a place for storing values (or instructions) is variously referred to as a *location*, a *word*, or a *cell*. When values are stored in such a location, the location acts as a *variable*, since the particular value in that



location can vary during the course of the program execution. A geometric analogy utilizing constants and variables may prove helpful at this point.

The circle illustrated has the equation: $x^2 + y^2 = 25$. That is, it is a circle centered on the origin, with a radius 5 units long. The circle itself is a continuous curve, but we can make a table of the coordinate values of some of the points that lie on the curve (though there is actually an infinite number of them). We may say that the circle's equation involves *constants* (actually one constant, the value 5 of the radius) and also *variables*. The variables are the coordinate placeholders x and y which, as we have seen from the table, can take on a range of values depending on which point on the circle we are describing.

We will make a similar distinction on the computer. It can accept and represent constant values, which may be integer, real, complex, characters, and so on. It can also store values in certain locations, and these locations will be distinguishable to the computer by their *addresses*, or numerically coded positions in memory, and to the programmer by their *names*. The contents of any such variable location may change during the program.



The compiler, which translates the FORTRAN program into machine language, will set up a *symbol table* of the relationships between variable names assigned in the program and the actual physical addresses given to these variables by the program, similar to:

N 733

Thus, every reference in the FORTRAN program to the variable named N will be translated into a reference to address 733.

Integers

The two common kinds of numeric values—integers and reals—that you are accustomed to can be represented in the computer. Integer values are whole numbers, with no fractional part; they may be positive or negative. No special nonnumeric characters may appear in an integer representation for the computer, including '\$', ',', '.', or '%'. The magnitude of an integer that can be stored on your computer depends on the word size of your machine. Most usually, if your machine has an n-bit word size, the magnitude of the largest integer that can be stored will be:

$$2^{n-1} - 1$$

We have already discussed the reasons for this. Here are some examples of legal and illegal integer values:

<u>Legal</u>	<u>Illegal</u>
365	45.8
-2	32%
1000000000	1,000,000,000

Reals (Floating-Point Values)

A true mathematical *real number*, which could have an infinite decimal expansion, cannot be represented on the computer since it only has a finite (and rather small) number of places to store significant digits. A *significant digit* is one that contributes to the overall magnitude of the value. For example, the value 3.14159 contains six significant digits, and 0.5 contains one significant digit. We also say that a value has a certain *precision*, which refers to the number of significant digits it can contain.

Numbers with fractional parts, if the values of the numbers are not too large or too small, can be represented internally on the machine to some definite limited precision. Of course, they will be converted to an equivalent binary representation before storage, and you may want to examine those representations at some time (in which case you should consult Appendix A). But, for the most part, the programmer can remain unaware of the actual internal representations and workings of the computer and, as long as you follow the rules, you can generally assume that the machine will be doing what you tell it, operating on the values you have given it.

The programmer should, however, be aware that the machine (binary) representations of fractions, with the exceptions of 0.5, 0.25, 0.625, and so on (combinations of negative powers of 2), cannot be represented exactly on the computer. Thus, for example, the binary representation for 0.4 stored on the machine is actually a value slightly less than the decimal 0.4. This can result in errors in computation, since five 0.4's on the machine will not quite add up to 2.0, giving results that are slightly off those expected.

We can express "real" values simply as a string of digits containing a decimal point (such as 3.25), or occasionally in an exponential, or "scientific," notation such as:

$$0.356 \times 10^7 \text{ or } 0.289 \times 10^{-4}$$

The latter values would be expressed when writing a FORTRAN program by letting the letter 'E' take the place of the "times ten to the power of" notation.

Thus they could be written:

0.356 E 7 0.289 E -4

In scientific notation, the number coming first, or the multiplier (or *mantissa*), is usually expressed as a real, with a decimal point (though it may be an integer), and the exponent following the E is always an integer value. This conforms to actual scientific notation, in which you never see a value expressed as a multiplier times 10 to a fractional power.

Simple real values are just expressed normally, containing a decimal point. Notice that a real value can be expressed a number of ways by utilizing scientific notation:

32456.7 or 3245.67 E 1 or 324.567 E 2 or 32.4567 E+3
or 3.24567 E 4 or 324567.0 E-1 or 3245670.0 E-2 etc.

Thus, the same value can be represented by changing the position of the decimal point (letting it "float") and adjusting the exponent appropriately to compensate. This is why real values are often referred to as "floating point" numbers.

Real values may legally be expressed in either normal or scientific notation, as long as no characters other than digits, ".", and "E" are used, and the values are not too big or too small for storage. To give you a sense for these limitations—which are not usually very severe given most of the problems you will deal with—we will look at the range of real values on a few representative computers. Recall from our earlier discussion that a real value will be stored as a signed multiplier (expressed in binary) and a signed exponent; the number of bits allocated to the exponent and to the multiplier will determine the magnitude of the number that can be stored and the number of significant digits of precision.

On a 32-bit IBM machine, a real can be stored to a precision of about six decimal digits, and have a magnitude on the order of 10^{75} . On a 32-bit VAX, however, the magnitude is about 10^{38} with about seven significant digits. The difference in magnitude for words of the same length occurs because the VAX stores and interprets the exponent as a power of 2, whereas the IBM stores and interprets the exponent as a power of 16. A 48-bit Unisys machine has a precision of about 11 decimal digits, and a range of magnitudes from 10^{-47} to 10^{68} . A 60-bit CDC machine can store values to a precision of 14 decimal digits, with magnitudes that fall in a range from 10^{-293} to 10^{322} . A Cray supercomputer, with a 64-bit word, can store values with a precision of about 13 decimal digits, and a magnitude of about 10^{2466} . An IBM microcomputer with a 16-bit word can store reals of a magnitude of about 10^{38} , and integer values from -32768 to +32767. Most serious scientific problems that do not merely deal with counting occurrences will need to make use of real numbers; for example, any

problem dealing with orbital calculations, or any force-field problems, will require reals.

You can thus see that there is a wide range of the values that can be stored on different computers. You should check the machine you are using to learn the numeric limitations it has.

Other Types of Constants

For the sake of completeness, we will briefly mention here the other types of constants available in a FORTRAN system, but we will wait until later to describe them in detail. Integers and reals will be sufficient to deal with most of the problems you will encounter.

Since we have already seen that there may be a tight limitation on the magnitude of real numbers that can be stored in a single memory word, FORTRAN provides the capability of storing a real in two consecutive locations in memory. This is called a *double precision* constant, and it extends the magnitude or the precision (or both) of a real that can be stored. Double precision constants may be represented on input or in the program in basic scientific notation, except that a "D" is used instead of the "E" preceding the exponent (e.g., 0.9976 D+150).

Complex constants can be represented in FORTRAN. These are numbers with a real and an imaginary part. They are stored in two consecutive locations in memory, the first containing the real part, the second the imaginary part. A complex constant is input, or represented in a FORTRAN program, as a pair of real numbers enclosed in parentheses and separated by a comma. The first value is the real part, the second the imaginary part. Thus the complex constant:

$$3.0 + 4.0 i$$

will be represented as: (3.0, 4.0) in FORTRAN.

Logical values, representing *true* and *false*, are stored in a single word in memory (actually needing only 1 bit). The Logical constants are designated .TRUE. (or .T.) and .FALSE. (or .F.) in a program or as input.

Character constants are simply strings of characters that may be entered, manipulated, or stored. In FORTRAN 77, a character string may be as long as you like, and it is enclosed in single quotes. Thus for example 'CAT' and '#\$%@#% %*%' are character constants which may be input, stored, or output. In Fortran 90, either single or double quotes may be used ("CAT").

We will discuss these additional constants and their uses as the occasion arises in later chapters.

Variables

We must now examine the computer locations used to save, examine, and alter the constants we have discussed so far. We have seen the relationship between a variable location and the value stored in it—somewhat like a box and its contents. These boxes have names and addresses, to distinguish them from one another. They may be initially “empty,” and have different contents stored in them at different times.

The computer has a numerical address to find each location in which a value has been stored, but the FORTRAN programmer does not usually have access to these addresses, and so must refer to variable locations by name. In FORTRAN 77 (and in all earlier versions of FORTRAN), these are the:

Rules for Naming Variables

1. Variable names may be one to six characters long.
2. Variable names may be composed of letters and/or digits, but must begin with a letter (A–Z).
3. By default, unless specified otherwise, all variables beginning with the letters I, J, K, L, M, or N (I–N) will be type INteger, that is, they will be used to contain only integer values. Variables beginning with the remaining letters (A–H, and O–Z) will by default, unless specified otherwise, be considered to be type real and contain real values.

Some processors will allow lower-case as well as upper-case alphabetic characters in names and programs; in such cases, FORTRAN considers the lower-case letter equivalent to its upper-case correspondent. Some compilers allow variable names to be longer than 6 characters, and the new Fortran 90 Standard will allow them to be up to 31 characters long and include the underline character (_), but the FORTRAN 77 Standard restricts names to 6 characters. If your compiler allows longer names, you may want to take advantage of this flexibility to give more meaningful names to your variables, but if you do, you must be aware that this will mean that your programs will not be portable under the Standard. Also consider that very long names such as AerodynamicThrust must be typed out in full every time, and are subject to typographical errors. A reasonable compromise should be reached; names should be long enough to be *meaningful*, so it is obvious what they represent in the program, but not so long as to be awkward.

Given the rules for naming variables, here is a list of examples of legal and illegal variable names under the Standard:

<u>LEGAL</u>	<u>ILLEGAL</u>
A	A\$ (no '\$')
M13	4F (can't begin with a digit)
CAT DOG	LONGNAME (more than 6 chars.)
C3P0	GET-UP (no '-')
R2D2	BY_BY (not in FORTRAN 77)
READ	{there are no <i>reserved words</i> that may not be used in FORTRAN, as there are in other languages in which the programmer must be on the alert not to use reserved words}

Variable Types

The rules for naming variables used the concept of a *data type*; in particular, they mentioned the integer and real types. In FORTRAN, a variable should be of the same type as the constant it contains. Thus, given the different kinds of constants available in FORTRAN, we must have the data types integer, real, double precision, complex, logical, and character, for containing these various constants. You can *declare* a variable to be of any of these types by the use of a *type statement*, which has the general form:

typename var1, var2, ...

One or more variables may be declared to be of a particular type in a single type statement, and this means that the constants stored in those variables should be of the declared type. Several examples of type statements follow:

```
INTEGER CAT, BAT, HORSE
REAL GOOD, DOG, MARK
LOGICAL X, Y, Z
CHARACTER ALPHA, BETA*3
INTEGER MAD, MAX
DOUBLE PRECISION PI, EPSILN
COMPLEX A, B, C
```

Notice that a particular kind of type statement designator can appear more than once (as INTEGER does in the example). Type statements may be more complex in Fortran 90; see Chapter 5.

Type statements must appear at the beginning of the FORTRAN program, with all other "declaration statements," and before any *executable* FORTRAN statements (those that are commands). If you do not declare the type of a variable in such a type statement, it will take on the *default* type indicated by the first letter of the variable name.

Default typing was built into FORTRAN from the beginning, with the

thought that most variables used would be either integer or real and that following the default convention would save the programmer the trouble of declaring the type of every variable used in the program. Thus, many FORTRAN programs are written making use of this convention, and many FORTRAN programmers will automatically expect any variable beginning with I–N to be INteger (a quick mnemonic memory trick), and any others real. The original designers of FORTRAN selected variables beginning with I, J, and K to be integer, since these are generally the letters used in mathematical notation for integer subscripts and the like. Then, according to Backus, they just added a few more letters to the list. If you deal with any older established programs, they may well rely on this variable-naming convention and have no explicit typing.

With the development of structured programming approaches over the past several years, and the influence of “strongly typed” languages (in which all variable types *must* be declared), there has been a marked trend toward declaring the types of all variables used at the beginning of a program. One advantage of such implementation in a strongly-typed language like Pascal, for example, is that if the compiler then encounters a variable name that has not appeared in a type statement, it will flag the presence of an error and not compile the program. This is a trap for misspelled variable names, or for variables that are used without being declared. However, this advantage would not work in current FORTRAN compilers, since an untyped variable will simply be given its default type, whether that is what the programmer would want done or not.

It is good programming practice to adopt the stylistic convention of declaring the types of all variables used in your programs at the beginning of the program. You should not, however, mistakenly assume that it will then automatically flag any spelling errors or undeclared variables, as it would in a strongly-typed language such as Pascal, since it will not do so.¹

There is thus a distinction built into FORTRAN between *implicit* typing of variables, that is, the default conventions discussed in the naming rules, and *explicit* typing, in which you specifically list all of the variables you will use in your program and classify each of these variables by its type. Veteran FORTRAN programmers will probably tend to adopt the implicit typing conventions, using type statements only when the meaningful name of an integer or real variable that they want to use conflicts with the convention or when variables of types other than integer or real are used. Newer programmers, especially those who have been touched by the influence of more recent strongly-typed languages such as Pascal, will declare all variables at the beginning of a program. You may also declare a range of variables beginning

¹You may look forward to the new Fortran 90 version, however, where by including a statement of the form IMPLICIT NONE at the beginning of your program, the Fortran 90 compiler will then *expect* all variables to be declared explicitly.

with certain letters to be of a specific type, by using the IMPLICIT type statement which will be discussed in Chapter 5.

It is good programming practice to explicitly type all variables at the beginning of a program. This kind of organizational, housekeeping chore makes the programmer pay close attention to the building blocks of the program and makes error correction and program modification easier. It is also a habit that will transfer readily to the use of many other languages, whereas FORTRAN's implicit typing is unique.

Since the I-N default convention for integer variables in FORTRAN had its origin in mathematical practice, it is probably not necessary to declare a simple integer variable such as I, J, K, L, M, or N as integer, when it is used for a loop index or as a subscript in an array. Just be aware of the considerations of good programming practice and of the current limitations of type checking in FORTRAN. Program examples in this text will generally make use of the implicit typing convention for simple integer variables, but will explicitly declare the rest.

Notice that any variables of type DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER must *always* be declared in FORTRAN 77.



ASSIGNMENT STATEMENTS

Now that you are relatively comfortable with the nature of constants and variables in FORTRAN, it is time to see how variables can take on values. There are basically three ways in which a variable gets a value: through input, through a DATA statement, and through an *assignment statement*. We will defer discussion of input and DATA statements to a slightly later time, and concentrate now on the use of the assignment statement, which is the most common way for a variable to acquire a value.

We have looked at a sort of assignment instruction already, in our construction of flowcharts. In our flowchart procedure boxes, if we wanted a named entity to take on a value, we used an arrow pointing from the value to the name, as $A \leftarrow 6$. The same sort of operation can be performed in FORTRAN, except that the syntax is of the form:

variablename = expression

The analogous statement in FORTRAN to our flowchart example would be $A = 6$. The “=” in the assignment statement is not the algebraic “equals” you are used to; instead, it has the same sense as the flowchart arrow (\leftarrow), “is assigned the value of.” Thus, you may write in FORTRAN an assignment statement such as:

$N = N + 1$

which would never be a legitimate algebraic equation; but in FORTRAN it has the result of taking the current value stored in location N (suppose that it is 4), increasing that value by 1, and storing the new result (in our case, 5) back into location N. In algebra, you have the commutative law: $a + b = b + a$, but

$$A + B = B + A \quad [\text{Illegal in FORTRAN!}]$$

would be flagged by the FORTRAN compiler as an error, since the entity on the left of the “=” must be a single variable.

The expression on the right of the “=” in an assignment statement should be of the appropriate type, that is, a character expression if assigned to a CHARACTER variable, a logical expression if assigned to a LOGICAL variable, and so forth. However, real and integer variables to which values are assigned are exceptions to this, since FORTRAN will perform the conversion to the appropriate type of numeric value upon storage. A *numeric* expression assigned to a real or integer variable may be a constant, a variable name, or any arithmetic expression in terms of constants and/or variables. Next we need to examine the form that arithmetic expressions may take in FORTRAN.



ARITHMETIC EXPRESSIONS, ORDER OF OPERATIONS

FORTRAN can perform all of the normal arithmetic operations: addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**). Further, there are many built-in FORTRAN system functions available to perform numeric operations, such as SQRT, SIN, COS, TAN, ABS, and so on (see Appendix B). An arithmetic expression can combine any selection of constants and variables using any of the arithmetic operators (or the system functions), and may be as long as you like. Very long assignments may be continued onto more than one line of the program, but these generally become very difficult to follow (for the human programmer, not for the computer!).

Thus, if we assume that real variables A, B, C, and D, and integer variables M and N have already been assigned values, then the following are legal FORTRAN assignment statements:

```
N = N + 1
M = N**2
X = B - C*4.2
Y = (B + C)/A*D
X = Y - A*B/C*D
```

When we have a complex arithmetic expression such as some of those in the examples, the question arises regarding the order in which the operations will be executed. For example, if the expression is $3 + 4*2$, which operation is performed first—the addition or the multiplication—since the answer will be different depending on which operation is done first. Your mathematical training tells you that the multiplication in this example would be done first, and you would be right, since FORTRAN tries as much as possible to follow mathematical and scientific conventions. But it is always important to check the rules in a programming language, since they may not always dictate what your experience expects.

In evaluating an arithmetic expression in FORTRAN, there can be no ambiguities, so that if your expression does not explicitly force the order in which operations are performed, they will be executed according to the hierarchy: any expressions in parentheses are executed first, then any function references, then any exponentiations, then multiplications and divisions, and finally any additions and subtractions. If required, then an assignment is performed.

Order of Precedence of Arithmetic Operations

()
function references
**
* or /
+ or -

This hierarchy table solves most of our problems, and tells us how the following expressions would be evaluated (the labelled boxes at the right indicate the contents of the storage locations):

M = 15 - 2*3 {as $15 - (2*3)$ }	
N = 2*4 - 2**2 {as $(2*4) - ((2**2))$ }	
K = 2*3**2 {as $2*(3**2)$ }	
KAT = (2+6)/(6-2)	

M	9
N	4
K	18
KAT	2

Notice that you can *force* certain operations to be performed first by enclosing them in parentheses. In our last example, the result would have been very different if we had written instead:

KAT = 2+6/6-2

KAT 1

However, the hierarchy table does not answer all of our questions about evaluation of expressions. For example, according to the precedence of operators, how would the following example be evaluated?

$$\text{MACK} = 3 * 4 / 2 * 2$$

Is the answer stored in MACK a 3, or is it a 12? The hierarchy table doesn't help us answer this, since it says that multiplications and divisions are on the same level regarding precedence. So what is the answer? FORTRAN will not allow such an expression to be ambiguous, or to have one answer on Mondays, Wednesdays, and Fridays, and the other answer on the remaining days of the week.

The FORTRAN compiler interprets such an expression, after all of the precedence rules have been applied, *from left to right* (with one exception—in successive exponentiations, they are evaluated right to left). And you say, "Of course! I knew that!", since that is what your mathematical training would tell you. However, you should be careful, and always be sure what the rules of the language say, since they may not always agree with your intuition or training. For example, in the programming language APL, arithmetic expressions are evaluated from right to left. But FORTRAN was designed to be close to mathematical convention, and generally evaluates from left to right.

To look at a simple problem we are now prepared to solve, let us examine the idea of the *nanosecond*. A nanosecond is one billionth of a second (10^{-9} sec.), and many computer operations are so fast that their speed is measured in nanoseconds. To get a feeling for just how fast this is (it takes a billion of them to make up one second), let us calculate how long a billion seconds would be, since we have an appreciation for the duration of one second. We will write a FORTRAN expression to calculate how many years a billion seconds would be:

$$\text{YEARS} = 1.0 \text{ E } 9 / (60.0 * 60.0 * 24.0 * 365.25)$$

When we have added output statements to our FORTRAN toolbox, we would be able to print out this result, which turns out to be roughly 31.688 years.

If any variable names appear in the arithmetic expression, the computer looks at the value currently stored in the location with that name, and uses that value in the computation. You should be careful that variables used in your expressions have been *initialized*, that is, given values, before you refer to them in arithmetic expressions. If you do not do this, the results are unpredictable. You might expect the value of a variable initially to be zero, and this may be true on some computers that "zero" memory between programs loaded in, but even then it would not be helpful for non-numeric variables. Many computers do not "zero" or "clear" memory between jobs; on such machines, if you access a location before you have assigned it a value, it may contain some "garbage" value left over from the last program, and so do real damage to your results. Some computers (such as CDC) set all uninitialized

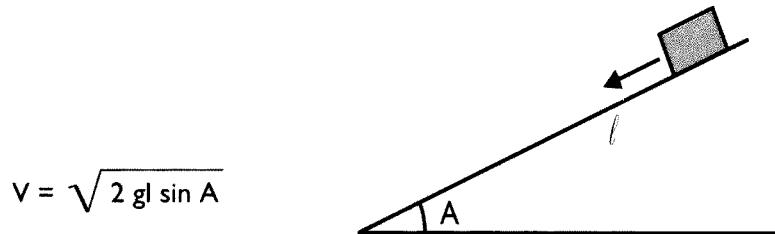
locations to an illegal value, so that if you try to use such a location in a calculation before you have initialized it, you will get an error during execution of your program, which stops the run.

To examine a short problem using variables, consider the distance a body falls in "free fall" under the force of gravity is $1/2 gt^2$, where g is the acceleration of gravity (which is 32.2 feet per second²), and t is the length of time of the fall, in seconds. Thus if we are given a length of time t of a fall, we can calculate the distance of the fall:

```
G = 32.2
T = 5.0
DIST = 0.5*G*T*T
```

(Notice that we use $T*T$ rather than $T**2$, because multiplication takes less time than exponentiation.)

We can make use of existing FORTRAN system functions in our calculations. For example, the velocity achieved by a body sliding down an inclined plane of length l , whose angle of inclination is A , if the plane has no friction, is:



To do this calculation, we need to know the length of the plane, its angle of inclination, and g . If we know these, we still need to be able to take the sine of an angle. We could look up the sine of our angle in a table, and enter it, but then we would have to do that for *any* angle involved; this would be very inefficient. It turns out that FORTRAN has a built-in system function SIN that will determine the sine of any real-valued angle expressed in radians. Thus, if we have an angle of the inclination of our plane expressed in degrees (such as 30°), we can convert it into radians by dividing by 360 and multiplying by 2 pi (since there are 2 pi radians in 360°). Then to take the square root of our product term, we could raise it to the 0.5 power, or we could use the system function SQRT, which takes the square root of a numeric argument and returns a value of the same type as the argument. Thus, to write our short program:

```
REAL PI, G, L, A, RADS, VEL
PI = 3.14159
G = 32.2
```

```

L = 3.4
A = 30.0
RADS = A/360.0*2.0*PI
VEL = SQRT( 2.0 * G * L * SIN(RADS) )

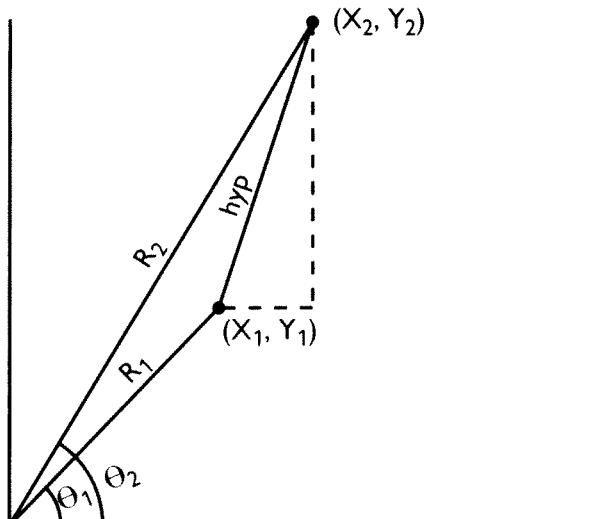
```

When we get more sophisticated, we can specify the values of the symbolic constants using a PARAMETER statement, such as:

```
PARAMETER (PI = 3.14159, G = 32.2) ,
```

input the values of L and A, and output the results. Note that the value of L should represent a length in *feet*, to correspond to our value of G, which is in terms of feet/second².

A complicated calculation can be broken up into several steps to make it simpler to handle; in such a case, various locations (variables) can be used to hold the intermediate results. Say that you are given the coordinates of two points in Euclidean two-dimensional space, (X₁, Y₁) and (X₂, Y₂). Part of your problem is to find the length of the line connecting these two points and the *slope* of the line. Each point can also be expressed in *polar* coordinates, as a *radius* R, or distance from the origin, plus the angle (θ) the radius vector pointing to the point makes. You are to translate the coordinates of each of the two points into polar coordinates. By examining the diagram of the points and the line connecting them we see that the *slope* of the line is just the y-difference (Y₂ - Y₁) divided by the x-difference (X₂ - X₁), and the length of the line is just the square root of the sum of the squares of the x-difference and the y-difference (by using the Pythagorean theorem). Once we have calculated these, we turn to the polar coordinates, and note the analogy to what we have already done. The radius vector R of each point is simply the line connecting the point to the origin, and its length is the hypoteneuse of the right triangle formed by the R-vector, the X-axis, and the perpendicular from the point to the X-axis. Thus we can easily calculate the R-values, and the angles are simply those which have the Y/X ratio as their tangents. The polar angles can be calculated as the arctangents of the Y/X ratios. We delve into Appendix B and find that FORTRAN has a function, ATAN (V), which returns the angle (in radians) which has the tangent value V.



```

REAL X1,Y1,X2,Y2,XDIF,YDIF,SLOPE,HYP,R1,R2,THETA1,THETA2
X1 = 3.0
Y1 = 4.0
X2 = 6.5
Y2 = 8.0
XDIF = X2 - X1
YDIF = Y2 - Y1
SLOPE = YDIF/XDIF
HYP = SQRT(XDIF*XDIF + YDIF*YDIF)
R1 = SQRT(X1*X1 + Y1*Y1)
R2 = SQRT(X2*X2 + Y2*Y2)
THETA1 = ATAN( Y1/X1 )
THETA2 = ATAN (Y2/X2 )

```

When we have learned how to do input and output, we could modify this program segment to *read* in the values of the coordinates of any two points, and *print* out the results.

Another question arises: when reals are combined with reals in an arithmetic operation, surely the numerical result will be a real; but what about when integers are combined with integers?

Integer Division/Truncation

When two integer values (constants or variables) are involved in an arithmetic operation in a FORTRAN expression, the result of that operation is also integer. This is fine for addition, subtraction, and multiplication—much as you would expect. But take a close look at integer division. If the result of an integer division is to be an integer, it may not give you the answer you anticipate. If you write $23/4$ in a FORTRAN expression, it will evaluate to the integer value 5. That is, the result of the division is *truncated*, the fractional part is lopped off, to make the result into an integer. If you are not aware of this, you may get some unexpected results in your programs. If you are aware of it, though, you can use it to your advantage in a number of computational situations.

Determine what values will be stored in the following integer locations as a result of the indicated instructions. Try them first, then check yourself against the answers given following the examples.

```

INTEGER M, N, K, KAT, LAD
M = 5
N = M**2
K = N/3 + 7
KAT = M*6/4
LAD = 1/2
M = M/3

```

[Location M is initially 5; N is 25; K is 15; KAT is 7; LAD is 0; and the value of location M is changed to 1 on the last line.]

Thus, if you had a number of containers (NCONT) and you wanted to know, if you divided them up into N equal-sized groups, how many would be in each group (NUM)—for example, 50 containers divided into 8 groups should give 6 in each group (and you would not worry about the two left over), you could calculate:

```
INTEGER NCONT, N, NUM
NUM = NCONT/N
```

Similarly, if you knew the group size you wanted (NUM), you could calculate how many complete equal-sized groups (N) you could get:

```
N = NCONT/NUM
```

Taking Remainders (A Useful Tool)

There turns out to be a number of situations in problems you will encounter where you need to determine the *remainder* when one whole number is divided by another. By remainder we do not mean the fractional part of a real quotient; a *remainder* is the number of whole units left over after the [integer] division, that is, after you have seen how many whole times the divisor will go into the dividend. For example, imagine that you have 39 apples to distribute *evenly* among 7 people. How many apples will each person get (the integer quotient), and how many will be left over for you (the remainder)? Each person will receive [39/7], or 5, apples, and there will be 4 apples left over (the remainder).

Let us generalize this operation to compute the remainder when any integer value M is divided by another integer value N. In the case of the apples, how did we calculate the remainder? First we determined the integer quotient 39/7 (or 5), which told us how many apples each person received, with no favoritism. Then we determined how many apples we had given away—the quotient (5) times the number of people we had distributed apples to (the divisor 7), or 35. Next, to determine how many we had left (the remainder), we subtracted the number given away (35) from the original number of apples (the dividend 39). So to calculate the number of remaining apples, we evaluated:

$$\text{remaining apples} = 39 - (\text{integer-quotient-of-}(39/7)) * 7$$

If we were writing this as a FORTRAN expression, since 39 and 7 are already integers, and when we divide 39 by 7 we will get the integer quotient, we can omit the qualifier “integer-quotient-of” and store the result in integer location NAPPLE:

```
NAPPLE = 39 - 39/7*7
```

Notice that no parentheses are needed, due to the left-to-right evaluation of the rightmost term of the expression. We can now generalize our “good old remainder formula” to calculate the remainder (call it LEFT) when any integer value M is divided by another integer value N:

```
INTEGER M, N, LEFT
LEFT = M - M/N*N
```

You can also use a system function called MOD, which will calculate the remainder when any integer value provided to it as the first “argument” (value in parentheses) is divided by the second integer argument. Thus, you could also write:

```
LEFT = MOD(M, N)
```

and get the same result.

In our container example in the previous section, if you had NCONT containers that were to be divided into N equal-sized groups, you might want to know how many odd containers would be left over, not included in any group; you could determine this by:

```
NLEFT = NCONT - NCONT/N*N or NLEFT = MOD (NCONT, N)
```

You will find that the ability to take a remainder will serve you in good stead in solving a number of different problems that you will face as you continue to program, including whether a year is a leap year, whether a number is a prime number, whether an integer is even or odd, and many other examples. Consider it another useful tool that you have just added to your collection of programming tools.

Mixed-Mode Arithmetic

We have seen what happens when reals are combined with reals in arithmetic operations, and when integers are combined with integers, but what about when reals are combined with integers (or when integers or reals are combined with complex or double precision values, which will be discussed in Chapter 5)? This is called “mixed-mode arithmetic,” and it was forbidden on early FORTRAN compilers, so that arithmetic expressions had to be entirely in terms of values of the same types. However, compilers have come a long way, and one of the added flexibilities is their ability to process mixed-mode arithmetic expressions. The rule is, if a real is combined with an integer, the

real value dominates, and the result will be real. This is simple enough, and applies in general. Thus, if you have an assignment statement:

$$A = 2.2 * 4$$

that involves a real and an integer constant, the result will be a real product, so the value 8.8 will be stored in location A. However, if we had written

$$N = 2.2 * 4$$

where N is by default an integer variable, the result (8.8) of the calculation would have been truncated to 8 before storing.

We have been looking at fairly simple arithmetic expressions in our examples so far, but they can become much more complex. In a complicated arithmetic expression, you must (as the compiled computer program does) look at *each* operation in the order it is performed, according to the order of precedence and the left-to-right rule, and determine for each operation whether its result will be real (if any real is involved) or integer (if the operands are both integer). Thus, examine the following statement:

$$A = 2.4 * 3 - 3.0 ** 2 + 13 / 5$$

In this expression, the exponentiation is performed first, giving a real result 9.0. Next, the multiplication is performed, following the left-to-right rule, giving a real result 7.2. Then the division is performed; since there are two integers involved in the division, the result will be integer, even though we have already had some real operations in the expression. Since this expression is an integer expression when it is executed, it will give an integer result, $13/5$, or 2. Next, the real 9.0 is subtracted from the real 7.2, giving a real result -1.8. Finally the real -1.8 is added to the integer 2, giving a real result of 0.2, which is then stored in the real location A.

Let us look at another complex arithmetic expression, and carefully trace the evaluation steps:

$$B = 3.5 / 2 + 5 - 4 * 2 + 5 * 6 / 4 - 5 * (6 / 4)$$

The diagram illustrates the step-by-step evaluation of the expression $B = 3.5 / 2 + 5 - 4 * 2 + 5 * 6 / 4 - 5 * (6 / 4)$. The expression is broken down into intermediate steps:

- $3.5 / 2 \rightarrow 1.75[3]$
- $5 \rightarrow 5$
- $4 * 2 \rightarrow 16[2]$
- $5 * 6 / 4 \rightarrow 30[4]$
- $5 * (6 / 4) \rightarrow 1[1]$

These intermediate results are then combined:

- $1.75[3] + 5 \rightarrow 6.75[9]$
- $6.75[9] - 16[2] \rightarrow -9.25[10]$
- $-9.25[10] + 30[4] \rightarrow 7[5]$
- $7[5] - 1[1] \rightarrow -2.25[11]$
- $-2.25[11] - 5[8] \rightarrow -7.75[12]$

The result of each operation is indicated, followed by a number in square brackets that indicates the order in which the operation was performed in evaluating the expression. Finally, the real result -7.75 is stored in real location B.

Now that you know how to evaluate any arithmetic expression that might appear on the right-hand side of an equals sign, we must pay attention to the locations in which the results of these evaluations will be stored. The assignment operation performed by the “=” makes sure that the right kind of numeric result is stored into the variable on the left, so that the type of the value stored agrees with the type of the variable. Thus, if the variable on the left of the “=” is an integer, then even if the result of the evaluation of the expression is real, it will be truncated before it is stored. Similarly, if the variable on the left of the “=” is type real then—even if the arithmetic expression on the right evaluates to an integer value—it will be “floated” (that is, converted into a real) before it is stored.

The type of the variable in which the result of the expression is to be stored does not come into play at all until the time the assignment is made; that is, it does not affect at all the evaluation process itself. But before the resulting value is stored, it is converted (if necessary) into the appropriate type of constant to match the numeric variable type. Again note that what we have said so far applies only in the case of expressions to be stored into integer or real variables. Similar conversions may be made at assignment for double precision and complex variables, and these will be discussed at the appropriate time, but a character value cannot be stored in a noncharacter type location, or a non-character value in a location of type character. These guidelines will help you to correctly interpret programs written by others, and to write expressions which will calculate as you intend. The following examples should help to clarify this.

	RESULT OF EVALUATION	STORED VALUE
INTEGER M, KAT		
REAL A, BAT, RAT		
M = 3.9	3.9	M 3
A = 8	8	A 8.0
KAT = 2.5*6 - 3.6	11.4	KAT 11
BAT = 4*6 + 3**2	33	BAT 33.0
RAT = 4**(1/2)	1	RAT 1.0

Sequence of Exponentiations—An Exception to a Rule

The last example ($4^{**}(1/2)$) might have been a bad attempt to take the square root of a number, and failed because the fraction $1/2$ truncated to 0. To evaluate exponential expressions, let us take a quick review of exponentiation operations.

$$\begin{aligned}
 a^n &= \underbrace{a \times a \times \dots \times a}_{n \text{ times}} \\
 a^{-n} &= 1/a^n \\
 a^{b+c} &= a^b \times a^c \\
 a^{bc} &= (a^b)^c \\
 a^{b/c} &= \sqrt[c]{a^b} \quad (\text{the } c\text{th root of } a^b)
 \end{aligned}$$

These rules will help us evaluate any exponential expressions that occur in FORTRAN. For example, if we want to take a square root of a value, we can raise it to the (1.0/2.0) power (or apply the SQRT function); to take the cube root of a number, we raise it to the (1.0/3.0) power; and so forth. Similarly, look carefully at the third rule to help you evaluate:

A = 16.0**2.5

The discussion of exponentiations raises one point that we must cover in the FORTRAN evaluation rules. We have said that, in evaluating an expression, if the precedence rules do not decide the order of operations, then they are evaluated left to right, *with one exception*. The exception is that of two (or more) exponentiations in a row. If we were to write:

MIKE = 2**3**2

without using any parentheses, the hierarchy table would not help us, and we would go to the left-to-right rule. However, in FORTRAN 77, that would be the wrong thing to do, since *this* is the exception to the left-to-right rule. In earlier FORTRAN compilers, there was no standardization regarding how such an expression would be evaluated. Some compilers applied the left-to-right rule, and others operated on the assumption that you really must have *meant* to evaluate the expression from right to left. FORTRAN 77 stepped in and resolved the uncertainty by dictating that the new Standard would evaluate such multiple exponentiation expressions from right to left.

The logic behind this decision is as follows. When we interpret the expression to be stored in MIKE, is it as

$(2^3)^2$ or as 2^{3^2} ?

The expression on the left, or $(2^{**3})^{**2}$, evaluates to 8^2 , or 64. But this could have been achieved more easily by evaluating the expression $2^{**}(3*2)$ or 2^{**6} , which is also 64, as we can see from the fourth of our rules regarding exponents. Thus, reasoned the designers of the FORTRAN 77 compiler, the programmer must have intended the expression to evaluate as on the right,

or $2^{**}(3^{**}2)$, which is $2^{**}9$, or 512, since if the other expression had been desired, there was a much simpler way to perform the evaluation.

If you have difficulty in remembering this exception to the normal left-to-right rule when you are writing a program, the solution is simple—just include parentheses to make perfectly clear the order in which you want the terms to be evaluated. But, in case you run across such an expression in someone else's program, without parentheses, you need to know the order in which the FORTRAN 77 compiler will evaluate it.

A *Fermat number* is one of the form $2^{2^n} + 1$; Fermat conjectured that numbers of this form are primes, and for $n = 0$ through $n = 4$ this proves true, but for $n = 5$ the Fermat number has a factor of 641. We can write a short program segment, using successive exponentiations, which will calculate the value of the Fermat number for a specified N (be careful of overflow!):

```
INTEGER FERMAT, N
N = 4
FERMAT = 2**2**N + 1
```

{for example}



SIMPLE (LIST-DIRECTED) INPUT/OUTPUT

With all of what we have covered so far, we still have not gotten beyond the stage where we can write anything more than a simple sequential program. To add testing (selection) and repetition (loops) we will wait until the next chapter, after you have mastered the fundamentals of handling FORTRAN expressions. Before that, however, there is one important capability to add to your repertoire—the ability to input values that your program can work on and the ability to output your results. You may not always want or need input to a program, since it may well be self-contained and provide all of the values that are needed, but you will always want to include one or more output statements, or else you will never find out the results of the calculations you have programmed the computer to make!

FORTRAN Input and Output (I/O) statements may be either formatted or list-directed. [Note: there is also a third alternative, that of unformatted I/O, but that is I/O in binary form that is only readable by another computer of the same type, and will not interest us at this point in the development of skills.] Formatted I/O is rather complicated, though it gives the FORTRAN programmer wonderful control over the handling of input information and the layout of output results; to go into it in great detail at this point would detract too much from our concentration on developing problem-solving tools. We will defer consideration of formatted I/O to a later chapter, and introduce a simpler form of I/O here, so we may begin writing programs.

List-directed I/O is an easy way to get information into and out of your

program, with a minimum of effort on your part. You do not have to provide any detailed descriptions of the way the information should look, if you are willing to follow a few simple rules regarding input and are willing to turn over control of the appearance of your output to the computer for the time being. A list-directed input statement is of the form:

`READ*, list`

and a list-directed output statement is of the form:

`PRINT*, list`

As the name indicates, the list of items following the “*” *directs* the way in which the information on the list will be handled by the computer.

In a READ* statement, the *list* is a list of variable names into which the values which are input will be stored. A READ* instruction reads values from the default input device, which for most of our programs will usually be the terminal from which the programmer is operating. In older batch systems, the default input was generally the card reader, and data on punched cards had to be prepared ahead of time and submitted at the end of the program deck. They would be read when the program executed the READ statements. In the case of a program today run from the terminal, the input may be interactive; it is entered at the terminal, using the keyboard, at the time when the program executes the READ* statement. This may be indicated by a “?” prompt on the screen, but some systems may have no prompt indicator. The user-conscious programmer will generally prefix any such request for input by a *prompt* to the user, such as:

`PRINT*, ‘ENTER YOUR AGE’`

Input may also be from tape or disk, in the form of prepared information that has already been written in *records* on the device. In such a case, a *unit* must be “opened” to the device so that the file of information may be read. For example, if your input data was written on disk in a file named ‘MYFILE’, a unit number *u* would have to be established to read from by using an OPEN statement (let us select unit number 4), of the form:

`OPEN (4, FILE = ‘MYFILE’, STATUS = ‘OLD’)`

and then READs from the file would be of the form:

`READ (4,*) list`

The use of external files and the OPEN statement will be discussed in greater length in Chapter 11.

Each successive READ accesses a new record, whether this is a record on the tape or disk, a punched card, or an input line from the terminal (the end of the input record from the terminal is indicated by the carriage return).

The items in the READ list may be any combination of types of variables. Initially we will assume that the list is no longer than the amount of corresponding input data that could be entered on one line at the terminal (or on one punched card), a limit generally of 72 to 80 columns, though on tape or disk it could be longer. Each item on the input data line should correspond in order, according to type, with its matching variable on the list. Real input values should contain a decimal point or be in E notation, integer values should be expressed as integers, character strings must be enclosed within single quotes, and so on. The input data items must be separated by blanks and/or commas. A representative list-directed READ* statement and a corresponding input line might be:

```
INTEGER GOALS, FREES, AGE
REAL HEIGHT, WEIGHT
CHARACTER NAME*10
```

{the "*10" in the CHARACTER type statement indicates that the variable NAME will contain a string 10 characters long}

```
READ*, NAME, AGE, HEIGHT, WEIGHT, GOALS, FREES
      /   \   /   \   /   \   /
' LARRY BIRD', 35, 81.0, 225.0, 16, 10
```

We have indicated the correspondence, term-by-term, of the variables on the list with the input data items. Any initial blanks on the input line are ignored, and the computer scans across the line until it reaches the first nonblank character to process. We have included extra blanks between the items for readability, but a single blank or a single comma is sufficient to separate two adjacent input items. The first four variables are fairly self-descriptive; GOALS is the number of field goals BIRD made in the game, and FREES is the number of free throws.

The list following a "PRINT*", for a list-directed output statement can be made up of variables, constants, expressions, and/or literal strings. The way in which these values will look on the output line (which either appears on the programmer's terminal screen or a line printer) is under the control of the computer. The values will appear in the *order* in which the programmer designated them, but the computer will determine whether to output real values in normal form or E format, the spacing between values, and so on; if duplicate values appear in a row, it might (on some systems, though this is nonstandard) output them using a repetition factor followed by a "*" and the

repeated value. Character strings will be output literally, without surrounding quotes, and there will be one or more blanks or a comma between output items; this is system-dependent.

A possible relationship between a PRINT* statement and its resulting output line is the following, where the real variable A contains the value 1.1, and the integer variable I has the value 3:

```
PRINT*, A, A**2, 5, I+2, 'END OF THE LINE'
      +-----+
      |       |
      1.1   1.21  5   5  END OF THE LINE
```

(or, on some systems,

```
1.1  1.21  2*5 END OF THE LINE
```

using a repetition factor r^* preceding a value that occurs several times successively). From these examples, you should be able to get a rough idea of how to use the list-directed I/O statements in a program.

We could modify our inclined-plane program to read in the values of the length of the plane L and the angle A, and to print out the resulting velocity VEL, by including statements:

```
READ*, L, A
PRINT*, 'THE RESULTING VELOCITY IS ', VEL, ' FEET PER SEC.'
```

We could also modify the program for the two points in Euclidean space to read in the coordinates of the two points:

```
READ*, X1, Y1, X2, Y2
```

and to print out the resulting length, slope, and polar coordinates:

```
PRINT*, 'LENGTH IS ', HYP, '; SLOPE IS ', SLOPE
PRINT*, 'POLAR COORDINATES OF POINT ONE ARE ', R1, THETA1
PRINT*, 'POLAR COORDINATES OF POINT TWO ARE ', R2, THETA2
```

The inclusion of input statements for variables allows us the flexibility to write programs that can work on many different sets of values, and the inclusion of output (PRINT) statements is crucial if we want to see the results of the calculations that the program performs.



CREATING FORTRAN ARITHMETIC EXPRESSIONS

You should be familiar with algebraic equations and scientific formulas which you might want to translate into FORTRAN statements for use in a program. This is relatively straightforward, except that variable names in such algebraic equations are generally single letters, two names written adjacent to one another implies multiplication, and so on; these operations must be properly translated into FORTRAN. Some examples of algebraic formulations and their translation into FORTRAN follow:

Algebraic	FORTRAN statement
$y = ax + b$	$Y = A*X + B$
$x^2 = 25 - y^2$ (solve for x)	$X = (25.0 - Y**2)**0.5$ or $X = SQRT(25.0 - Y**2)$
$z = \frac{a + b}{c + d}$	$Z = (A + B)/(C + D)$ {the parentheses are necessary; why?}
$c = mv^2$	$C = M*V**2$
$d = \frac{b + t^3}{ac}$	$D = (B + T**3)/(A*C)$ {are both sets of parentheses needed?}
$a + b = cd$ (solve for a)	$A = C*D - B$
$x = 3y + 4z$	$X = 3*Y + 4*Z$

We can now write a simple program using input and output which will calculate the recoil velocity of a large cannon, after shooting out a clown at the circus, if it is sitting on wheels on a frictionless surface. We will need to specify the weight of the gun (WTGUN), the weight of the shot (that is, the clown) fired (WTSBOT), and the velocity at which the clown travels (VSHOT). The law of conservation of momentum indicates that the total momentum of the system (which is initially at rest, so equal to zero) must remain the same. Thus,

$$0 = m_g v_g + m_s v_s$$

or

$$m_g v_g = -m_s v_s \quad \text{or} \quad v_g = -(m_s/m_g)v_s$$

The formulas for momentum are in terms of mass x velocity ($p = mv$), and we have weights rather than masses; however, since weight = mass x g, and since we are using a ratio of two masses in our final equation, the g's would cancel out, so we can just calculate the ratio of the two weights. Thus a program might be:

```

REAL WTGUN, WTSBOT, VGUN, VSHOT
PRINT*, 'INPUT THE WEIGHTS OF THE CANNON AND THE CLOWN'
READ*, WTGUN, WTSBOT
PRINT*, 'INPUT THE VELOCITY OF THE SHOT'
PRINT*, VSHOT
VGUN = - (WTSBOT/WTGUN)*VSHOT
PRINT*, 'THE RESULTING VELOCITY OF THE CANNON IS ', VGUN
PRINT*, 'THAT IS, IT MOVES AT THAT SPEED IN THE DIRECTION'
PRINT*, 'OPPOSITE TO THAT OF THE CLOWN'
```



OTHER SYNTACTIC RULES FOR WELL-FORMED INSTRUCTIONS

In forming arithmetic assignment statements, we have already looked at the general form of the statement. We know that *only* the name of a single variable can appear to the left of the “=”, and that the arithmetic expressions to the right must represent legitimate arithmetic operations (or else invoke functions that do so), and there must be an operator for every pair of operands. Thus, multiplication cannot be expressed as it is in algebra. If you want to multiply A by B you must write A*B, *not* AB. There are some other rules of *syntax* (proper form or grammar) that we must be aware of in writing FORTRAN assignment statements.

1. If there are multiple sets of parentheses in a statement, they must *match* exactly. Each left parenthesis must have a matching right parenthesis, so there must be the same number of left parens as of right parens; e.g., $(3 + (4 * (5 * 2)))$.
2. No two operators may be written adjacent to one another. Of course you would not want to write A*/B, but this also means that you may not write A*-B or X**-3. To get around this difficulty, enclose the negated quantity in parentheses, and thus avoid the illegality. You may write A*(-B) or -B*A, and X**(-3).
3. You may not raise a negative quantity to a real power. This is to avoid trying to take imaginary roots. There is a provision for complex numbers in FORTRAN, but it is handled in a special way, and you cannot take imaginary roots to create complex values. Thus, $(-4.0)**0.5$ would be illegal in FORTRAN, but $(2.6)**(-1.0/3.0)$ is legal, and $(-3.3)**4$ is legal.



THE PHYSICAL LAYOUT OF A FORTRAN STATEMENT

Though some systems allow what is called “free-field” for a FORTRAN instruction (that is, column locations do not matter), this is not standard, so you must learn the appropriate form in which to format your FORTRAN instructions. Since, originally, FORTRAN was run batch from decks of punched cards, the structure of a line of FORTRAN is tied to the 80-column format of a punched card. A FORTRAN instruction was expressed in terms of four *fields*, or areas of the input line. It is simplest to visualize these as segments of an 80-column card, or of a line on a screen:

1 . . . 5 6 7 . . .	72 73.. 80
statement * FORTRAN instruction	sequence #
number	[optional]

Thus, columns 1–5 of a program line are to be used for a statement number (if any) to label the line. Column 6 (which we have identified with a “*”) is the *continuation column*; if an instruction is too long to fit onto one program line, it may be continued onto succeeding lines, but each line which represents a continuation of an earlier line must have some character (anything other than blank or zero) in column 6. Free-field systems do not recognize the traditional FORTRAN fields, and thus they must have some other method for designating continuation lines; if you have such a system, consult your instructor or your computer manual to determine the convention at your installation. However, most free-field systems are not so “free” that you can put the statement number anywhere you please; on most, it must still appear in the first five columns.

Columns 7–72 are used for the FORTRAN instruction itself (assignment statement, I/O, or test or repetition statements we have not yet covered, or type declaration statements, etc.). On a free-field system, a statement that does not have a statement number labelling it may begin in any column 1–72. Columns 73–80 are not interpreted by the compiler, and on old punched-card systems were often used to sequence the cards in a program deck, just in case someone dropped it! These columns may not be available on your terminal screen, and are generally disregarded in modern FORTRAN programming.

On most systems implementing the FORTRAN 77 Standard, you must conform to these field restrictions. Most statements will not require a statement number; for readability it is best only to put them on statements that need them. Thus, for the majority of your FORTRAN instructions, you must space over to at least column 7 of your terminal screen (or punched card) before beginning the statement. [Note: The Fortran 90 revision will not enforce these fixed-field conventions.]

The Standard FORTRAN 77 character set consists of 26 capital letters (A .. Z), ten digits (0 .. 9), a blank, and 12 special characters (= + - * / () . , \$ ' :). On many

systems, these are the only symbols available for your FORTRAN programs, and so to conform with the Standard on these systems, all FORTRAN instructions must be represented using these symbols. This means that all FORTRAN commands, such as PRINT, must be written in capital letters. Many current systems will allow (non-standardly) FORTRAN commands in lower-case letters (such as print) or a combination of upper- and lower-case letters (such as Print); the lower-case letter is considered by FORTRAN to be the equivalent of its upper-case sibling. In this text we will use the restricted Standard of all capitals. If your system allows lower-case as well, you may use them, but realize that this may make your program non-portable to some installations.

Standard FORTRAN 77 only allows one FORTRAN statement per line. Thus, whether numbered or not, each new statement must begin on a new line. Some compilers allow (non-standardly) the writing of more than one short instruction per line, as long as they are separated by the appropriate special character (usually a ";"). This feature will be incorporated into Fortran 90.

```
A = B ; C = 5.3 ; D = 9.0*A**2 ; N = 2 [non-Standard]
```

In a FORTRAN instruction, the compiler ignores blanks except when they occur as part of a literal string (that is, characters in quotes). Thus, you may use liberal spacing to make your program more readable, and you do not *have* to begin statements in column 7. Thus, the following statements are interpreted the same way by the compiler:

```
A=B*3+C/D
A = B * 3 + C / D
```

This also allows indentation of certain "blocks" of code that form a unit (such as in a loop or a Block IF, which we will discuss) so that they are recognizable as a group. In the case of a continuation statement, the continuation character is not considered part of the rest of the instruction, but everything in columns 7-72 of the continuation line is considered as an extension of the line before it.

```
col. 6
A = 3 * B**T-4 - X/Y/Z + TIME*DIST**2 - 4*BREAK/UP +
# CANNOT*DOIT**END
```

In this example, the "#" is in column 6, and indicates that what follows it is a continuation of the assignment statement for A. In Fortran 90, a '&' at the end of a line will indicate that the following line is a continuation of this one.

Comments

An exception to beginning a FORTRAN instruction in column 7 or thereafter is the Comment statement, which allows the programmer to insert helpful remarks about the program, the algorithms used, any tricky code, and so on. Comment statements are ignored by the compiler, but they appear as part of any listing of the program on the terminal screen or the printer, and they constitute useful internal documentation of the program. The traditional way to designate a comment statement in FORTRAN was to put a C in column 1 of the line; then the rest of the line is ignored by the compiler, but still displayed. This convention is still used, but FORTRAN 77 has added the rule that an asterisk (*) may also be used in column 1 to designate a comment. This allows the writing of comments that stand out, such as:

```
***** JUDY'S FIRST PROGRAM *****
```

Systems that use free-field must have a different convention for comments, since columns do not have special significance on such a system; if you have a free-field implementation, consult the FORTRAN manual for your machine regarding indicating comments (for example, a comment may require either a C* or a C- in the first two columns of the line). The Fortran 90 revision will adopt an additional convention for comments, such that anything following a '!' on a line is a comment (see Appendix E).

You may not use a continuation line to continue a comment.



PROGRAM STATEMENT

FORTRAN 77 allows the optional inclusion of a PROGRAM statement at the beginning of a program, to give it a name:

```
PROGRAM name
```

where *name* conforms to the variable-naming conventions in the language. This option did not exist, except on a few compilers, prior to FORTRAN 77. It has been included to allow reference to the main program by name, since subprograms are identified by name. In the past, the basic program unit was usually identified simply as the "main" program.

 **STOP AND END STATEMENTS**

In a main program (the only kind we will be writing initially), it is optional to include one or more STOP statements at points in the program where execution is to cease. The STOP statement notifies the system that this particular program is finished with its job, and it can go on to another program. If a STOP statement is not included in a program in FORTRAN 77, the END statement is assumed to indicate termination of execution.

A non-optional statement that is required at the very end of the main program (and at the end of any other [sub]program unit) is the END statement. This statement serves as a flag to the compiler that it has completed all of the instructions in the program unit that need to be translated into machine code. This is necessary because we may have more than one program unit (for example, a main program and several subprograms) in one file.

 **DEFINED SYMBOLIC CONSTANTS
(THE PARAMETER STATEMENT)**

A new feature added in FORTRAN 77 was the PARAMETER statement, which allows the programmer to give names to important constants that may be used throughout the program. A name given to a constant in this way differs from assigning a constant to a variable (such as $N = 5$), since the variable, in this case N , can always be changed during the course of the program. If instead N is a symbolic name given to the constant 5 by a PARAMETER statement {PARAMETER ($N=5$)}, the value of N can *never* be changed in the program unit in which it appears. This is an advantage, since it prevents you from accidentally changing a value that you do not want changed, and yet it gives you the flexibility of using a symbolic name instead of the constant throughout. This allows such a constant value which is used many times in the program to be readily altered by changing just *one* statement, the PARAMETER statement, if the "parameters" of the program definition change. However, if your program contains several program units (such as subprograms), PARAMETERs must be declared appropriately in each program unit.

Suppose, for example, that your program was initially written to process a set of 100 names (say, of biological species), alphabetize them, group them according to some criterion such as protein differences, and perform several other tasks on the names. The value 100 thus would appear many times throughout the program to determine the number of loop repetitions and so on. Then someone adopts your program, but wants it to run for 1000 names. If you had used the constant 100 throughout, you would have to go through

and make many changes of the value to 1000, and you might miss some. If the parameter 100 had been given a symbolic name at the beginning of the program in a PARAMETER statement, then you would only have to change that one occurrence of 100 to 1000. This is a great advantage in providing flexibility for your program while maintaining its *integrity*, or freedom from accidental error.

The PARAMETER statement must appear at the beginning of the program, along with other declaration statements such as type statements, and before any executable statements. It has the general form:

```
PARAMETER (name1=exp1 [,name2=exp2 . . .])
```

where one or more symbolic names may be defined using a constant or a constant expression (any arithmetic expression in terms of constants or previously defined symbolic constants). If a name appearing in a PARAMETER statement does not follow the implicit typing of FORTRAN, its type *must* have been declared in a type statement that precedes the PARAMETER statement. Any symbolic constant so defined may not be altered in the program.

The following are examples of the use of the PARAMETER statement in a program:

```
PARAMETER (PI = 3.14159, N = 100)
CHARACTER ALPHA
REAL MAD
PARAMETER (TWOPI = 2*PI, ALPHA = 'A', MAD = N**2, KAT = 2)
```

In addition to allowing a parameter to be changed easily throughout a program, and preventing accidental altering of a value that should remain constant, having a symbolic name instead of a constant in the program logic will generally be much more meaningful to anyone who has to read and understand the program.



SIMPLE FORTRAN PROGRAM EXAMPLES

We will include two different program examples here, one scientific and one general-purpose. Throughout the text we will mix nonscientific examples in with scientific ones, to vary the pace and flavor and applicability of what we are learning. Even though this text is aimed primarily at scientists and engineers, not all interesting problems in life are of a scientific nature. You may also, for example, have many administrative tasks to perform. FORTRAN has broad problem-solving capabilities, and we want to explore their range. The tools you develop through these examples will be useful in a wide range of problems.

Gravitational Force (Two-Body Problem)

The principle of Universal Gravitation as expressed by Newton is:

If spheres be however dissimilar (as to density of matter and attractive force) in the same ratio onward from the center to the circumference, but everywhere similar at every given distance from the center, on all sides round about; and the attractive force of every point decreases as the square of the distance of the body attracted: I say that the whole force with which one of these spheres attracts the other will be inversely proportional to the square of the distance of the centers.

Principia, Book I, Proposition LXXVI

A more modern formulation of the law is:

Every particle of matter attracts every other particle with a force varying directly as the product of their masses and inversely as the square of the distance between them.

This law is expressed in the formula

$$F = \frac{GMm}{r^2}$$

where M and m are the two masses (in kg), r is the distance between them (in meters), and G is the universal gravitational constant: $G = 6.67 \times 10^{-11}$ newton-meters²/kg². You can then write a program to determine the gravitational force between any two bodies, given their distances (center-to-center) and their masses. We will write a program which will accept two masses and the distance between them, and calculate and print out the gravitational force (which will be in newtons). The main FORTRAN expression that is needed to do the calculation will be:

FORCE = G*MASS1*MASS2/R**2

and the rest of the program involves defining the value of G, prompting for the input of MASS1, MASS2, and R, printing the output, and some comments for documentation. If you want to make use of this program, the following table of planet masses and their distances from the sun could be run through the program to calculate the force between each planet and the sun. If you know the distances between planets, you can calculate the gravitational force between any two planets. We will also suggest a modification of this problem for you as an exercise.

Note that the mean distances of the planets in the table (on the next page) are in terms of kilometers (km); be sure to convert them to meters (m) before using them in the program.

Mass of the Sun = 1.99×10^{30} kg.

THE PLANETS

Planet	Distance from the Sun	Mass
Mercury	57.9 x 10^6 km	3.30×10^{23} kg
Venus	108 x 10^6 km	4.87×10^{24} kg
Earth	150 x 10^6 km	5.98×10^{24} kg
Mars	228 x 10^6 km	6.42×10^{23} kg
Jupiter	778 x 10^6 km	1.90×10^{27} kg
Saturn	1430 x 10^6 km	5.67×10^{26} kg
Uranus	2870 x 10^6 km	8.70×10^{25} kg
Neptune	4500 x 10^6 km	1.03×10^{26} kg
Pluto	5910 x 10^6 km	6.6×10^{23} kg

```
***** UNIVERSAL GRAVITATION PROGRAM *****
***** FORMULA THANKS TO SIR ISAAC NEWTON *****
CHARACTER BODY1*10, BODY2*10
REAL MASS1, MASS2, R, FORCE, G
PARAMETER ( G = 6.67 E -11 )
***** PROMPT USER TO INPUT DATA *****
PRINT*, 'WELCOME TO THE UNIVERSAL GRAVITATION PROGRAM'
PRINT*
PRINT*, 'PLEASE ENTER THE NAMES OF YOUR TWO BODIES IN'
PRINT*, 'SINGLE QUOTES - AS ''VENUS'', ''SUN'' '
READ*, BODY1, BODY2
PRINT*, 'PLEASE ENTER THE MASSES OF YOUR TWO BODIES IN'
PRINT*, 'KILOGRAMS, AS REALS, SEPARATED BY A COMMA'
READ*, MASS1, MASS2
PRINT*, 'PLEASE ENTER THE DISTANCE BETWEEN THE TWO'
PRINT*, 'BODIES IN METERS, EXPRESSED AS A REAL'
READ*, R
FORCE = G*MASS1*MASS2/R**2
PRINT*, 'THE FORCE BETWEEN ', BODY1, ' AND ', BODY2
PRINT*, ' IS ', FORCE, ' NEWTONS'
STOP
END
```

Notice that we could have simply defined the two masses and their distance in the program, using assignment statements, and made it much shorter. But this way it is more flexible and can be used many times to calculate the force between different body pairs.

Installment Loan Payments

When you go to the bank to borrow money to buy a computer or a car, the banker asks how much you want to borrow, and for how many years, and then quickly looks up or calculates how much your monthly payments will be for that amount and time period for the going rate of interest. Sometimes the number you are given comes as a shock. It would be nice to be able to calculate for yourself, before you go into the bank, how much the monthly payments would be, so that you are prepared. In addition, the calculation would also provide you with a comforting check on the bank's calculations, so that the whole thing does not seem like some mysterious ritual.

All you need to know is the amount you want to borrow (principal P), the number of years you want to take to pay off the loan (years Y), and the current yearly bank rate of interest (R), expressed in the form of the decimal equivalent of percent. Then all you have to do is plug these values into the following somewhat complicated formula, and you can calculate the monthly payment for any set of these three values.

$$\text{monthly payment} = \frac{PR/12}{1 - (1+R/12)^{(-12Y)}}$$

This is a slightly complicated formula, but since (so far) we can only write sequential programs, at least we can pick one that would be difficult to do on most hand calculators.

First, we should organize the problem according to the input we expect to enter, and the form of the output we want; then, we can concern ourselves about the details in between.

INPUT: Amount you want to borrow (call it LOAN)
 Number of years of loan (call it NYEARS)
 Current bank interest in % (Call it PRCENT)

Note: It is much easier to have the user input the current bank rate of interest as a percent, since that is the way it will be expressed in the newspapers or in the bank publications. We can then have the *program* convert this amount into a decimal equivalent of percent simply by dividing by 100.

A "bare-bones" version of the program, with no comments and no prompt, and a simple output statement which just prints the result would look like the following:

```

READ*, LOAN, NYEARS, PRCENT
RATE = PRCENT/100.0
TERM = RATE/12.0
PAYMNT = LOAN*TERM/(1 - (1 + TERM)**(-12*NYEARS))
PRINT*, 'MONTHLY PAYMENT IS $', PAYMNT
STOP
END

```

However, we will "flesh out" this program with descriptive comments, informative prompts and output, and a method to make the output amount look more readable. The output from the program should reflect the values we input (the amount borrowed, the number of years, the going rate of interest) and the calculated monthly payment. The amount borrowed is expressed as an integer variable LOAN, since you seldom go to the bank and ask to borrow a big amount expressed in dollars *and cents*, but the monthly payment will be a real variable, since loan payments generally *are* expressed in dollars and cents. However, since we will be using list-directed output, which is all you know so far, we cannot control the form of the output to look like a dollars-and-cents amount, and it will most likely come out something like 145.87342, which is not desirable. Thus, we will try a trick to make it look more like a money amount. If the digits after the first two cents digits are zeros, the list-directed I/O might not print them (and even if it does, we can still concentrate only on the first two significant cents digits, since the rest will be zeros). If we multiply the amount (rounded) by 100 and store it in an integer location, it will only retain two places of the cents amount; then we divide it again by 100.0 before output to return it to the proper amount.

So that this program can be used by you or by any of your friends, we have included "prompts" indicating when the data should be input, and in what order it should be typed. Our output statements will be in a meaningful form.

```

***** COMPUTER OR CAR LOAN PROGRAM *****
***** WRITTEN BY I. M. BROKE ON 1/1/93 *****
* INPUT VARIABLES ARE:
*   LOAN (AMOUNT TO BE BORROWED)
*   NYEARS (PERIOD OF THE LOAN IN YRS)
*   PRCENT (BANK'S CURRENT RATE IN %)
* OUTPUT VARIABLE IS:
*   PAYMNT (CALCULATED PAYMENT IN $)

* OTHER VARIABLES ARE:
*   RATE (PRCENT/100, DECIMAL EQUIV.)
*   TERM (IN EQUATION, = RATE/12)

```

```

*      MNTHLY (100*PAYMNT, FOR DISPLAY)
**** IMPLICIT TYPING OF VARIABLES IS USED IN THIS PROGRAM ****
***** PROMPT USER TO ENTER INPUT VALUES *****
      PRINT*, 'ENTER AN AMOUNT TO BE BORROWED AS A WHOLE NUMBER,'
      PRINT*, 'THE NUMBER OF YEARS FOR WHICH YOU WANT THE LOAN,'
      PRINT*, 'AND THE CURRENT BANK LOAN RATE EXPRESSED AS A '
      PRINT*, 'REAL VALUE WITH A DECIMAL POINT AND REPRESENTING'
      PRINT*, 'A PERCENTAGE RATE. ENTER THEM ALL ON ONE LINE, '
      PRINT*, 'SEPARATED BY BLANKS, AND DO NOT INCLUDE ANY '
      PRINT*, 'SPECIAL CHARACTERS SUCH AS $ OR %'
      READ*, LOAN, NYEARS, PRCENT

      RATE = PRCENT/100.0
      TERM = RATE/12.0
      PAYMNT = LOAN*TERM/(1 - (1 + TERM)**(-12*NYEARS))
***** SCALE UP BY 100 TO TRUNCATE TRAILING DIGITS *****
      MNTHLY = (PAYMNT + .005)*100
      PAYMNT = MNTHLY/100.0

      PRINT*, 'LOAN OF $',LOAN,' OVER ',NYEARS,' YEARS'
      PRINT*, 'AT A RATE OF ', PRCENT, '%'
      PRINT*, 'RESULTS IN A MONTHLY PAYMENT OF $', PAYMNT
      STOP
      END

```

We have used extensive comments at the beginning of the program to describe every variable that is used in the program. Modern programming practice suggests such detailed internal documentation, to enhance program readability. We also made the input prompt very detailed, assuming an inexperienced user. This may not be necessary in all cases—for example, in programs for your own exclusive use or in non-interactive programs. We used blank lines to separate the input, computational, and output sections of the program for readability. If your system will not accept a blank line as part of a program, put a “**” in the first column—the effect is much the same. Detailed comments and readable programs will be of great benefit to programmers working on a team to solve a large-scale problem.

In the next chapter, we will cover FORTRAN constructs which will allow you to perform tests and to create primitive repetition structures. You will then be able to write even more interesting programs.



FORTRAN 90 FEATURES

The length of a variable may be up to 31 characters (alphabetic and numeric) long, and may include an underline (_).

Character strings may be put in either single or double quotes.

Type statements may be expressed in additional ways, and may be used to initialize variables (see Chapter 5 and Appendix E).

The IMPLICIT NONE statement may be used at the beginning of a program to require that all variables be explicitly typed.

The fixed-field restrictions may not apply in Fortran 90 processors. Free-field and fixed-field statements may not be mixed in one program unit. In a *free source form*, a line may have up to 132 characters; a blank must be used to separate adjacent names, constants, or statement labels.

The statement separator ";" may be used to separate several (non-numbered) statements appearing on the same line.

The character "&" at the end of a line indicates that the statement is continued on the next line.

A comment follows a "!" wherever it appears on a line (unless the "!" is part of a character list in quotes).



SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

The distinction between a *constant* (such as 3, 0.567, or 'HELLO') and a *variable* (a location to contain constants, given a name such as LENGTH or PAYMNT), and their relations to computer operations, were discussed. The six data types available in FORTRAN are: integer, real, double precision, complex, character, and logical. Rules for naming variables and declaring the type of values they will contain were presented. The *assignment statement* as a means of storing a value in a variable was explained. To review, the order of precedence of arithmetic operations is: expressions in parentheses, function references, exponentiations, multiplications and divisions, additions and subtractions, and finally assignment (if necessary). If the precedence rule still leaves any ambiguity, expressions are evaluated left to right (except for successive exponentiations).

Rules according to which mixed-mode expressions give results were discussed, as was the fact that integer division truncates its result, and provides a method for calculating remainders. Simple input and output can be performed using the *list-directed* READ* and PRINT* commands. FORTRAN stands for FORMula TRANSLator, so examples were given of the translation of algebraic formulas into FORTRAN statements. Additional syntax (grammatical) rules for well-formed FORTRAN statements were presented, along with the proper layout of a FORTRAN instruction (first five columns for statement number, column 6 for a continuation indicator, and columns 7–72 for the FORTRAN instruction).

Comments may be included in a program by preceding them with a C or * in column 1. The PROGRAM statement can be used to give an identifying name

to a program; a STOP statement may be used to indicate the point at which program operation terminates, and every program unit must have an END statement as its last command (to indicate to the compiler the *end* of statements to be translated into FORTRAN). The PARAMETER statement may be used to define symbolic names for constants, whose values then cannot be altered in the program unit in which the PARAMETER statement appears.



EXERCISES

1. Identify each of the following appropriately as a legal implicitly typed variable name, a legal constant (indicate what type of variable or constant it is), or as illegal. For those cases that are illegal, say why they are illegal.

```
'B'      375      HELPME    $75.43   #1      PRINT     8.54 E-3
TERMINATOR TEST    NAME    M16    2,645,876.99   .04 E +2000
```

2. Assume that the following assignments of values have been made to these variables, which are implicitly typed: A = 3.5, B = 6.0, C = 4.0, M = 5, N = 3, K = 2. Indicate what value would be stored in the integer location J or the real location D as a result of each of the following FORTRAN assignment statements.

J = M + N/K	D = B + C**2	J = M - M/K*K
J = B + A	D = C**(1/K)	D = C**(-0.5)
• J = C+B/K+N	D = -A+B	J = N*N**K
D = C**A	J = 25**1/2	D = B*C/K*K

3. Write the following algebraic equations as FORTRAN instructions which store the appropriate result in location X, assuming that values exist for the variables A, B, C, D, and Y (which stand, respectively, for the algebraic variables a, b, c, d and y; and X for the algebraic variable x). All variables are type real.

$$\bullet \quad x = \sqrt[3]{a^2 + b^2}$$

$$x = \frac{a}{b} - \frac{c+d}{by}$$

$$x^2 + 5 = y^2$$

$$\bullet \quad x = \frac{1}{2}ay^2$$

$$x = 3y - 4ab$$

$$x = \frac{ab^3}{cd}$$

4. Write a program that will read in your name and print it out, followed by your vital statistics. Use a variable of type CHARACTER*n for your name, where n is an integer large enough to accommodate all of the letters and spaces in your name.

5. Write a program to print out your first name in large block letters like the following:

A	BBBBB
A A	B B
A A	B B
AAAAAAA	BBBBBB
A A	B B
A A	B B
A	BBBBB (or better)

6. Write a program which will accept a positive real number as input, and print out the number, its square, its cube, its square root, and its cube root; label your output appropriately.

- 7. Steve Martin, playing a fire chief in the movie "Roxanne," directs a stream of water from a fire hose at a building that is x feet away. The hose is held at an angle a (radians) above the horizontal. The velocity of the stream of water is v. The time t the water takes to reach the building can be determined from the equation $x = (v_x)t$, where v_x is the component of the velocity v in the horizontal (x-) direction. You want to determine the height y on the building where the stream of water will hit. This is determined by $(v_y)t - 1/2gt^2$. Input the relevant values, solve for the height y, and output the answer. Use the system functions SIN and COS, which take a real argument expressed in radians, such as SIN(A) and COS(B), where A and B are real values expressed as angles in radians.

8. When resistors are wired in *parallel*, their effective resistance R is calculated from the following equation:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \dots + \frac{1}{R_n}$$

Write a program which will accept the values of resistance for five resistors wired in parallel, and compute and print out the overall effective resistance of the system.

9. The universe is estimated to be on the order of 4×10^{17} seconds old. Write a program to input your own age in years, days, hours, minutes, and seconds, calculate your age in seconds, and then print out your age in seconds and the fraction representing the ratio of your age to the age of the universe. Try expressing your age as a percentage of that of the universe.

10. Ammonia is 82% nitrogen and 18% hydrogen. If you have 10 grams each of nitrogen and hydrogen, how much ammonia (by weight) will this make? Write a FORTRAN program to calculate and print out the answer to the problem. Now generalize your program so that it will accept as input any two amounts of nitrogen and hydrogen (in grams) and calculate and print out how much ammonia it will make, in an informative manner. You may assume that there is enough hydrogen to go with the nitrogen. (After the next chapter, you will be able to test for that, and handle the possibility that there is not enough hydrogen to go with all of the nitrogen.)

- **11.** A *calorie* is defined as the quantity of heat required to raise the temperature of 1 gram of water by 1° Centigrade (specifically, from 4°C to 5°C). The *specific heat* of a substance is the amount of heat needed to raise the temperature of 1 gram of that substance by 1°C . The specific heat of gold is 0.0306 (calories/gram-degree). Write a FORTRAN program that will calculate and print out how many calories it will take to raise the temperature of 120 grams of gold from 45°C to 60°C . Then generalize the program so that it will accept as input any amount of gold (expressed in grams), and any temperature range (two temperatures input, the lower one first), and print out an informative result, saying how many calories it will take to raise X grams of gold from A to B $^{\circ}\text{C}$.
 - Now generalize the program even further, so that it will accept as input the name of an element and its specific heat, as well as the amount of the element and the temperature range. Have your program print out the name of the element, the amount and temperature range, and the number of calories required. Make use of information from the table in the next problem.
- 12.** The Law of Dulong and Petit states that the product of the atomic weight and the specific heat of many solid elements is approximately 6.3. Write a FORTRAN program to test out the validity of this law by reading in the name of an element, its specific heat, and its atomic weight, and calculating the Dulong-Petit product. Print out the name of the element and the product you have calculated. Use the following table:

Element	Specific Heat (cal./g-deg)	Atomic Weight
Gold	0.0306	197.0
Silver	0.0565	107.88
Copper	0.0922	63.54
Tin	0.0531	118.7
Calcium	0.1157	40.08
Aluminum	0.216	26.98

Since we have not learned how to use loops yet, you will have to run your program multiple times to try out the several different elements. This will lead you to look forward to loops.

13. Read in two times, each with three components, expressed as military time (from 0:0:0 to 23:59:59), the earlier time first. Have your FORTRAN program calculate and print out the difference between the two times in seconds. Then convert the seconds to hours, minutes, and seconds, and print out the result.

14. Write a FORTRAN program which will convert a value in degrees Fahrenheit to one in degrees Celsius, according to:

$$^{\circ}\text{C} = 5/9 (^{\circ}\text{F} - 32)$$

Then convert the result to degrees Kelvin ($^{\circ}\text{K} = ^{\circ}\text{C} + 273.16$).

Now write a program which converts in the other direction, from degrees Celsius to degrees Fahrenheit. You should be able to determine the formula for this from the first formula. Then convert the temperature in degrees Fahrenheit to degrees Rankine ($^{\circ}\text{R} = ^{\circ}\text{F} + 459.69$).

- **15.** Modify the Universal Gravitation program to calculate the force between you and the Earth. You will need your own weight in kilograms (you probably know it in pounds; have the program perform the conversion for you; 1 pound-mass = 0.453592 kg). You will also need to know the radius of the Earth (its mass is given in the table in the text); the radius of the Earth is 6378 kilometers (at the equator).

16. Using the Universal Gravitation formula, we can use information regarding the distance of our solar system from the center of our galaxy (which is the Milky Way) and its orbital velocity with respect to the center of the galaxy to estimate the mass of the Milky Way galaxy. The force of attraction between our sun and the center of the galaxy can be represented by the gravitation formula:

$$F = G M_{\text{galaxy}} m_{\text{sun}} / r^2$$

and, since $F = ma$, the acceleration a of the sun toward the center of the galaxy $a = F/m = v^2/r$; thus, by substitution,

$$a = v_{\text{sun}}^2 / r = G M_{\text{galaxy}} / r^2$$

so

$$M_{\text{galaxy}} = r v_{\text{sun}}^2 / G$$

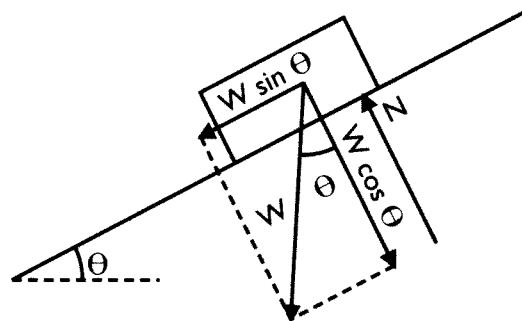
We already have a value for G ; if the orbital velocity of the sun with respect to the rest of the galaxy v_{sun} is approximately 3×10^5 m/sec, and its distance r from the center about 5×10^{20} meters, use this information to calculate the mass of the galaxy. (This should come out to be approximately 6.75×10^{41} kilograms, or approximately 3.39×10^{11} times the mass of our sun.)

17. Nobel Prize winner Linus Pauling said once that if a man now 50 years old had smoked a pack of cigarettes a day since he was 21, he had shortened his life expectancy by 8.5 years. Write a short FORTRAN program which will calculate and print out the number of minutes each cigarette shortened the man's life, if Pauling's claim is correct. (There are 20 cigarettes in a pack.)
18. In the particle accelerator at Fermilab, protons travel in a circle with diameter 2.0 kilometers, with a momentum p of 5.3×10^{-16} kg-m/sec. Write a FORTRAN instruction to compute the orbital *angular momentum* L of such a proton ($L = rp$; r is the magnitude of the position vector—i.e., the radius of the circle).
- 19. The mathematician John von Neumann said that the optimum lecture length is a micro-century; how long is this in minutes?
20. In the National Accelerator (Batavia, Illinois), protons are accelerated through 50,000 trips around a ring of magnets with a radius of 1000 meters. What is the total distance such a proton travels while being accelerated, in meters? in miles? What is the equivalent number of trips around the earth this represents? (The mean radius of the earth is approximately 6.37×10^6 meters.)
21. The earliest recorded attempt to estimate the radius of the earth was made by the Greek mathematician Eratosthenes, who lived in the third century B.C., and was a friend of Archimedes (we will later examine Eratosthenes' *sieve* for finding prime numbers). It was observed that when the sun was at its highest point (summer solstice), at a time when it was directly overhead in Syene (so that the sundials cast no shadow), that in Alexandria, which was 5000 stade (or about 805 km.) north of Syene, a 1-meter pointer on a sundial cast a shadow of 12.6 cm. Use this information, as did Eratosthenes, to calculate the radius of the earth. Assume that rays of the sun striking both cities are parallel. What percent error would this estimation have, given the accurate radius in question 20?
22. How long does it take light to travel from the sun to the earth, if the distance is on the order of 1.5×10^{11} meters, and the speed of light is 3×10^8 m/sec.? How far does light travel in one nanosecond (10^{-9} sec.)?
- 23. We do not yet have the capability to make tests in FORTRAN; that will come in the next chapter. Yet, if you are clever, you should be able to figure out a way, using just the arithmetic operations we have covered, and perhaps a system function like ABS (which takes the absolute value of the value given to it) to find the *smaller* of two values A and B that have either been calculated in your program or input. Notice that if you can find the smaller, you can also determine the larger. Note: do not cheat and use the system functions MIN, MAX, or any other of the functions in that group, such as AMIN0, etc.
- ◆ 24. The maximum height of a projectile with initial velocity V fired at an

angle A radians above the horizontal is $V^2 \sin^2 A / 2g$, where $g = 9.81 \text{ m/s}^2$. What is the maximum height reached by a rock from a volcano ejected at 700 m/s at an angle of 60° ? (Use the SIN function.)

*Note: Throughout the book, we have flagged a few of the problems at the ends of chapters with a diamond (◆) to indicate that they are all related, as stages of a larger problem to be solved later on in the book. This "progressive" problem approach illustrates how the new techniques learned in each chapter contribute to a broader understanding of problem solving in FORTRAN, and how they extend your capabilities to solve more and more complex problems.

25. Static friction. A block on an inclined plane is held in place by static friction as long as the friction is great enough and the angle is not too steep. In the picture shown, a steel block rests on a steel surface inclined at an angle θ . The coefficient of static friction, μ_s , of steel on steel is 0.6. What is the maximum angle θ_{\max} of incline of the plane before the block will begin to slip? [Hints on problem solution: for a block of weight W , the force which will make it slide is the component of the weight directed parallel to the plane, that is, $W \sin \theta$, and at the point of slipping, $W \sin \theta$ must be equal to the force of static friction, that is, $\mu_s N$, where N is the normal force with which the plane pushes on the block. This force is also the same as the force which the block exerts on the plane, or $W \cos \theta$. Thus at the point of slipping,



$$\begin{aligned} W \cos \theta_{\max} &= N \\ W \sin \theta_{\max} &= \mu_s N \end{aligned}$$

and by dividing through, we get $\tan \theta_{\max} = \mu_s$. Thus you should use the ATAN system function (see Appendix B) to determine the magnitude of the maximum angle of inclination of the plane.]

As an extension of this problem, have your program read in other pairs of materials and their coefficient of static friction and determine the maximum angle of incline possible. Use

```

CHARACTER MATA*8, MATB*8
READ*, MATA, MATB, STATIC
.....           {calculation}
PRINT*, 'MAXIMUM ANGLE BETWEEN ', MATA, ' AND ', MATB
PRINT*, 'IS ', THETA, ' RADIANS'
```

You might want to convert the angle in radians to degrees. There are 2π radians in 360° . Here is a table of coefficients:

Materials	μ_s
Steel on ice	0.03
Rope on wood	0.5
Copper on cast iron	1.1
Rubber tire on dry concrete	1.0
Rubber tire on wet concrete	0.7

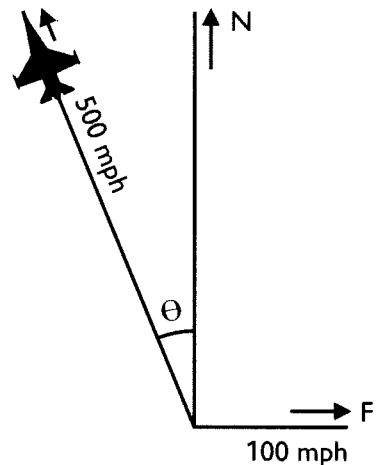
An entry to the READ statement might then be something like:

'Tire', 'Wet Road', 0.7

- 26.** Our airplane can fly at 500 miles per hour, but it encounters a crosswind of 100 miles per hour, blowing due east. At what angle from due north should the plane head in order to travel in a due north direction? What is its net speed in the northward direction?

- 27.** A car travelling at a constant 70 miles per hour passes a motionless policeman on a motorcycle. If the motorcycle accelerates at 1.6 feet/sec^2 , how long does it take the policeman to overtake the car, and how far has he travelled? [Hint: the distance travelled by a body moving at an initial velocity v_0 and accelerating at rate a over time t is $d = v_0 t + \frac{1}{2} a t^2$.]

- 28.** A baseball pitcher throws a ball which is initially horizontal toward the plate 60 feet away. If the ball drops 1 foot by the time it reaches the plate, what was its initial speed?



CHAPTER 3



CONTROL STATEMENTS

The tasks you have learned to perform on a computer thus far could have been done on a calculator or an abacus. To begin to tap into the computing power of your machine, you need to learn how to have your program make tests and act on the results of the tests, vary the order of executing commands, and perform repetitions. The ability to repeat a set of operations is one of the most powerful capabilities of a digital computer. The tools for testing are flexible, allowing you to write programs to solve many *realistic* problems. The world rarely seems to be simply “yes” or “no”—it is more complex.

NASA’s Mission Control
beams essential commands
to spacecraft in flight.

"Everything has two handles, by one of which it ought to be carried and by the other not."

- Epictetus, Enchiridion, Sec. 43

Up to this point, the text has covered the elements of FORTRAN (constants and variables), and the ways they can be manipulated in arithmetic assignment statements. You have also learned simple I/O, and how to construct a simple, sequential program. In this chapter and the next, you will be introduced to FORTRAN control structures that will allow you to perform testing (*selection*), and to construct loops (*repetition*). These are probably the most powerful tools you will acquire for writing complex programs. They make the computer you program into much more than just a big calculator, since they allow the program to ask questions and then *select* different routes (sets of instructions) depending on the answers, and they make possible the construction of processes that will *repeat* until some specified condition is met. Such operations make the computer less like a glorified abacus and more like an artificially intelligent device (since intelligence may be correlated with the ability to make reasoned, justified decisions).

The only major tools you have still to acquire in the future will be arrays (as a foundation for many data structures) and subprograms. Other features of the language will make your programming easier and your results more elegant, but they are not essential to writing programs, as are control structures, arrays, and subprograms. This may well be the single most important chapter in the book.



MAKING DECISIONS

From our discussions of flowcharts, we know that it is necessary, in order to implement any interesting algorithms, to have the ability to ask questions. The FORTRAN constructs that correlate with decision boxes used in flowcharts are various kinds of IF statements. These tools (the various IF decision structures) will be at the basis of all of the more interesting programs we will write in FORTRAN.

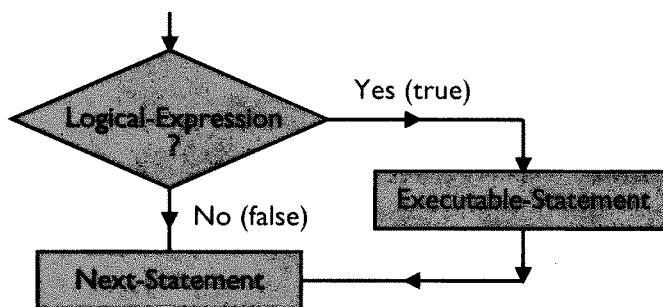
Logical IF Statement

We have seen in our examination of flowcharts that one generally wants to ask a question that has a "yes" or "no" answer, and to execute one operation if the answer is "yes," another if it is "no." Often this takes the form of an operation to be executed *conditionally* (that is, on the *condition* that the answer to the question is "yes"), and to be skipped otherwise. For example, on the

condition that a temperature is above a certain level, we may want to print out a warning; but if it is in the acceptable range, we want to skip the warning. A Logical IF can implement this sort of test, and the statement has the general form:

```
IF (logical-expression) executable-statement
next-statement
```

which can be expressed in our flowchart notation as:



The “logical-expression” in parentheses is some expression (or combination of such expressions) comparing two values (or some combination of such comparisons); the comparison statement may be *true* or *false*. If we examine all of the possible relationships between two values x and y , where x and y may be either constants, variables, or expressions in constants and/or variables, the possible comparisons and their formulations in FORTRAN are as follows (Fortran 90 will add a more mathematical version, so when it is available you can use either form):

Relation	FORTRAN 77	Fortran 90
$x < y$	$x .LT. y$	$x < y$
$x \leq y$	$x .LE. y$	$x \leq y$
$x > y$	$x .GT. y$	$x > y$
$x \geq y$	$x .GE. y$	$x \geq y$
$x = y$	$x .EQ. y$	$x == y$
$x \neq y$	$x .NE. y$	$x /= y$

If you think about it, you will become convinced that these are the *only* possible relationships. As we said, x and y are placeholders that can stand for any expression that could appear on the right-hand side of the “=” in an assignment statement. Any such *relation* between two such expressions will be either true or false. If it is true, the “executable-statement” that follows the parentheses in the IF statement will be executed; that is, it is executed *conditionally*, on the condition that the relation is true. If the relation is false, that statement

is not executed, and the program goes on to execute the "next-statement" following the IF.

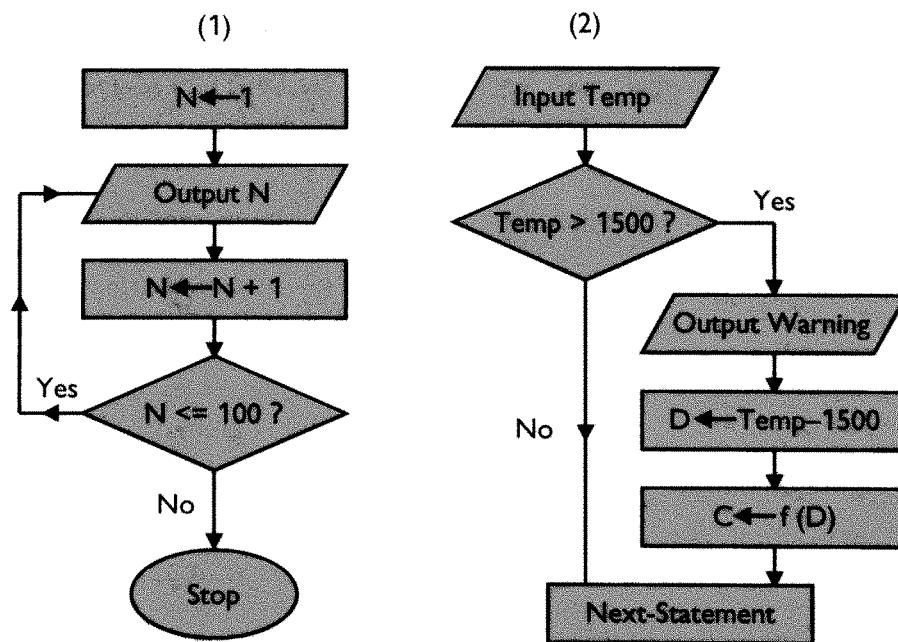
Given this decision structure, we can then implement the high-temperature warning system we mentioned as follows:

```

REAL TEMP
READ*, TEMP
IF (TEMP .GT. 1500.0) PRINT*, 'WARNING! TURN HEAT OFF!'
....{program to continue calculations}

```

However, not all of our test problems resolve themselves easily into a matter of conditionally inserting *one* instruction. We have other possibilities to handle which will require another tool to be used in conjunction with the Logical IF; this tool is the unconditional branch, or transfer of control. Consider the following two problems: (1) we want to program a "loop" that will repeat 100 times, printing out the integers from 1 to 100; (2) we make a test in which, if the condition is true, we want to execute a block of instructions, not just one. For example, in (2), we want to expand our high-temperature control program to test for a dangerous temperature and, if it finds one, to print out a warning, calculate how much into the danger range it is, and determine a correction factor based on this. These two problems are represented by the following flowcharts:



Problem (1) could be implemented if we had the FORTRAN tool of a DO loop, but we do not have that yet (next chapter), so we must temporarily learn

how to manage without it. Problem (2) could be implemented using a Block IF structure, but we have not encountered that yet (or perhaps we have only an older FORTRAN compiler) so that again we must make do with what we have. We shall see later that the adaptability we develop here by "making do" will benefit us in creating other useful structures.

We can clearly see from these two problems that some additional tool is required besides the Logical IF. We need the ability to alter the normal sequential execution of a program, and to jump from one instruction to another which is not the one immediately following it. In a flowchart, this is simply accomplished by a flow line which goes where you want it. In FORTRAN, we need an instruction which will alter the normal sequential execution of instructions; this is the unconditional branch, or GO TO statement.

Transfer of Control—The Unconditional Branch

We have seen that FORTRAN program statements are executed in the order given; this is the sequential structure for problem solving. If you want to alter the order in which these instructions are executed, you need a way to change the normal sequence. The simplest way this can be done is by the use of a *branch* statement that will send the program to some instruction other than the one which is next in the sequence. A branch to a statement, without any testing, is called an *unconditional branch*, and in FORTRAN is implemented through the use of a GO TO statement. The form of a GO TO statement is:

GO TO n

where n is the statement number of the instruction you want the program to branch to (that is, to execute next).

Statement numbers are optional on most FORTRAN instructions, and, to improve program readability, it is best not to label statements if they do not *require* labels. However, some statements *do* require statement numbers—the target of a branch instruction, a FORMAT statement, or the terminal statement of a DO loop (covered in the next chapter). We will discuss the latter two cases at the appropriate time. For now, we see that if we are going to implement an instruction to GO TO some destination, we must have a way of identifying that destination—that is, a statement number.

A statement number must fall in the first 5 columns of a line, and must be a positive integer, so statement numbers may be any value from 1 through 99999. The programmer may choose statement numbers freely within this range, as long as no duplicate numbers are used to label statements in a program unit. Thus, statement numbers may be assigned in any order, and statement number 100 may appear in a program before statement number 3. However, to make your program more readable, it is advisable to choose statement numbers that follow a uniform pattern and increase throughout the program.

As we saw when we were constructing flowcharts, to modify sequential logic structures merely by including a branch to an instruction out of the normal order will only give us the ability to create infinite loops. The simplest infinite loop you could write in FORTRAN would be a statement that branched to itself (such as # **GO TO** #), but this is definitely not a good idea (some FORTRAN compilers will even flag such a statement as an error, and not compile the program). Similarly, you could write a loop to just keep printing out integers infinitely (or until the system stopped you or the program created an integer *overflow*—a condition when the value of a location exceeds its capacity), analogous to our “road-gang” flowchart example. Even a sophisticated compiler will not recognize this as a *syntax* (grammatical) error, and so the program would run—on and on and on. One of your jobs as a good programmer will be to be on the alert that you do not write any such infinite loops.

Thus, as we saw in the case of algorithms, just the ability to change the order in which instructions are executed is not enough; it does give us the ability to repeat, but only to repeat infinitely. However, the unconditional branch when used in conjunction with a *conditional branch* (such as our Logical IF), one that changes the order of instructions based on some test or tests, can be very effective.

The original versions of FORTRAN (I, II, IV, and 66) had been in use for many years when a concern was raised (in 1968, by Edsger Dijkstra) that excessive use of the GO TO statement was “harmful.” It was noticed that programs written with many GO TOs carelessly used were very difficult to follow, and if one were to draw lines on the program representing the flow of control of the program logic, it resembled a plate of randomly thrown spaghetti (thus earning the appellation “spaghetti code”). For this reason, language structures were developed (“block structures”) which greatly reduced the need to use GO TO statements in programs..

Structures of this kind were incorporated into FORTRAN 77. Once we have seen the other control tools that are available in FORTRAN 77, we will see that they greatly reduce the need for GO TO statements. However, many programs still in existence were written in the earlier versions of FORTRAN, and *had* to use GO TOs. You may someday have to read and update such a program, so we will introduce you to the full range of conditional and unconditional branch statements; thus you will be well prepared for any program structures you may encounter, or have available. Further, the judicious use of the GO TO will allow you to construct structures (such as “repeat/until”, “while”, or “exit”) which may not be available in your FORTRAN compiler.

As you become acquainted with the various structures in FORTRAN, you will also come to appreciate why new control structures were added, even though the job could be done with the earliest forms. Each new control structure was added to make things easier for the programmer, and to make programs easier to read and understand. GO TO statements cannot be

eliminated completely in FORTRAN, since they allow us to create structures not otherwise available in the language. However, in accord with the goal of promoting good program style, we will keep their use to a minimum once more elegant structures have been introduced.

Let us now turn to the implementation of our two problems, that of printing out the integers 1 through 100, and that of the high-temperature control program, now that we have the new tool of the unconditional branch. The program to print out numbers from 1 to 100 can be written as follows:

```
***** PROGRAM TO PRINT INTEGERS FROM 1 THROUGH 100 *****
***** INITIALIZE COUNTER N *****
N = 1
10   PRINT *, N
      N = N + 1
      IF (N .LE. 100) GO TO 10
      STOP
      END
```

Note that we have indented only the *body* of the repetition structure (loop), that is, the instruction that is repeated. We have left those instructions that control the repetitions of the loop aligned at the left margin. We will maintain this convention throughout the text when we construct loops of this kind, so that the body of the loop will stand out.

If the conditionally executed statement is a normal command (such as an assignment or an I/O statement), then it is simply inserted in the pattern of executable statements if the relation condition is true, and then the next sequential statement is executed. However, if the conditionally executed statement is a branch statement such as a GO TO, and the condition is true, the branch will be taken, and the "next-statement" will not be executed. Compare the following two uses of the Logical IF:

IF (N . GT. 40) M = N - 5	IF (A . LT. B) GO TO 7
PRINT*, M, N	PRINT *, A, B

In the example on the left, if the condition ($N > 40$) is true, then the statement following the test, the assignment statement $M = N - 5$, will be executed, followed by the execution of the following PRINT* statement. Thus, the assignment statement will simply be inserted conditionally into the execution sequence. If the condition ($N > 40$) is false, the assignment statement will not be executed, and the next statement (PRINT*, M, N) will be executed. However, in the example on the right, if the condition ($A < B$) is true, the statement GO TO 7 will be executed, transferring control to statement number 7, wherever it is in the program, and then following the sequence initiated by execution of that statement; the "next-statement" (PRINT*, A, B) will not be

executed. If the condition ($A < B$) is false, the branch to statement 7 will not be taken, and the PRINT* statement will be executed. This is a very important difference to comprehend.

Our second problem was that of testing for a dangerously high temperature, and executing a group of statements if we did encounter one. If you examine the flowchart we drew earlier for this problem, you will see it involved executing three operations if the temperature was above the cutoff value; but the form of the Logical IF only allows you to conditionally execute *one* statement, not three. In the next section, we will see another form of control structure, the block IF, which will allow you to do this directly. But for now, we will see how to accomplish this using the Logical IF. The Logical IF only lets you execute one statement if its condition is true, and drops through to the next statement otherwise. If we thus put the block of statements we want executed if the temperature *is* in the danger range immediately after the test, then reverse the test, it will work. The pseudocode representation for this would be:

```
If (temperature-not-in-danger-range) branch to #
    print warning
    calculate amount over cutoff temperature
    calculate correction factor
# continue the program
```

The implementation of this in FORTRAN is direct (since we want the block executed if $T > 1500$, we make the test $T \leq 1500$):

```
IF ( TEMP .LE. 1500.0) GO TO 20
PRINT*, 'WARNING! TURN HEAT OFF!'
DIFF = TEMP - 1500.0
CORR = {some expression involving DIFF}
20 {continuation of program}
```

We have seen that the Logical IF *can* be used to execute a block of statements, simply by reversing the form of the test, using a GO TO, and putting the block to be executed after the test. In the next section, the Block IF structure will provide a more elegant form to accomplish the same job.

If you have a problem where a measurement is read in and you need to assess whether it is less than, equal to, or greater than the average of previous measurements, it could be written the following way using Logical IFs:

```
***** READING AND TESTING A MEASUREMENT AGAINST THE AVERAGE *****
***** AVERAGE OF PREVIOUS MEASUREMENTS IS 70 *****
INTEGER MEASUR
READ*, MEASUR
IF (MEASUR .LT. 70) PRINT*, MEASUR, ' IS BELOW AVERAGE'
```

```
IF (MEASUR .EQ. 70) PRINT*, MEASUR, ' IS AVERAGE'
IF (MEASUR .GT. 70) PRINT*, MEASUR, ' IS ABOVE AVERAGE'
```

If the measurement is neither less than or equal to the average, it must be greater than the average; does this mean we can eliminate the third test, and simply put a PRINT statement that indicates the measurement is above average on the last line? Consider this question carefully. [Answer: you cannot omit the last test, since then 'ABOVE AVERAGE' would print in *all* cases.]

Similarly, if we were reading in information on students (their names and grades), we could determine who was on the honor roll, print out the names and grades of those on the honor roll, *count* how many are on the honor roll, and, at the end, determine the average grade for an honor-roll student. We could use the reversed form of the test to allow us to execute these three statements if a student is on the honor roll:

```
COUNT = 0
...
...
IF ( GRADE .LT. 90) GO TO 10
  PRINT*, NAME, GRADE
  COUNT = COUNT + 1
  SUM = SUM + GRADE
10 {next-statement}
```

We have slipped in the notion of a "counter" here, a variable which is initialized to zero, and then increased by 1 every time a condition it is to count is met.

We will now write a whole program which will read in 200 grades for the senior class, print out those on the honor roll, and determine the number of students on the honor roll and their average grade, using the structure we have indicated. We must use GO TOs in two places in this program: one to repeat the loop to read 200 sets of data, and the other to execute the block of statements if a student is on the honor roll. Later, we will be able to replace the first by using a DO loop, and the second by using a Block IF structure. Note the CHARACTER type statement which declares that the variable NAME will contain 15 characters.

```
***** THE HONORS CLASS AND STATISTICS *****
CHARACTER*15 NAME
REAL GRADE, SUM, AVERAG
INTEGER COUNT, N
***** SET COUNTER AND SUM OF HIGH GRADES INITIALLY TO ZERO *****
COUNT = 0
SUM = 0
```

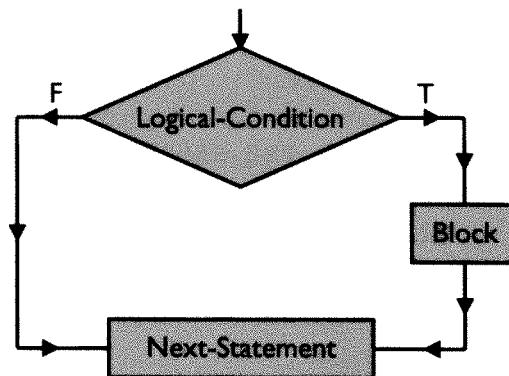
```
***** LOOP TO READ IN 200 GRADES OF SENIOR CLASS *****
*** N IS THE NUMBER OF THE STUDENT WHOSE GRADE IS TO BE READ NEXT ***
N = 1
5      READ*, NAME, GRADE
       IF (GRADE .LT. 90) GO TO 10
       PRINT*, NAME, GRADE
       COUNT = COUNT + 1
       SUM = SUM + GRADE
10   N = N + 1
       IF (N .LE. 200) GO TO 5
       AVERAG = SUM/COUNT
       PRINT*
       PRINT*
       PRINT*, 'THERE ARE ', N, ' STUDENTS ON THE HONOR ROLL'
       PRINT*, 'WITH AN AVERAGE GRADE OF ', AVERAG
END
```

We will now examine a block-structured IF statement, which will allow us to conditionally execute a number of statements (occurring in a “block”) in a more direct manner.

Block IF

FORTTRAN 77 introduced the “Block IF” statement and its variants to improve even more on program readability and the ease of use of FORTTRAN by the programmer. The use of these structures will greatly reduce the need for using GO TOs, and thus reduce “spaghetti code” at the high-level language level. Just keep in mind that these instructions are *actually* still implemented using jumps at the machine-language level, so the program may have code without any GO TOs, but they are in reality built into the structures we do use. The form of the simple Block IF statement is as follows:

```
IF (logical-condition) THEN
  block of executable
  statements
ENDIF
next-statement
```



Thus if the logical condition in parentheses is true, the entire block of statements, which extends from the first statement after the IF/THEN to the ENDIF statement, will be executed in normal order. If the logical condition is false, the entire block of statements will be skipped, and the next statement executed will be that following the ENDIF. The ENDIF is needed to show where the conditional block of statements ends and the program itself resumes.

You can easily see how this Block IF statement could be used in the "honors students statistics" program. We would merely replace the four lines directly following the READ at statement 5 with the following:

```
IF ( GRADE .GE. 90) THEN
    PRINT*, NAME, GRADE
    COUNT = COUNT + 1
    SUM = SUM + GRADE
ENDIF
```

The $N = N + 1$ statement following (the "next-statement" for this block) would no longer need a statement number (it had the number 10).

Many programmers simply adopt the Block IF statement, and abandon the Logical IF altogether. However, the Logical IF may be neater and more compact if only *one* statement is to be conditionally executed. Compare the following two options:

<pre>IF (A .GT. B) THEN B = A ENDIF</pre>	<pre>IF (A .GT. B) B = A</pre>
--	--------------------------------

It should, however, be taken into consideration whether the conditionally-executed statement might ever become a group of statements; in such a case, the Block IF structure on the left would then be the better choice (more readily modified).

IF/THEN/ELSE Structures

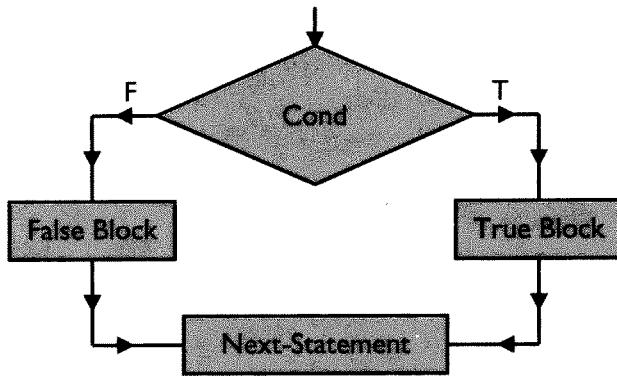
Sometimes the situation arises where there is a block of one or more statements you want to execute if a particular condition is true, and another block you want to execute only if it is false; on no occasion would both blocks be executed, but one of the blocks would always be executed. Imagine the simple case in which you read in two real values A and B, and you want to store the *larger* of the two values in location C (and if they are equal it does not matter which one you store in C). In this case, you want to execute one

instruction if $A > B$, and the other if that is not the case. An IF/THEN/ELSE structure has the form:

```

IF (logical-condition) THEN
    True block
ELSE
    False block
ENDIF
next-statement

```



Thus, in our example, we could write:

```

READ*, A, B
IF ( A .GT. B ) THEN
    C = A
ELSE
    C = B
ENDIF

```

This is a very simple example, and each of the two "blocks" only contains one statement. However, the blocks could contain as many statements as necessary, depending on the nature of the problem being programmed. We will suggest some more complicated examples in the Exercises at the end of the chapter.

ELSEIF Structures

The IF/THEN and IF/THEN/ELSE structures we have discussed are the ones you will use more often in your programs. However, on occasion the nature of a testing process becomes even more complex, and requires a more complex structure for its solution. Situations may occur where there are *several* possible sets of conditions, and a different action is called for under each set of conditions. Examples of such situations occur: in a graduated income tax; in calculating pay when a worker may be paid regular time, time-and-a-half, or double time; in the case where numerical grades must be converted to letter grades; in determining what sound level classification to give to an audio signal; or in any case where multiple conditions may occur.

Let us examine the problem of determining what subjective classification we will give to a particular sound level we encounter, according to the following table:

Sound Intensity Level		Classification
1.0 watts/meter ²		Threshold of pain
10^{-1}	"	Low-flying jet
10^{-2}	"	Rock band
10^{-3}	"	5 p.m. traffic
10^{-4}	"	Vacuum cleaner
10^{-5}	"	Conversation
10^{-6}	"	Office
10^{-7}	"	Quiet office
10^{-8}	"	Library
10^{-9}	"	Bambi
10^{-10}	"	Whisper
10^{-11}	"	Breeze
10^{-12}	"	Hearing threshold

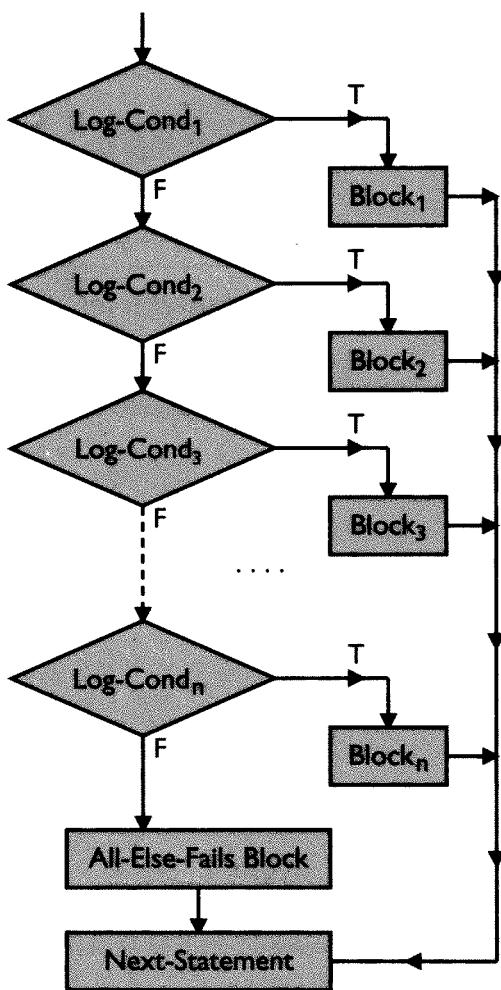
This problem requires a number of successive tests on the sound level, until one of the tests is passed. Our simple Block IF or IF/THEN structures *could* be used, but it would be awkward, and involve considerable "nesting" of IFs that would be difficult to write and to make error-free (try it!), especially since there are so many levels here. But there is a new variant on the Block IF structure that allows us to handle this problem elegantly.

FORTRAN 77 designers recognized the existence of problems, such as our sound level problem, which have many alternate solutions depending on the conditions. Thus, they developed a control structure that would allow successive testing of a number of condition levels, and taking appropriate action as soon as the condition was fulfilled. Instead of having merely one block of instructions as the alternate to the "true" state of the initial condition, they allowed the substitution of an "ELSEIF" clause for the simple "ELSE", so that additional conditions might be tested if the first one failed. The structure is as follows:

```

IF (log-cond1) THEN
    block1
ELSEIF (log-cond2) THEN
    block2
ELSEIF (log-cond3) THEN
    block3
...
...
ELSEIF (log-condn) THEN
    blockn
ELSE
    all-else-fails block
ENDIF
next-statement

```



You can see from this structure and its corresponding flowchart that you can now test a range of conditions successively, taking appropriate action when the relevant condition is met. Thus, to get back to our sound level problem, we could write the segment of the program that reads in and tests a sound intensity, expressed in watts per square meter (meter²) as follows:

```

***** PROGRAM TO CLASSIFY SOUND INTENSITIES *****
REAL SOUND
PRINT*, 'INPUT A SOUND INTENSITY (REAL) IN WATTS/M**2'
READ*, SOUND
IF (SOUND .GE. 1.0) THEN
    PRINT*, 'ABOVE THE THRESHOLD OF PAIN!'
ELSEIF (SOUND.GE. 0.1) THEN
    PRINT*, 'LOW-FLYING JET'
ELSEIF (SOUND .GE. 1.0 E-2) THEN
    PRINT*, 'ROCK BAND'
ELSEIF (SOUND .GE. 1.0 E-3) THEN

```

```

PRINT*, '5 P.M. TRAFFIC'
.....
ELSEIF (SOUND .GE. 1.0 E-11) THEN
    PRINT*, 'BREEZE'
ELSE
    PRINT*, 'I CAN'T HEAR ANYTHING!'
ENDIF
STOP
END

```

We left out some of the intermediate test categories, since it should be clear how they would be filled in to fit the table. Note that our *ranges* after the first one are handled by just one IF condition. For example, the second range is from 0.1 to 1.0 (jet), but since the previous test asked whether SOUND was ≥ 1.0 , we do not have to ask in this test whether SOUND is < 1.0 , since we would not be at this point unless it was. Thus, we only have to ask whether SOUND is ≥ 0.1 for this category.

The "ELSE" clause in an IF/THEN/ELSEIF structure is optional (just as it is in the Block IF/THEN). It is included so that there can be some last "catch-all" action to take if none of the specified conditions are met. However, if the original condition plus those in the ELSEIF clauses cover all possible conditions, or if there is no action to be taken if none of the specified conditions apply, then the ELSE may be omitted. You will get more used to the different block IF structures as we cover more examples using them, and as you begin to write programs of your own which take advantage of them.

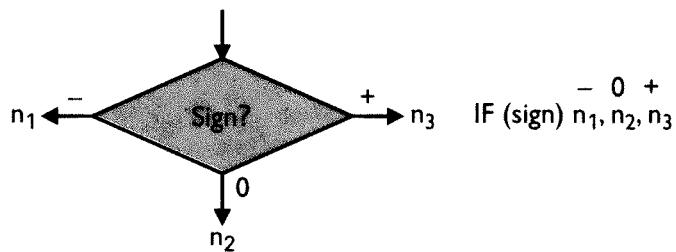
Arithmetic IF

The Arithmetic IF was the first control structure to be implemented in FORTRAN (in the 1957 version), but we present it last—it is certainly not the best, by any means, but it allows you to see the development and improvement in control constructs over the years. The Arithmetic IF is still available in FORTRAN 77, and will be included in the Fortran 90 revision, to maintain FORTRAN's excellent record of upward compatibility. However, the Arithmetic IF has been declared *obsolete* by the Fortran 90 Committee, which means it may be deleted in a future version. All of the tests that can be performed by the later IF statements introduced could have been done only using the Arithmetic IF, but, as you will recognize as we compare them, these later developments are much easier for the human programmer to use, and much easier to follow. Examination of this historical development of constructs reflects the evolution of FORTRAN.

We have seen that an *unconditional branch* (a GO TO statement in FORTRAN) is a transfer of control to a particular statement. A *conditional branch*,

implemented by an IF statement, will perform one or more transfers of control depending on the results of a test made. The GO TO identifies its target statement by a statement number. The Arithmetic IF allows you to select among as many as *three* target statements, depending on the outcome of a test. The simplest test that could be made on an IBM 704, for which the original FORTRAN was designed, was on the *sign* of a quantity; there were underlying machine language instructions on the 704 that could make conditional *jumps* if a sign were negative, zero, or positive. Thus, this feature was mirrored in the original FORTRAN Arithmetic IF statement.

An Arithmetic IF is a three-branch test, which can transfer control to one statement [number] if the sign of the tested quantity is negative, to a second statement [number] if the sign is zero, and to a third if the sign of the quantity is positive. It is also possible that two of these statement numbers might be the same, since you might want, for instance, to branch to the same statement if the quantity were zero or positive, and to a different statement if it were negative. But it would be silly (though syntactically acceptable) to have all three statement numbers the same, since that could be expressed more simply as a GO TO the intended statement number. The Arithmetic IF and its corresponding decision box are thus expressed as follows:



Thus the quantity in parentheses (which can be any arithmetic expression) will determine which branch is taken. If its sign is negative, it will take the branch to statement n_1 , if it is zero the branch to n_2 , and it will go to n_3 if the quantity is positive. You can turn the test(s) for most arithmetic conditions you would use to determine repetitions, or which action is to be taken, into arithmetic expressions, usually ones that compare two quantities. Thus, if you wanted to test a measurement against the average of 70 (as we did earlier using Logical IFs), and print out either "Below Average", "Average", or "Above Average", depending on the measurement, you could do it with one Arithmetic IF:

```

***** READING AND TESTING A MEASUREMENT AGAINST THE AVERAGE *****
READ*, MEASUR
IF (MEASUR - 70) 3, 4, 5
3 PRINT*, MEASUR, ' IS BELOW AVERAGE'
GO TO 6
4 PRINT*, MEASUR, ' IS AVERAGE'

```

```

GO TO 6
5 PRINT*, MEASUR, ' IS ABOVE AVERAGE'
6 .....

```

Note the necessity of the two GO TO statements in this test structure, so that, if a measurement is determined to be below average, the program does not just surge ahead and also print that it is average and above average. It is largely because of this need to use extra GO TO statements, and because it involves its own (conditional) go to's, that the Arithmetic IF is looked down upon by those who want to maintain structured programming and avoid spaghetti code. This example is structured reasonably well, but it does involve GO TO jumps.

We can also construct a simple loop using the Arithmetic IF (or a Logical IF, as we have seen). Reconsider the problem of printing out the first 100 integers, and then stopping, for which we already drew a flowchart. We start a variable N off at one, print it, then increase N by 1 and repeat until we have printed the first 100 numbers. Whether to continue or stop the loop is determined by a decision box in which we asked the question, "IS N <= 100?". If the answer was "yes," we branched back to the Print statement and repeated from there; if it was "no," we stopped. We can readily turn the test on N into an Arithmetic IF, as we can see from this program.

```

***** PRINT OUT THE NUMBERS FROM 1 TO 100 *****
N = 1
10   PRINT *, N
      N = N + 1
      IF ( N - 100 ) 10, 10, 20
20   STOP

```

This is a little awkward, since you must turn the natural way of phrasing the test, that is, whether N is less than or equal to 100, into a test on some arithmetic difference, but one could get used to this eventually. However, the Logical IF and Block IFs that FORTRAN developed are more natural ways of making such tests. The Arithmetic IF test is best suited for a three-way condition such as our example of the measurement.

Logical Connectives (AND, OR, NOT, EQV and NEQV)

So far in logical tests we have only used simple relations, such as (A .GT. B); if we also wanted to test whether (A .GT. C), it would have to have gone in another, embedded IF test. However, there are logical *connectives* that allow us to make the logical expression which is tested in an IF a *compound* condition. These connectives are .AND., .OR., .NOT., .EQV., and .NEQV., and their

interpretation follows closely our natural understanding of the expressions in English. If we want to test whether two conditions are *both* true, we could write:

```
IF (A .GT. B) THEN          IF (A.GT.B .AND. A.GT.C) THEN
  IF (A .GT. C) THEN        PRINT*, A
    PRINT*, A
  ENDIF
ENDIF
```

It is easy to see that the right expression is simpler and more elegant, and it also is easier to follow the test it performs. The "and" is a strong connective, and claims that both of the parts that it connects are true. Thus, if the letters x and y stand for any two logical values, that is, which have the values .TRUE. or .FALSE. (as they are expressed in FORTRAN), then the following table represents the way in which the expression x .AND. y depends on the individual truth values of x and y. Such a table is called a "truth table," and you may already be familiar with it from some other context, such as a logic or a digital electronics course. It is a convenient shorthand description of how the value of a compound "and" expression is determined:

x	y	x .AND. y
T	T	T
T	F	F
F	T	F
F	F	F

It is clear that the table represents all possible combinations of truth values for x and y. Thus, if the component truth values are known, the value of the compound expression is also known.

The connective "or" is a weaker connective. If we say "x or y," then if either or both of the component parts are true, the compound expression is true; only if both parts are false will the new, more complex statement be false. This is also called the "inclusive or," since it allows for the possibility that both parts are true, and still the whole is true; later, we will look at the "exclusive or," which is false if both parts are true. This dependence of the value of the compound "or" statement on the values of its component parts is represented by the truth table for "OR":

x	y	x .OR. y
T	T	T
T	F	T
F	T	T
F	F	F

You can also *negate* the truth value of an expression by using the logical modifier "not." Thus, if "Xenon is a rare gas" is true, then "Xenon is not a rare gas" or "It is not the case that Xenon is a rare gas" is false. Similarly, you can negate the logical value of an expression in a conditional statement by using the modifier .NOT. The truth table for .NOT. is:

x	.NOT. x
T	F
F	T

There are two other compound relationships you can test in FORTRAN—whether two logical values are "equivalent" (.EQV.), that is, whether they have the same truth values; or whether they are "nonequivalent" (.NEQV.), that is, have different truth values. The truth table for these connectives is:

x	y	x .EQV. y	x .NEQV. y
T	T	T	F
T	F	F	T
F	T	F	T
F	F	T	F

Note that NEQV accomplishes the "exclusive or" test we discussed.

Thus, if you know the truth values of component parts of a logical

expression, you can determine the truth value of as complex a compound expression as you like. A complex expression can be made up of any “well-formed” expressions, where any of the following constitute well-formed expressions: x , $.NOT.x$, $x .OR. y$, $x .AND. y$, $x .EQV. y$, $x .NEQV. y$. Thus, you can form a legal complex expression such as $((a .OR. b) .AND. .NOT.(a .AND. b))$ out of logical expressions a and b , but $(.NOT.a .EQV. .OR.b)$ is *not* a well-formed, legal logical expression.

DeMorgan's Laws

The logician and mathematician Augustus DeMorgan, who was the mathematical mentor for Ada, Lady Lovelace (the “first programmer,” and after whom the new Department of Defense language for embedded systems is named), proved the following relationships among logical compound expressions involving *and*, *or*, and *not*:

$$\text{not } (x \text{ or } y) = (\text{not } x) \text{ and } (\text{not } y)$$

$$\text{not } (x \text{ and } y) = (\text{not } x) \text{ or } (\text{not } y)$$

The logical truth tables which we can construct, given what we have learned already, confirm these relationships. Notice that each column of the truth table can be evaluated from some combination of the previous columns by applying the simple tables for *.AND.*, *.OR.*, and *.NOT.* that were given previously.

x	y	$x .OR. y$	$.NOT.(x .OR. y)$	$.NOT.x$	$.NOT.y$	$.NOT.x .AND. .NOT.y$
T	T	T	F	F	F	F
T	F	T	F	F	T	F
F	T	T	F	T	F	F
F	F	F	T	T	T	T

x	y	$x .AND. y$	$.NOT.(x .AND. y)$	$.NOT.x$	$.NOT.y$	$.NOT.x .OR. .NOT.y$
T	T	T	F	F	F	F
T	F	F	T	F	T	T
F	T	F	T	T	F	T
F	F	F	T	T	T	T

Order of Operations in Logical Expressions

In complicated arithmetic expressions, FORTRAN has a default hierarchy in which the operations are evaluated if there are no parentheses to direct which operations to perform first. Similarly, in a complex logical expression there is also a hierarchy in which the logical operations are performed.

Hierarchy of Logical Operations

Relational expression

.NOT.

.AND.

.OR.

.EQV. or .NEQV.

Thus, if the expression (A .GE. B .OR. A. GE. C .AND..NOT.C.EQ.D) is evaluated for arithmetic variables A, B, C, and D, the relational expressions (A.GE.B, A.GE.C, C.EQ.D) will be evaluated first, then the .NOT. C.EQ.D, then the (A.GE.C.AND..NOT.C.EQ.D), and the ".OR." last. The fully parenthesized equivalent expression indicating the order of evaluation would be:

((A .GE. B) .OR. ((A .GE. C) .AND. (.NOT. (C. EQ. D))))

We can also see that there is a simpler way to write the compound test which PRINTs a message either if A is greater than B *or* if A is greater than C:

<pre>IF (A.GT.B) THEN PRINT*, 'MESSAGE' ELSEIF (A.GT.C) THEN PRINT*, 'MESSAGE' ENDIF</pre>	<pre>IF (A.GT.B.OR.A.GT.C) THEN PRINT*, 'MESSAGE' ENDIF</pre>
--	---

This facility to construct compound tests will greatly increase our flexibility in writing programs.



CONSTRUCTING SIMPLE REPETITIONS WITH THESE ELEMENTS

We have already set up simple *loops* using Arithmetic IFs or Logical IFs, but let us take a closer look at such repetitive structures. There are a number of different ways that we may want to set up repetitions. One type of repetition is that which executes a set of statements a definite number of times, such as the loops we wrote to read in 200 student grade records. Such a repetition structure will later be implemented using a FORTRAN "DO loop," but for now we can set up loops using branch and test statements we have available so far.

Loops That Repeat N Times

We may have a simple instruction or set of instructions that we want repeated a definite number of times; call the number of desired repetitions NUMBER. The general approach is to set up a loop counter (we shall call it LOOP), start it off at 1, and then let it keep track of which repetition of the block of instructions we are currently executing. Each time a repetition is completed, the loop counter will be increased by 1, and we will then test it to see if we have completed NUMBER repetitions yet. If the count of the loop repetition to be performed next is less than or equal to NUMBER (the number of repetitions we want), we will go back to the beginning of the loop and repeat the instructions again. If it is greater than NUMBER, we have completed our NUMBER of repetitions, and will go on to the next part of the program. The general structure of such a loop pattern to repeat NUMBER times is as follows:

Set counter to 1	LOOP = 1
# Statements composing body of loop	# {body of loop}
Increase loop counter	LOOP = LOOP + 1
Test loop counter	IF (LOOP .LE. NUMBER) GO TO #
Next-statement	{next-statement}

A simple example of such a loop would be one to print out 100 times "I will not chew gum in class" for your cranky teacher who caught you chewing gum and has assigned this as punishment. You, being a clever programmer, decide to have the computer do most of the work for you.

```
***** ASSIGNMENT FOR MR. SCROOGE *****
***** WRITE SENTENCE 100 TIMES (NUMBER = 100) *****
      LOOP = 1
20 PRINT*, 'I WILL NOT CHEW GUM IN CLASS'
      LOOP = LOOP + 1
```

```

IF (LOOP .LE. 100) GO TO 20
STOP
END

```

Notice that the "body" of this loop only included one statement to be repeated, the PRINT statement, but it could have had as many statements as you like. In this example, the loop counter value is not used in the body of the loop itself, but it could be, as in the examples we programmed earlier to print out the integers from 1 to 100. In those examples, the counter was called N, and it was used in the body of the loop to indicate what was to be printed, as well as to control the number of repetitions of the loop.

Another variant on a loop that will repeat a definite number of times is one where the loop counter does not necessarily begin at 1 and increase by 1 every time. For example, imagine that you wanted to modify your number-printing program to print out the first 100 even numbers. In this case, your loop counter would begin at 2, it would increase by 2 every time, and it would stop after it had printed out the number 200. We know that this is a loop that will repeat exactly 100 times, but the upper limit on the loop counter is not the number of repetitions in this case.

```
***** OUTPUT EVEN NUMBERS FROM 2 THROUGH 200 *****
NEVEN = 2
30    PRINT*, NEVEN
      NEVEN = NEVEN + 2
      IF (NEVEN .LE. 200) GO TO 30
      STOP
      END
```

There is also no necessity that such loops should always be controlled by counters that *increase* every time. A loop controlled by a loop counter that began at 100, decreased by 1 every time, and stopped after the loop counter had completed the repetition associated with a loop counter value of 1, would also execute exactly 100 times. A similar loop might be used, for instance, to print out all the lyrics (should you ever want to do so) of the old song, "99 Bottles of Beer on the Wall." The program might be written as follows:

```
***** PRINT OUT FAMOUS OLD SONG *****
NUMB = 99
55    PRINT*, NUMB, ' BOTTLES OF BEER ON THE WALL,', NUMB,
      $   ' BOTTLES OF BEER,'
      PRINT*, ' IF ONE OF THOSE BOTTLES SHOULD HAPPEN TO FALL,', '
      &   NUMB - 1, ' BOTTLES OF BEER ON THE WALL'
      NUMB = NUMB - 1
      IF (NUMB .GE. 1) GO TO 55
      STOP
      END
```

Loops That Repeat Until Some Condition Is Met (Posttest Loops)

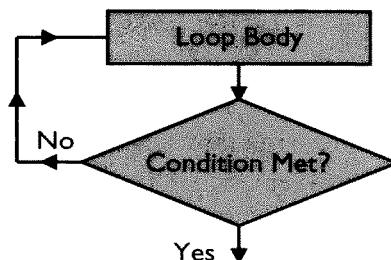
Another type of repetitive loop that you can construct now with your branch and test statements is one that will “Repeat Until” some terminating condition is met. Some languages, such as Pascal, implement such a structure directly in the language; we can simulate such a structure using our IFs and GO TOs. The general form of such a loop would be:

```
Repeat
    Body of Loop
    Until (condition)
```

Such a loop is referred to as a “posttest” loop, since the body of the loop is repeated each time, and the test for termination is made *after* the repetition.

To set up our FORTRAN equivalent of a “Repeat/Until” loop, we will introduce a new statement called the CONTINUE statement. The CONTINUE instruction in FORTRAN is a “do-nothing” statement that effectively says to continue on to the next executable instruction in the sequence. It is occasionally helpful to use such a statement as a neutral place-marker at the beginning or end of a loop structure (and it takes less time than an alternative do-nothing statement such as “*A = A*”). We will use the CONTINUE statement in that way to mark the beginning of our “Repeat/Until” block simulated in FORTRAN.

We will precede the pseudocode representations for such loops with a flowchart representation:



```
# CONTINUE
      Body of loop
      IF (not-condition) GO TO #
```

```
REPEAT
      Body of loop
      UNTIL (condition)
```

It is clear that the body of the loop must change *something* each time, so that a terminal condition that may not be met initially will be met at some time, or else we will have constructed an infinite loop. An example might be that you are going to read in information on elements and their wavelengths, and print out the names of only the first 10 that meet the ultraviolet qualification (roughly 5×10^{-7} to 10^{-8} meter). We will assume that there are at least 10

entries which will meet the ultraviolet condition, or else we would have an infinite loop that would be terminated by the monitor or the user when we ran out of data to read in. The data is read in from a sequential disk file that was created earlier.

We will use a counter (discussed briefly already, to be covered in more detail in the next section) to keep track of how many candidates pass the ultraviolet test. We will call the counter NUV, and initialize it to zero, since before we have read in any data we do not have any entries that have been tested.

```
***** ULTRAVIOLET RANGE ELEMENTS *****
CHARACTER NAME*15
INTEGER NUV
REAL LENGTH
***** SET COUNTER NUV FOR # IN U-V RANGE TO ZERO *****
NUV = 0
***** BEGIN READING IN CANDIDATE DATA - NAME AND WAVELENGTH (M) *****
OPEN (4, FILE = 'ELEMENTS', STATUS = 'OLD')

      PRINT*, ' LIST OF THOSE ELEMENTS IN ULTRAVIOLET RANGE'
30   CONTINUE
      3) READ (4,*) NAME, LENGTH
      1) IF (LENGTH .LE. 5. E-7 .AND. LENGTH .GE. 1.0 E-8) THEN
          PRINT*, NAME, LENGTH
          NUV = NUV + 1
      ENDIF
      2) IF (NUV .LT. 10) GO TO 30
      1) STOP
      END
```

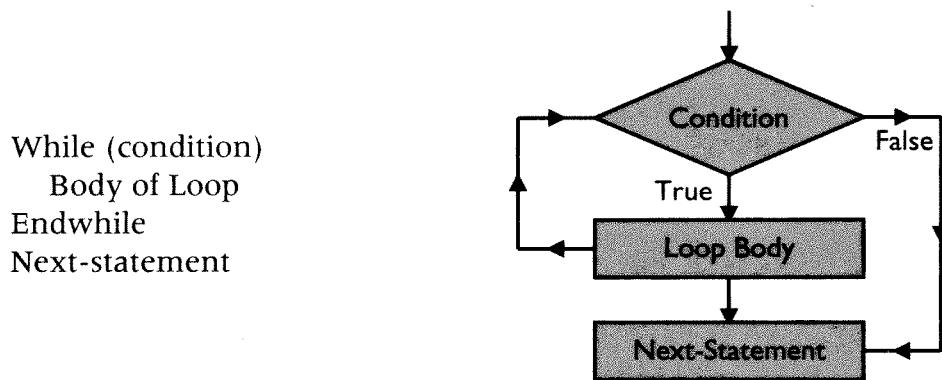
Notice that there *is* a value (NUV) that changes within the loop body, so that the terminal condition can eventually be met (as long as there are at least 10 candidates that meet the wavelength condition). Also notice that we have used spaces (since FORTRAN ignores blanks) to indent the body of the "Repeat/Until" loop, and also to indent the block conditionally executed in the IF/THEN test. Such indentations make your program structure easier to follow.

Loops that Continue WHILE Some Condition Holds (Pretest Loops)

Another form of repetition to be considered is a loop body that is repeated "While" (or "as long as") some condition holds. Some languages (such as Pascal) have a built-in program structure which directly implements this

repetitive structure; some FORTRAN 77 compilers also implement such a structure, but it is not part of the Standard. Thus, we will discuss here how to "simulate" such a repetition structure using branches and tests, and discuss later, in their proper place, the non-Standard WHILE structures, in case they are available on your computer.

The general form of a "While loop" is:



Such a loop is called a "pretest" loop, since the condition is tested each time *before* the loop body is executed. If the condition is still true, the loop executes; if it is false, you drop out of the While structure to the next instruction.

We can write FORTRAN instructions which will *simulate* a While loop as follows:

FORTRAN Simulation

```
# IF (condition) THEN
  Body of loop
  GO TO #
ENDIF
Next-statement
```

While Structure

```
WHILE (condition)
  Body of loop
ENDWHILE
Next-statement
```

The condition is tested *before* the loop body is executed. If the condition is true, then the body of the loop will be executed, and another test will be made; if the condition is false, the entire loop body will be skipped, and the next-statement following the loop will be executed.

An interesting way to use this simulated "While" structure would be that of having a "flag" or "sentinel" value to mark the end of good data to be processed. Say you wrote a weight-processing program that may, for example, average a set of weights of rock samples, find the highest weight, count how many fall in a certain range, and so on. You would probably like to be able to use that program many times for many different sets of weights from different collections, or just generally for any collection whose size you did not want to bother to count ahead of time. If a size is built into the program, then it will

only process a set of that many weights, and so run out of data for smaller sets or not include all the weights from a larger set.

It would be useful to have a way of *marking* the end of the data to be processed, without having to count it or to build a size into the program. The way to accomplish this is to put a “flag” value at the end of the good data, as a signal to stop reading in data and go on to the next step of the processing. If the chemist who will be using the weight-processing program does not give negative weights to anything (a reasonable assumption), then you can use a negative weight as the flag value at the end of the good weight data. That means that the weight values input to the program might be something like:

```
99.5
65.0
73.0
.....
.....
94.0
-40.0
```

The program should then keep reading weights until it hits the negative weight, and it should go through the processing steps *while* (that is, as long as) the weights read in are non-negative. Such a program should read in the very first weight before the loop begins, so that the “while” condition has something to test, and then continue the Reads in the loop, after each stage of processing. The skeleton outline of the processing segment of such a program should look like the following:

```
READ*, WEIGHT
44 IF (WEIGHT .GE. 0.) THEN [While]
    {processing segment for weights}
    READ*, WEIGHT
    GO TO 44
ENDIF [Endwhile]
```

We will actually implement such loops in future examples.



We have already used counters in several program examples—for instance, the program which kept track of how many elements fell in the ultraviolet range. A *counter* is a very important element of programming, often used for

keeping track of loop repetitions, so that the loop terminates when it should. It may also be used for actual "counting" of how many values the program encounters satisfy some condition or set of conditions. Using such values to keep track of what is going on, and the testing of such values, lies at the heart of many interesting and complex programs. Notice that true "counters" are whole number values, and so should be stored in integer locations in FORTRAN. We will add one more program example using counting here, to increase your familiarity with the concept.

Your program is to examine all of the integers from 1 to 99 and *count* how many of them have the sum of their digits add up to 9, and are also divisible by 9. To "extract" the digits from each integer (we will consider each number to have two digits, even though some of these are leading zeros) we will use integer division to get the tens digit of the number, and the remainder approach to get the units digit. We will effectively be using a second counter in this program, one to control the loop and generate the integers.

```
***** HOW MANY INTEGERS 1-99 HAVE THEIR DIGITS SUM TO 9 *****
***** AND ARE ALSO EXACTLY DIVISIBLE BY 9? *****
***** USE A LOOP COUNTER N TO GENERATE INTEGERS 1 - 99 *****
      INTEGER N, UNITS, TENS, COUNT
      COUNT = 0
      N = 1
***** USE INTEGER DIVISION AND REMAINDER FOR DIGITS *****
25  CONTINUE
      TENS = N/10
      *          UNITS IS N MOD 10          *
      UNITS = N - TENS*10
      IF (TENS + UNITS .EQ. 9 .AND. MOD(N,9).EQ.0) THEN
          COUNT = COUNT + 1
      ENDIF
      N = N + 1
      IF (N .LE. 99) GO TO 25
      PRINT*, 'THERE ARE ', COUNT, ' INTEGERS FROM 1 TO 99',
      & ' WHOSE DIGITS SUM TO 9 AND ARE DIVISIBLE BY 9'
      STOP
END
```

Notice that we have taken the "hard way" to solve the problem, but it did allow us to examine a method for extracting the digits of an integer value, which may be useful in the future. However, for this problem, once the tens digit had been set, there is only one units digit that would work to add up to 9, and it could have been simply computed as $(9 - \text{TENS})$. Also note that *all* numbers whose digits summed to 9 turned out to be divisible by 9.

You will find many useful applications for counters in your programs, and we will explore many of these in the book.



ACCUMULATION

Another common application of program loops is to *accumulate* a sum or a product of terms generated or processed. If you want to average grades, you must first sum them up. If you want to know what your running bank balance is, you must perform an accumulation as you do regularly in your checkbook. If you want to find the factorial of a number, you must accumulate a product of integers. And so on.

If you are accumulating a sum, as in averaging, you must first “clear” the accumulator location to zero before beginning to add values into it. You do this when you use your calculator. If you want to add up what you spent in the grocery store on your calculator, which you had just loaned to your friend, you first hit the “CLEAR” button before adding in any values, to get rid of whatever value had been saved in the display and memory from the previous use. To accumulate a product, you initialize to 1. Note that in accumulating products, as in the factorial case, these values can get very large very fast, and may lead to an overflow; the largest factorial that could be accommodated on a 32-bit word machine would be $12!$, or 447001600, since the next value, $13!$ or 6227020800, exceeds the largest integer which can be represented on a 32-bit machine (that is, 2147483647).

As a simple example, we will add up the integers from 1 to 100. We are already familiar with setting up a loop to generate (and print) those integers. All we will do is modify this loop to accumulate the sum of these integers as well. Before proceeding, recall the nature of these earlier programs. How can you accumulate a sum of these integers? Is there any variable already used in the program that you can use for the sum? No. You cannot use N, the counter that takes on the successive integer values, to also keep track of the sum, since they are distinctly different values after the first term. Thus you clearly need a new variable, one which has only one purpose—that of containing the partial sums as you go along. This SUM variable must be initialized to zero, and then each new integer value will be added into it as it is generated.

```
***** SUM OF THE INTEGERS FROM 1 TO 100 *****
      INTEGER N, SUM
*****      INITIALIZE ACCUMULATOR TO ZERO      *****
      SUM = 0
      N = 1
20      SUM = SUM + N
      N = N + 1
```

```

IF (N .LE. 100) GO TO 20
PRINT*, 'THE SUM OF THE INTEGERS FROM 1 TO 100 IS ', SUM
STOP
END

```

You can also consider how to expand the weight-processing loop to average weights; this will be suggested as an assignment at the end of the chapter. You will find the skill of taking sums or products a very useful one as you continue developing various programs.

EXCHANGING VALUES

Many occasions will arise where you are asked to *interchange* the values in two variables. This will come up in sorting procedures and many other applications. Thus it is a good tool to put into your collection of skills at this point. The method for doing this could easily have been covered in the previous chapter, but the need to perform an interchange usually arises as a result of some test that has been made (such as, if the first of two values which have been input is not the larger, then interchange the values), so it seemed to be more appropriate to address the problem here.

Two values have been read into the integer variables M and N. Since they have been read in, they could be any pair of values. Because the program needs the larger of the two values stored in location M, you must make the test, and if M is not the larger, you must interchange them.

```

IF (M .LT. N) THEN
    {perform interchange}
ENDIF

```

It is your job to switch the two values, so that the one originally stored in M is stored in N, and the value originally stored in N is now stored in M. Your first impulse might be to write:

	<i>M</i>	<i>N</i>
M = N	\$	8
N = M	8	\$
		8

but this will not work. Assume two values initially for M and N, and then look at the results of your two assignment statements. The original value of N is stored into M all right, but then the "new" value of M is stored into N, thus leaving the *same* value in both locations. This example makes it clear that we need some way to temporarily *save* the original value of M while the inter-

change takes place; do this in location MTEMP:

MTEMP = M	M	N	MTEMP
M = N	\$	8	5
N = MTEMP	8	\$	
		5	

An examination of the values and how they are changed by the instructions indicates how the program segment succeeds in accomplishing its task. This method will work for interchanging all sorts of values, including character values, logicals, double precision, and records.

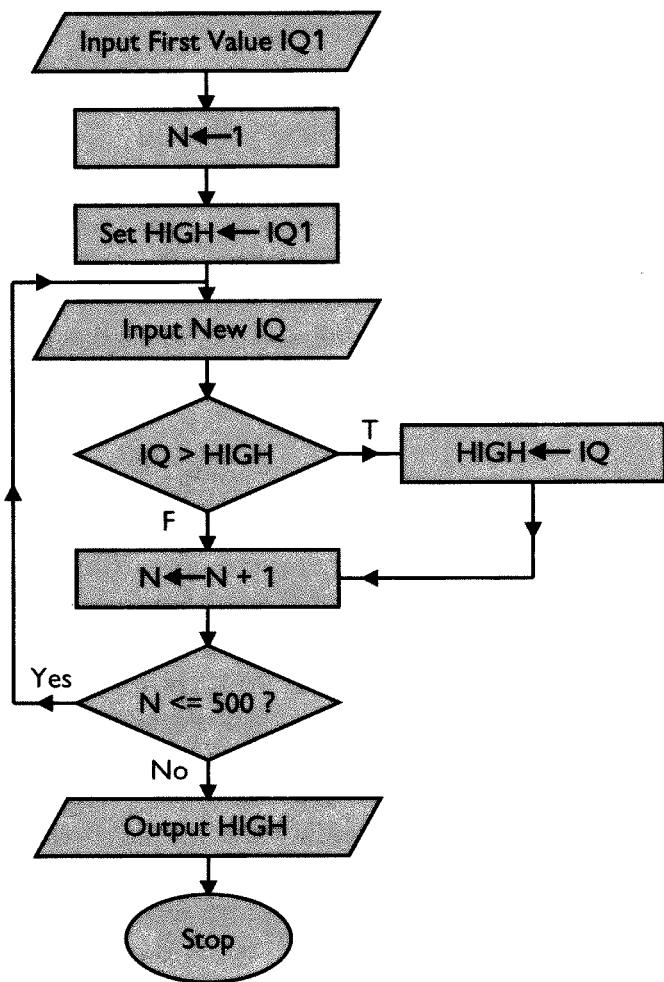


SOME SIMPLE (BUT NOT TRIVIAL) PROGRAMS

Write a program which will read in 500 integer values representing IQs and print out the highest IQ in the set. To do this, you need a variable to keep track of the highest IQ value you have seen so far. You should initialize this variable to some impossibly low IQ value (say -35), or (preferably) to the first value you read in. Then, as each new value is read in, compare it to the value in the variable that keeps track of the highest. If the new value is greater than the one you have saved in HIGH, then replace the value of HIGH with this new IQ; if the value read in is less than or equal to HIGH, leave things alone.

Assuming the loop structure is already set up to read in the IQs (since we have already done many similar examples), we will concentrate on developing the part of a flowchart to handle this process of determining the highest value in the whole set. (See flowchart on the next page.)

```
***** FINDING THE HIGHEST IQ IN A SET OF 500 *****
INTEGER IQ, IQ1, HIGH, N
***** INITIALIZE VALUE OF HIGHEST TO FIRST ONE READ *****
PRINT*, 'ENTER AN IQ VALUE'
READ*, IQ1
HIGH = IQ1
N = 1
88 CONTINUE
    PRINT*, 'ENTER AN IQ VALUE'
    READ*, IQ
    IF (IQ .GT. HIGH) HIGH = IQ
    N = N + 1
    IF (N .LE. 500) GO TO 88
    PRINT*, 'THE HIGHEST I.Q. IN THE SET IS ', HIGH
    STOP
END
```



You will find that the ability to find the largest (or smallest) value in a group of values will find considerable use as you write new programs.

Write a program which will count how many basketball candidates are 6 feet tall or taller and have a field goal percentage of at least 60%. Read in data on 100 candidates. We will assume that this data has already been stored as a sequential file on disk (file name 'JOCKS'), so that all we have to do is OPEN it and READ from it; there is no need to prompt the user each time to enter information. Note that your tryout list will include only the names of those who pass *two* tests.

```

***** BASKETBALL TRYOUT ROSTER *****
***** ONLY THOSE AT LEAST SIX FEET TALL AND WITH A *****
***** FIELD GOAL PERCENTAGE OF AT LEAST 60% *****

INTEGER COUNT, N
REAL HEIGHT, FGOAL
CHARACTER NAME*15
OPEN (3, FILE = 'JOCKS', STATUS = 'OLD')

```

```

COUNT = 0
N = 1
50 CONTINUE
    READ (3,*) NAME, HEIGHT, FGOAL
    IF (HEIGHT .GE. 72.0 .AND. FGOAL .GE. 60.0) THEN
        COUNT = COUNT + 1
        PRINT*, NAME, HEIGHT, FGOAL
    ENDIF
    N = N + 1
    IF (N .LE. 100) GO TO 50
    PRINT*, 'THERE ARE ', COUNT, ' GOOD CANDIDATES'
    STOP
END

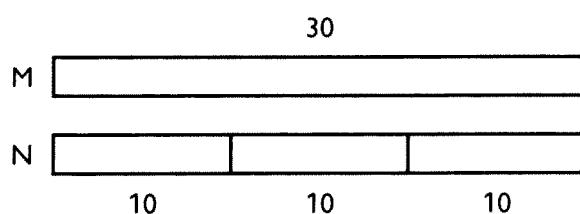
```

Greatest Common Divisor

A paradigm algorithm is the method developed by Euclid for finding the greatest common divisor, or the greatest common measure, of two whole numbers (Euclid's *Elements*, Book VII, Proposition 2, and Book X, Proposition 3). The algorithm for finding the greatest common divisor of any two integers (call them M and N) may be expressed as follows:

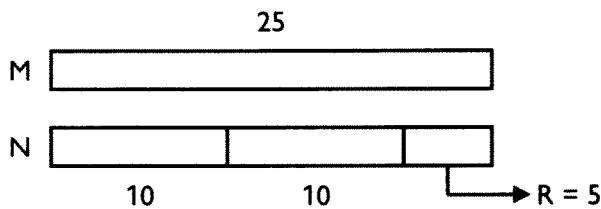
Make sure the value you are calling M is the larger value. (1) Divide M by N and calculate the remainder R (if N does not measure M exactly—that is, divide into M exactly—then R is the amount left over after as many multiples of N as possible are removed from M). If the remainder is zero (0), then the procedure is finished, and the value currently called N is the Greatest Common Divisor (GCD) of the two original values (output it and stop). If R is not zero, then replace the value of M by N and that of N by R, and repeat the procedure from (1) again.

A few diagrams and utilizing the notion of greatest common *measure* may help in understanding this algorithm. Let us suppose that M and N represent two physical (integer-valued) lengths. Then the GCD is the longest possible measure, or ruler, that will measure them both exactly. Examine the simple case where M is an exact multiple of N; for example, M is 30 and N is 10. We then see that M can be exactly measured by N, laid off against it three times. We can see this from the diagram:

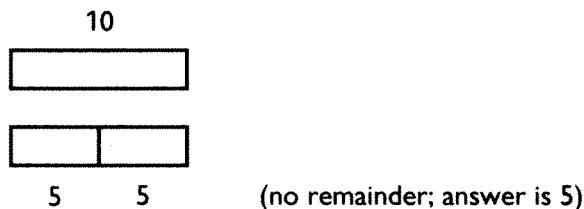


Since N obviously can measure itself, and we see it also measures M exactly, it is the greatest common measure of both lengths.

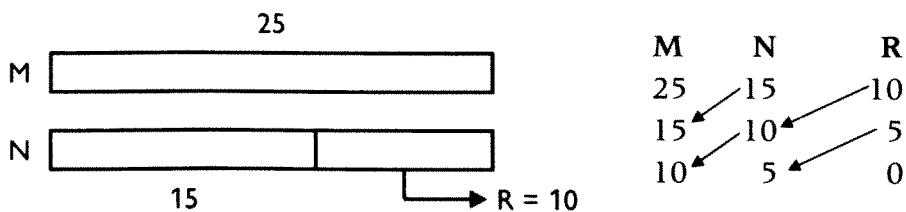
Now look at a case where M is *not* an exact multiple of N ; for example, M is 25 and N is 10. When we try to measure M by N , we see that something (R , of length 5) is left over.

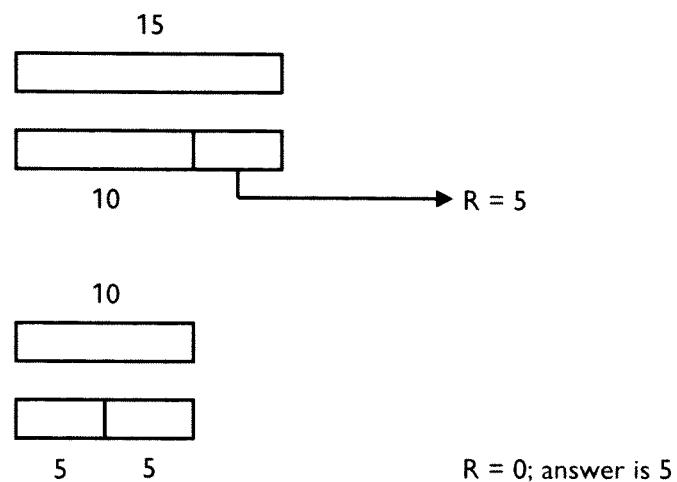


However, if we can find something that measures N (10) and R (5) exactly, it will also measure M , since M is made up of two N 's and an R . We already have a procedure to find the GCD of two values, the algorithm beginning at (1), as long as the larger value is called M and the smaller value N . Thus we put N into M and R into N (in that order), and repeat from (1). We wish to reduce the problem to one we *can* solve, that is, one where the remainder comes out to be zero. This time, we measure 10 by 5 and find out it goes exactly, so the answer is 5.

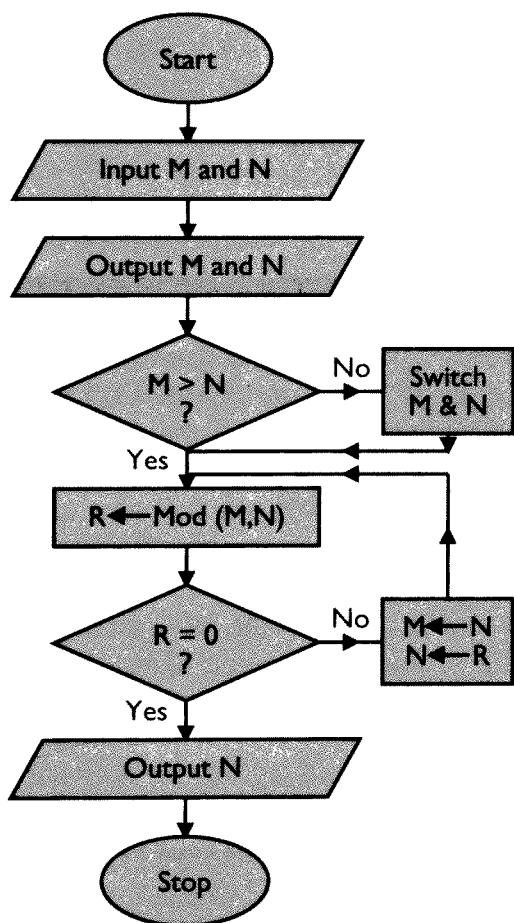


Let us examine a slightly more complicated case, and look at how the values of M and N change as we proceed. Suppose that M is 25 and N is 15. We try our first measure, and find that the remainder is 10. Thus, we need to find a value that will exactly measure 15 and 10, so we put these values into M and N , respectively, and repeat from (1). This time we have a remainder of 5. Thus we still have not finished. We put 10 into M and 5 into N , repeat the procedure, and get a remainder of 0. Thus our last value of N (that is, 5) is the answer we sought. 5 is the GCD of 25 and 15, because it divides exactly into 10, and 15 is made up of 10 and 5, and 25 of two 10s and a 5.





We see that this procedure will work for *any* two integers.



```

INTEGER M, N, R
READ*, M, N
PRINT*, 'GCD OF ', M, ' & ', N
IF (M.LT.N) THEN
  MTEMP = N
  M = N
  N = MTEMP
ENDIF
5 CONTINUE
R = MOD(M,N)
IF (R.EQ.0) THEN
  PRINT*, ' IS ', N
  STOP
ENDIF
M = N
N = R
IF (R.NE.0) GO TO 5
END
  
```

The output from various runs of this program would look like:

```

GCD OF 15 & 25
IS 5
GCD OF 629 & 1591
IS 37
GCD OF 30 & 99
IS 3
GCD OF 17 & 59
IS 1
GCD OF 728 & 1776
IS 8

```

The program could be modified to ask the user at the end whether he or she wants to enter another pair of numbers and, if the user answers 'YES', repeat from the READ.

Revolving Charge Account

Another problem we could solve at this point is that of how long it would take to pay off a charge made on your Faster Charge account if you spent a certain amount, and paid it off at the minimum payment rate (which is 1/30 of the balance, truncated to dollar amount, or \$10, whichever is greater). Each month you must make a payment, and each month you are charged 1.5% on the remaining balance. We will set up a variable BAL for the unpaid balance, PAY for payment made, INT for the interest charged, and M as the month counter to keep track of how long we are paying this off; we will also use a variable COST to keep track of the total of all the payments made. Thus at the end of this loop (which will be of the form of a "Repeat/Until" the balance is zero or less loop), we will be able to output how long it took. We can draw a flowchart to implement the solution to the problem (see page 128).

The program to implement the flowchart is then straightforward.

```

***** FASTER CHARGE PAYMENT PROGRAM *****
***** VARIABLE DICTIONARY *****
*          BAL: IS UNPAID BALANCE (INPUT INITIALLY)
*          PAY: IS PAYMENT CALCULATED EACH MONTH
*          COST: IS TOTAL COST TO THE USER-MONTH BY MONTH
*          M: IS COUNTER TO KEEP TRACK OF HOW MANY MONTHS
*          INT: IS THE INTEREST CHARGED BY THE BANK EACH MONTH

```

```

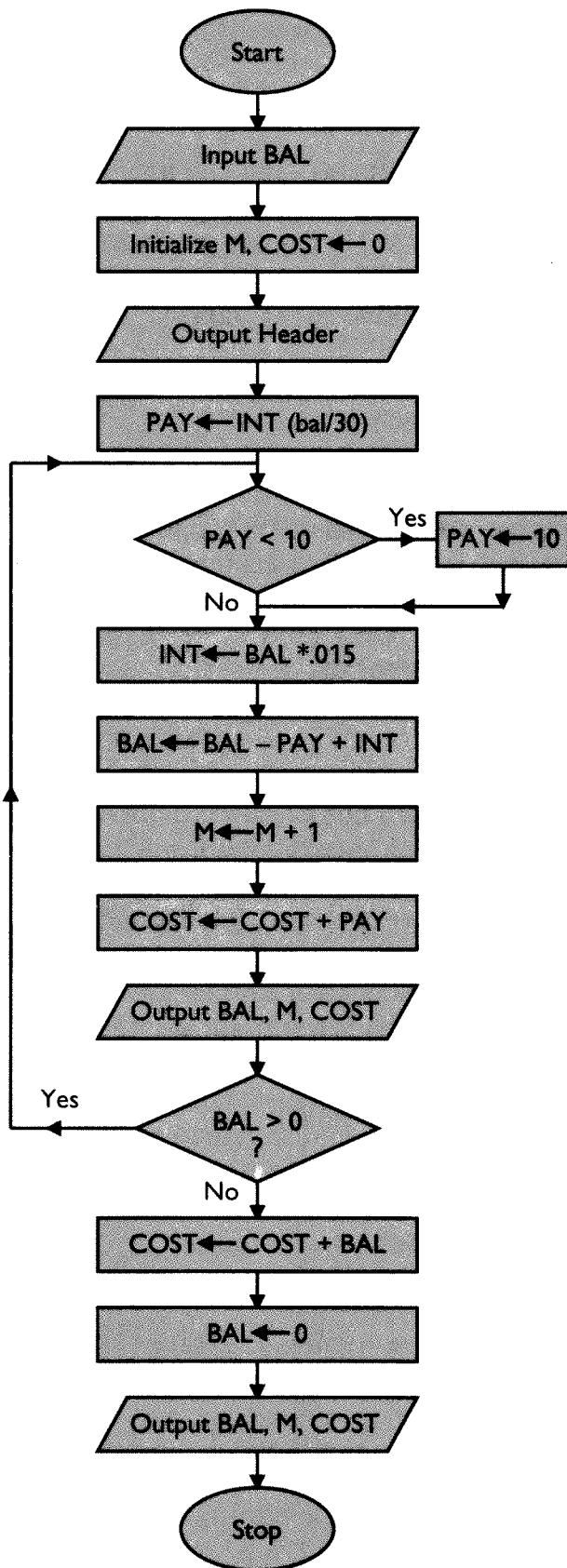
REAL BAL, PAY, COST, INT
INTEGER M
M = 0
COST = 0.0
PRINT*, 'ENTER THE AMOUNT YOU SPENT, AS A REAL VALUE'
READ*, BAL
PRINT*, 'FASTER CHARGE PAYMENTS ON $', BAL
PRINT*
PRINT*, 'BALANCE MONTH TOTAL COST'
PRINT*
50 CONTINUE
    PAY = IFIX(BAL/30)
    IF (PAY .LT. 10) PAY = 10
    INT = BAL * 0.015
    BAL = BAL - PAY + INT
    M = M + 1
    COST = COST + PAY
    PRINT*, BAL, M, ' ', COST
    IF (BAL .GT. 0) GO TO 50
***** ADJUSTMENT IN CASE LAST FULL PAYMENT OVERPAYS *****
COST = COST + BAL
BAL = 0.
PRINT*, BAL, M, ' ', COST
STOP
END

```

If the amount spent were \$500.00, the output of this program would look something like:

FASTER CHARGE PAYMENTS ON \$500.00		
BALANCE	MONTH	TOTAL COST
491.50	1	16.00000
482.87	2	32.00000
474.12	3	48.00000
466.23	4	63.00000
...
17.44	68	759.0000
7.70	69	769.0000
-2.18	70	779.0000
0.00	70	776.8200

Thus we see that, if we make the minimum payment each month, it took 70 months to pay it off, and cost \$276.82 in interest!



The Barbarian and the Roman Soldier

In this problem, a soldier in the Roman legion has been caught, out in the open without his weapons, by a wandering barbarian. The barbarian shoots an arrow at the soldier, who begins to run at the same instant that the arrow is released from the bow. If we assume that the arrow travels at a constant velocity (a somewhat inaccurate assumption, since it must accelerate from rest until its terminal velocity, determined by friction, is reached) of 72 miles per hour, and the Roman soldier runs at a constant speed (again, a somewhat inaccurate assumption) of 12 miles per hour, print out a table of the distance between the arrow and the soldier every second until the arrow hits him, if he was 100 feet away from the barbarian when the arrow was loosed.

To solve this problem, we clearly need first of all to use consistent units. Since the distance between barbarian and soldier is expressed in feet, and we are to print out a table of distances every second, our speeds should also be expressed in feet per second. We know that a mile is 5280 feet, and that an hour contains 3600 seconds, so we can have our program convert miles per hour into feet per second. Then we need to construct a table of distances every second until the arrow hits the soldier; this will be a "Repeat/Until" loop that continues until the arrow and the soldier meet. We could analytically calculate just how long the arrow will be in flight. Since the speed differential between soldier and arrow is 60 mph, we simply have to calculate how long it takes something travelling 60 mph to cover 100 feet (100 feet/88 feet per sec.), and discover it will take 1.13636+ seconds, or not quite 12 tenths of a second. We could, then, use a definite iterative loop that executed 12 times, but the problem should instead be done in terms of the decreasing distances between the arrow and the soldier. Thus, we will assume a "snapshot" view of the action, seeing a still shot every second, and continuing until the distance passes zero.

```
***** BARBARIAN AND THE ROMAN SOLDIER *****
***** ASSUME ARROW TRAVELS 72 MPH, SOLDIER TRAVELS 12 MPH *****
***** AND THEY BEGIN 100 FEET APART *****

      REAL SPEEDA, SPEEDS, DIST, POSA, POSS
      INTEGER TSEC
      SPEEDA = 72*5280/36000.0
      SPEEDS = 12*5280/36000.0
      DIST = 100.0
      POSA = 0.
      POSS = 100.
      TSEC = 0
      PRINT*, ' SOLDIER ARROW DISTANCE'
      60  CONTINUE
          TSEC = TSEC + 1
```

```

POSA = POSA + SPEEDA
POSS = POSS + SPEEDS
DIST = POSS - POSA
PRINT*, POSS, POSA, DIST
IF (DIST .GT. 0) GO TO 60
PRINT*, 'DEAD SOLDIER IN LESS THAN ', TSEC, 'SEC/10'
STOP
END

```

We could elaborate the program to look in more detail at the last excruciating partial second, say in hundredths of a second. Note that our output would tell us the relative positions of arrow and soldier at the end of 11 tenths-of-a-second, to begin.

FORTRAN 90 FEATURES

For the logical relations in FORTRAN 77 (.LT., .LE., .GT., .GE., .EQ., and .NE.), Fortran 90 adds the following corresponding alternatives: <, <=, >, >=, ==, and /=.

SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

To set up *selection* and *repetition*, our language needs the ability to make *tests*, to have a command that implements the flowchart *decision box*. The sorts of *relations* we can test for between two values are whether they are .LT. (less than), .LE. (less than or equal to), .GT., .GE., .EQ., or .NE. each other. Such a relation (which will have the logical value *true* or *false*) can then be tested in a Logical IF statement. These tests may be used in a simple Logical IF structure, of the form:

```

IF (rel'n) executable statement
    next-statement

```

where, if the relation tested is true, the statement directly following the test is executed; otherwise not. This is said to be a *conditional* execution, and can be used to insert a command on the condition that some relation turns out to be true. Used in conjunction with an *unconditional branch*, of the form GO TO n, it can be used to create finite repetition structures. For example, to repeat a set of instructions 10 times, we can write:

```

N = 1
4      [body of instructions to be repeated]
N = N + 1
IF (N .LE. 10) GO TO 4

```

The variable N here is used as a *counter*, to control the loop.

The logical test can be used in more elaborate structures as well. The Block IF, or IF/THEN, structure, allows any number of statements (not just one, as in the Logical IF), to be executed conditionally. If is of the form:

```
IF (rel'n) THEN
    [block of statements to be executed]
ENDIF
```

The Block IF also has a variation, in which one set of statements can be executed if the relation turns out true, another set to be executed if it turns out to be false:

```
IF (rel'n) THEN
    [True block]
ELSE
    [False block]
ENDIF
```

The final variation on the block IF form is to allow a number of conditions to be tested successively, and the block to be executed following the first condition (if any) that is satisfied; if none is satisfied, an optional ELSE may be used:

```
IF (rel'n1) THEN
    [block1]
ELSEIF (rel'n2) THEN
    [block2]
...
ELSEIF (rel'nk) THEN
    [blockk]
ELSE
    [all-else-fails block—optional]
ENDIF
```

The logical relations expressed in block IFs can also be compound conditions, combining simple logical expressions with connectives such as .NOT., .AND., .OR, and .EQV. or .NEQV. (listed in order of precedence).

One other form of IF, the Arithmetic IF, tests the *sign* of a numeric value, and branches to one statement number (n_1) if it is negative, a second (n_2) if it is zero, and a third (n_3) if it is positive. It is considered *obsolescent* in Fortran 90.

```
IF (arith. exp.) n1, n2, n3
```

These branch and test statements may be used to set up several different kinds of loops. We have already seen a loop structure which executes exactly N times:

```
K = 1
n      [block of loop]
K = K + 1
IF (K .LE. N) GO TO n
```

We can also construct a loop that will “repeat until” some specified condition is met. To implement this, we used the neutral CONTINUE statement to mark the beginning of the loop:

Repeat	n	CONTINUE
[block]		[block]
Until condition		IF (not-cond) GO TO n

A loop that will repeat *while* some condition holds (a pretest loop, as opposed to the earlier posttest loop forms) is set up:

While (condition)	IF (condition) THEN
[block]	[block]
Endwhile	ENDIF

An *accumulator* (of the form $S = S + T$) can sum up totals.



EXERCISES

- 1. Read in an integer value representing a year and determine whether it is a leap year (divisible by 4) or not.
- 2. Create a letter-grade program that will read in a set of 5 letter grades (including A, A-, B+, etc.), convert them to equivalent numeric grades (4.0, 3.7, 3.3, etc.), and compute the average of the 5 numeric grades, that is, a G.P.A.
- 3. Write a loop which will sum up the terms in the series:

$$1 + 1/2 + 1/3 + \dots + 1/100$$

in the order given and print out the result. Then write another loop which will sum up the series in reverse order.

- 4. Write a program which will read in any relatively small integer n , and compute and print out $n!$ (n factorial).
- 5. Write a weight-processing program that reads weights until a negative “flag” weight is encountered so that it averages all of the good weights read in and prints out the average.
- 6. Write a program that would read in 1000 integer grades and count how many are in the range from 75 to 85 inclusive.
- 7. Read in a four-digit integer and print out the integer with its digits reversed (e.g., 4576 reverses to 6754).

8. Add up the terms in the series:

$1 + 1/2 + 1/4 + 1/8 + \dots$ until the sum exceeds 1.95. Print out sum and number of terms.

- **9.** For each of the following, what value will be stored in A or J, respectively, at the end of the sequence? Please assume default typing for all variables here.

1) A = 2.0	2) M = 9	3) T = 1.0
N = 1	J = 1	A = 0.0
3 A = A*A	2 CONTINUE	M = 1
N = N + 1	J = J + 1	4 IF (M.LT.4) THEN
IF(N-3) 3,3,6	M = M/J	M = M + 1
6 PRINT*, A	IF(M.GT.0)GO TO 2	A = A + T
	PRINT*, J	T = T + 2.0
		GO TO 4
		ENDIF
		PRINT*, A

10. For each of the following sequences, what value will be stored in A or J, respectively, at the end of the sequence? Assume default typing.

1) J = 0	•2) M = 1	3) A = 1.0
K = 6	J = 0	T = 1.0
6 K = K - 1	9 CONTINUE	7 IF (A .LE. 1.9)THEN
IF(K) 8, 7, 7	J = J + M	T = T/2.0
7 J = J + 1	M = M*3	A = A + T
GO TO 6	IF(M.LT.50)GO TO 9	GO TO 7
8 PRINT*, J	PRINT*, J	ENDIF
		PRINT*, A

11. Recall the problem in the last chapter (question 10) where you were to read in any amounts of nitrogen and hydrogen (in grams), and then calculate and print out how much ammonia you could make with those amounts. Modify that program (now that you have the power of decision statements) so that you can test to see whether it is the amount of nitrogen or the amount of hydrogen that will limit the amount that can be combined, and print out the resulting amount of ammonia in either case.

12. Write FORTRAN instructions which will store the largest of three input values A, B, and C into location BIG.

13. Write a program which will calculate and print out the product of all of the odd integers from 31 to 55.

- 14.** Assume you have been given the task to interchange the values in two integer variables, and the added restriction that you may not use a third temporary variable location to do so. See if you can arrive at a method to accomplish this.
- **15.** Write a loop which will add up all of the fractional terms in the indicated sequence, *in the order given*.

1/200, 1/199, 1/198, 1/197, . . . , 1/101

- 16.** If you are adding up the integers in their natural sequence (1, 2, 3, etc.), how many terms must you add up for the sum to exceed 10,000? Write a program to determine this.

- 17.** You drew a flowchart for this problem in Chapter 1, and now you can write a FORTRAN program to implement it. An employee is paid on a graduated pay scale, so that he receives \$4 an hour for regular time, that is, hours up to 40 hours worked; he receives time-and-a-half, or \$6 an hour, for any hours worked between 40 and 60 hours; and he receives double time, or \$8 an hour, for any hours over 60 hours worked. Read in an input number of hours worked, and have your program calculate and print out the employee's gross salary. You may use any of the decision structures (IFs) that we covered in this chapter.

- **18.** This problem is *analogous* to the previous one, and also like one for which you drew a flowchart in Chapter 1. People in a certain country pay taxes on their income according to a graduated scale. The scale is represented by the following table:

Income	Tax on Income
0 – \$5000	No Tax
\$5000 – \$20,000	12% on amount over \$5000
\$20,000 – \$45,000	20% on amount over \$20,000
\$45,000 up	28% on amount over \$45,000

Be careful in solving this problem. Notice, for instance, that someone making over \$45,000 pays 28% on any dollars over \$45,000, *and also* 20% on the dollars in the range from \$20,000 to \$45,000, and also 12% on the dollars in the range from \$5000 to \$20,000. Write a FORTRAN program which will read in any taxable amount of income, and implement this table to calculate how much the person should pay in taxes. Print out the bad news.

- 19.** The kingdom of Updown has an inverted income tax, where the rich pay little or nothing in taxes, and the poor pay a lot (sound familiar?). The rules are as follows: anyone who makes \$60,000 a year or more pays no tax at all. A person who makes between \$40,000 and \$60,000 pays 5% on dollars in that range, and correspondingly more on the dollars below \$40,000 (consult the

rest of the set of rules). Persons making between \$20,000 and \$40,000 pay 10% on dollars in that range, and more on the dollars below \$20,000. Persons making between \$0 and \$20,000 pay 20% on any dollars in that range. Write a program which will read in any income amount in dollars, and figure that person's income tax according to this graduated, perverted scale.

- **20.** Recall the program you wrote at the end of the last chapter (problem 12), where you read in an element, its specific heat, and its atomic weight, and tested out the Law of Dulong and Petit. In that chapter, you did not have loops, so you had to run the program six times if you wanted to test all six elements in the table. Now that you know about loops, modify that program so it will accept six inputs, and print out an informatively labelled table of results.

21. Special numbers called *numerical centers* are integers such that, if you add up all of the positive integers in sequence that are less than the center, you get a certain low sum, which can then be balanced by an equal high sum of the integers in sequence greater than the center. For example, 6 is a numerical center because its low sum is $1 + 2 + 3 + 4 + 5$, or 15, and if you begin adding up the integers in sequence that are greater than 6, that is 7, then $7 + 8 = 15$, you get a balance. There are two more numerical centers between 7 and 220. Write a program which will find them and print them out. Make your program as efficient as possible, by noticing that each new low sum is related to the previous one, and you can determine it by simply adding one term to the previous low sum. This is much more efficient (that is, takes much less time) than adding up all of the integers from 1 to $n-1$ for each candidate numerical center n .

22. Now assume that the *first* numerical center you found in the previous program represented the number of a locker at a bus station that contains money that was stolen 35 years ago, and is thus beyond the statute of limitations, and the *second* numerical center you found is a three-digit number which, when broken up into its individual digits, represents the combination to the locker. If you can print out the locker number and the combination, you will be rich. Modify your program to do so.

23. The state of Pennsylvania has a speed limit of 55 mph, and charges fines if anyone exceeds the speed limit according to:

55 – 60 mph	\$92.50
60 – 65 mph	\$102.50
65 – 70 mph	\$112.50
70 – 75 mph	\$132.50

Since 55 is the legal speed limit, it is clear that the Pennsylvania state police must enforce the table as follows: speeds over 55 but less than or equal to 60 are fined \$92.50, speeds over 60 but less than or equal to 65 are fined \$102.50, and so on. The posted sign does not make it clear what happens to speeders

who go over 75 miles an hour; perhaps they are shot on the spot. Write a FORTRAN program which will calculate the amount of the fine for an input speed of the offender.

- 24.** Newton's Second Law (action = reaction) implies the principle of conservation of momentum (mass x velocity). If a rocket of mass 2000 kg. is shot vertically upward at a velocity of 100 m/sec., with what velocity does the earth recoil (the mass of the earth is approximately 5.98×10^{24} kg.)? The kinetic energy of a body of mass m moving with velocity v is $1/2 mv^2$; what is the ratio of the kinetic energy of the rocket to the kinetic energy of the earth generated in reaction? Run a FORTRAN program which examines this problem for velocities of the rocket varying from 100 to 160 m/sec., in steps of 10 m/sec.
- ◆ **25.** This is similar to problem 24 at the end of the last chapter. If a projectile is fired with a velocity V at an angle of A radians above the horizontal, its maximum horizontal range is $V^2 \sin 2A/g$ (where g is the acceleration of gravity). If a cannon ball is shot off with a velocity of 700 m/sec. at various angles from 20° to 70° , in steps of 10° , print out the angles and the various horizontal ranges they have. Remember to convert degrees (d) to radians, use $\pi d/180$. Use the system SIN function (see Appendix B), which will give the sine of an angle A expressed in radians, from the FORTRAN expression SIN(A).
- 26.** An index to measure water quality as a function of pH value X is represented in four *segments*, as follows:

$0 \leq X \leq 5$	$\text{Index} = -0.4 X^2 + 14$
$5 < X \leq 7$	$\text{Index} = -2X + 14$
$7 < X \leq 9$	$\text{Index} = X^2 - 14X + 49$
$9 < X \leq 14$	$\text{Index} = -0.4 X^2 + 11.2X - 64.4$

Write a FORTRAN program which creates a table of water quality index values for pH values (X) from 0 to 14 in steps of 0.5. Later you can plot this data. [The relations expressed in this set of rules were found in Wayne R. Ott, *Environmental Indices: Theory and Practice* (Ann Arbor Science Publishers, 1978), p. 65.]

- 27.** The National Ambient Air Quality Standards (NAAQS) for pollutants can be expressed as limits (in mg/m³) which are not to be exceeded more than once a year. (These values are based on Ott, p. 99.) Write a FORTRAN program which will read in a two-letter code for a pollutant and its average measure in mg/m³ and, according to the table, print out an ALERT message if the standard level specified is exceeded. These limits for 6 different pollutants can be expressed roughly in the following table:

Pollutant	Primary NAAQS (in mg/m ³)
Carbon Monoxide (CO)	10
Nitrogen Dioxide (ND)	0.1
Hydrocarbons (HC)	0.16
Photochemical Oxidants (PO)	0.16
Particulate Matter (PM)	0.26
Sulfur Dioxide (SD)	0.365

28. The Environmental Protection Agency has a Pollution Standards Index (PSI) with the following ranges of values and the corresponding health effect descriptors:

Index Value (PSI)	Health Effect Descriptor
0 – 50	Good
50 – 100	Moderate
100 – 200	Unhealthful
200 – 300	Very Unhealthful
300 – 400	Hazardous
400 – 500	Very Hazardous

Write a FORTRAN program which will read in a PSI value and (using IF tests) print out the appropriate health effect descriptor. [Note: This table is from Ott, pp. 149–150. At the end of the chapter on arrays (7), we will have a more detailed problem.]

29. *Conservation of Momentum.* Brian Boitano and Debi Thomas are on skates facing each other at rest. Brian (who weighs 180 pounds) gives Debi (who weighs 110 pounds) a push and sends her backwards at 15 miles per hour. Write a FORTRAN program to determine at what speed Brian moves in the opposite direction and (assuming no friction) output a table of the distances they both cover in the next 5 seconds, in steps of 0.5 seconds.

30. A *perfectly inelastic collision* is one in which two bodies collide and then move off together as one body. Assuming conservation of momentum, if a bullet of 20 grams strikes a wood block of 5 kg resting on ice with a velocity of 400 m/sec, write a short FORTRAN program to determine the speed they move away at.

CHAPTER 4



REPETITION STRUCTURES

Repetition, whether it is just creating a multiplication table or testing many alternative branches for a move in a sophisticated chess-playing program, is one of the things a computer does best. In your computer-problem-solving career you will write hundreds or even thousands of loops designed to perform a wide variety of tasks. Most interesting problems involve some kind of a loop. FORTRAN implements one easy-to-use and powerful formal loop structure—the DO loop—which you will learn in this chapter. With a knowledge of repetition structures, you should be able to meet a whole new range of problem-solving challenges.

"What if one does say the same things,—of course in a little different form each time—over and over? If he has anything to say worth saying, that is just what he ought to do."

- Oliver Wendell Holmes, Over the Teacups

We have already seen how to set up various kinds of loops using the branch and test structures you have learned. However, as you have seen, this is awkward, and generally programs written with such loops do not obviously display their intent to the casual reader of the program. Thus FORTRAN has standardly implemented one of the loop structures, that which repeats a definite number of times, as a "DO loop." We will examine this repetition structure in some detail. Some FORTRAN 77 compilers have implemented a type of "While" structure—and we will briefly examine those—but this is not part of the Standard, so you may have to rely on the simulation of the "While" structure we have discussed. There is no "Repeat/Until" structure in Standard FORTRAN, so you will have to use the simulation. We will review these simulation structures and their applications for you in this chapter, and discuss other variations that will handle similar problem situations.



THE DO LOOP

A DO loop will execute instructions a definite number of times. As we have seen when we have constructed loops of this type using IFs and GO TOs, such a loop is controlled by a loop *index*—a variable which changes as the loop progresses, and whose value determines when the loop is completed. This index has an *initial value*, a *final value*, and a *step size*. The step size is the amount by which the index value changes after each repetition of the loop. The Standard form for the initiating statement of a DO loop in FORTRAN 77 is:

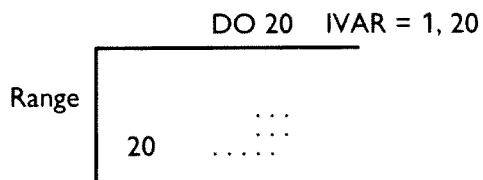
DO # [,] var = initvalue, finalvalue [,stepsize]

In this statement, # stands for a statement number which will mark the *last* executable statement in the loop body. Var is a variable, which may be integer, real, or double precision, which is the loop index. As we generally will do in this book, any part of a statement that is *optional* we will enclose in square brackets—as the [,] after the DO # and the [,stepsize]. The initial value and final value (and step size, if used) for the loop index may be any legal integer, real, or double precision expression. If the step size is not specified, it will be assumed to be +1, since this is the most common step size used to control loops. The "DO" statement thus sets up the DO loop *parameters*—the values that control the execution of the loop.

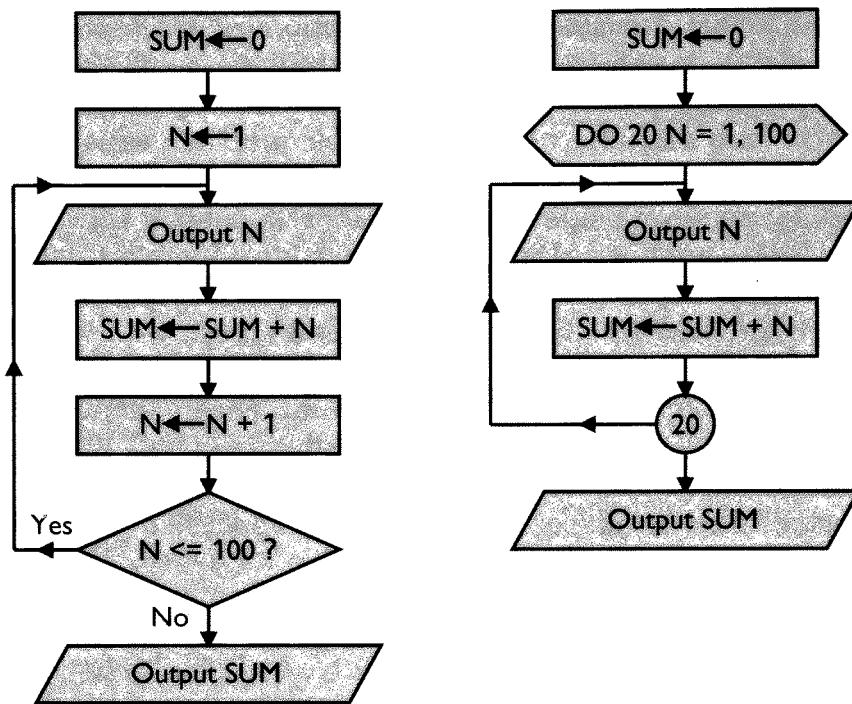
[Note: In all standard versions of FORTRAN prior to FORTRAN 77, the index variable could only be an integer variable, and the initial value, final value, and step size had to be integer constants or variables, not complicated expressions. In Fortran 90, you may also use real or double precision variables as index variables to control a DO loop, and the other DO loop parameters can be any *scalar* (that is, nonarray) numeric expressions (including integer, real, or double precision); but the real and double precision options are designated as *obsolescent* (which means they may well be removed as options in the next version after Fortran 90). Thus, if you choose to make use of real or double precision parameters, be aware that these options are only available in FORTRAN 77 (and Fortran 90), but not in any earlier compilers, and may not be available after Fortran 90. We will generally adhere to using integer variables to control the DO loops in this book, and if we do use a real variable we will include a comment regarding its limited applicability.]

Range of a DO Loop

Just as we needed an ENDIF to mark the range of statements controlled by a block IF, ELSE, or ELSEIF statement, we need to know what the range of a repetition structure is—that is, which statements are included under its control. The Standard DO statement begins “DO #”, where # represents a statement number which follows the DO in the program. The DO loop *body*, the set of instructions that repeats under control of the DO loop parameters, is the group of instructions beginning at the one immediately following the DO statement and ending on the statement with the indicated statement number (#)—this is called the *range* of the DO loop. For example,



We will illustrate this with an example of a loop structure we already learned how to do without DO loops—that of generating the integers from 1 through 100, adding them up, and printing out each integer as it was generated and then printing out the total. We will write the program the “old way” on the left, and the “new way,” using DO loops, on the right, so that you can readily see the comparison, and the parallelism of the structures. We have preceded the program segments by flowcharts for both versions, introducing a new flowchart notation for the DO loop.



```

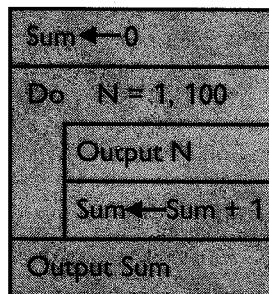
INTEGER SUM, N
SUM = 0
N = 1
5   PRINT*, N
     SUM = SUM + N
     N = N + 1
     IF (N .LE. 100) GO TO 5
PRINT*, 'SUM IS ', SUM
  
```

```

INTEGER SUM, N
SUM = 0
DO 20 N = 1, 100
      PRINT*, N
      SUM = SUM + N
20  CONTINUE
PRINT*, 'SUM IS ', SUM
  
```

These two program structures accomplish the same job, and you can see that the one on the right, using the DO loop, does it much more clearly and simply. Further, it is easier to read the program on the right and determine what it is doing. The "range" of statements executed by both program loops is the same. The difference is that the DO statement accomplishes in one statement (plus the marking of the terminal statement of the loop) what it took three statements and a statement marker to accomplish in the "primitive way": (1) initializing the index variable, (2) incrementing the index variable by the step size (whether it is implicit or explicit), and (3) testing to see if the loop should be repeated another time. We have indented the *body* of the loop (that is, the set of statements that is repeated by the loop structure) in both examples. This is a good habit to develop, since it enhances program readability.

You may find the Nassi-Schneiderman (N-S) notation preferable for DO loops. For our problem, the N-S representation would be:



The loop parameters (index value, initial and final values, and step size) should not be changed in the code in the body of the loop. In FORTRAN 77, the loop index variable *may not* be altered in the loop itself (such an attempt to change it would result in a compiler error). Older versions of FORTRAN did not forbid this, only advised against it; such alteration of the loop index could cause strange results. The other parameters of the loop (initial and final values and step size) may be expressions, as well as constants. Their values should not be changed during the course of the loop. However, if they are variables or expressions, they are evaluated and their values saved when the loop is initiated; thus changing values involved in the expressions will not alter the actual limits of the loop.

A word about the loop index itself. Older programming texts often used to say that the value of the loop index was “undefined” upon leaving the loop, in an attempt to discourage the programmer from trying to use the loop index value on exit. This is simply not true; the value of the loop variable is indeed defined—it is the *last* value it held during the execution of the loop. However, it is best not to try to make use of the loop index value after leaving the loop.

Our loop examples so far have always run “forward,” that is, they have had positive step sizes. A DO loop in FORTRAN 77 (and Fortran 90, though not in most earlier FORTRAN versions) may also be made to run “backward,” that is, from some higher value to some lower value, using a negative step size. An example of such a loop would be the following, to print out the *even* numbers from 100 down to 2:

```

DO 60 N = 100, 2, -2
60      PRINT*, N
  
```

DO loops have an additional feature in FORTRAN 77, that of “prechecking” a loop, a feature we will discuss next.

Prechecked Loops

We have indicated earlier that a DO loop accomplishes the same job as a parallel loop set up with an index, using IFs and GO TOs. For example,

```
DO 10 I = 2, 25, 3           I = 2
      ...
10  CONTINUE                 6      ...
                                I = I + 3
                                IF (I .LE. 25) GO TO 6
```

perform the same task. And, since a DO loop in FORTRAN 77 (or Fortran 90) can run backward, that is, with a negative step size, another parallel setup in which the program segments accomplish the same job would be:

```
DO 20 M = 100, 1, -3         M = 100
      ...
20  CONTINUE                 9      ...
                                M = M - 3
                                IF (M .GE. 1) GO TO 9
```

In all versions of FORTRAN before FORTRAN 77, DO loops actually were implemented in the way our previous parallel examples suggest. FORTRAN 77, however, is implemented differently, and we will discuss that implementation shortly. Generally, these loops work the same way, whether set up using DOs or IFs and GO TOs. In fact, in versions of FORTRAN up to, but not including, FORTRAN 77, such parallels helped the programmer to understand how FORTRAN would handle a seemingly "impossible" loop such as:

```
DO 30 N = 5, 3
30      PRINT*, N
```

Because the index variable N *cannot* go from 5 to 3 (in steps of +1), this seems to be a loop that cannot execute. However, all earlier FORTRAN compilers would execute this loop once, printing out a value of 5, and then quit the loop. If you construct the parallel loop using IFs and GO TOs, you will see why this is so. However, the FORTRAN 77 designers decided that such a loop should not be executed, since it is an *impossible loop*, and should be skipped entirely. Prior to FORTRAN 77, the programmer could just avoid writing loops like the one in our example, or could test the lower and upper limits against the step size, and skip the loop if it would not work, as in this example:

```
READ*, MAX
IF (MAX .LE. 12) THEN
    DO 40 MM = MAX, 12
        ...
40      CONTINUE
ENDIF
```

However, FORTRAN 77 has a built-in feature which *prechecks* the legitimate range of all DO loop parameters *before* executing the loop. If the loop turns out to be an impossible loop, it is skipped, and the program picks up at the next executable statement following the loop. The FORTRAN 77 precheck is performed automatically by the compiler in the following way. Assume that you have a DO statement with these parameters:

```
DO # var = init, final, step
```

The computer calculates the number of times the loop will repeat, or its *iteration count*, as the maximum of zero and the following calculated result (where INT takes the integer part of the value):

$$\text{INT}((\text{final} - \text{init} + \text{step})/\text{step})$$

Using this procedure, we calculate the iteration counts for the following DO statements:

<u>DO STATEMENT</u>	<u>ITERATION COUNT</u>
DO 20 I = 1, 100	$(100 - 1 + 1)/1 = 100$
DO 30 M = 40, 2, -2	$(2 - 40 + (-2))/(-2) = 20$
DO 35 J = 3, 100, 4	$(100 - 3 + 4)/4 = 25$
DO 40 K = 5, 3	$(3 - 5 + 1)/1 = -1; \rightarrow 0$
DO 50 N = 2, 10, -2	$(10 - 2 + (-2))/(-2) = -3; \rightarrow 0$

Whenever the iteration count is calculated to be zero (as in our last two examples), the DO loop is skipped entirely. A DO loop is said to be *inactive* until its DO statement is executed, when it becomes *active*. However, an impossible loop—one which has an iteration count of zero—remains inactive.

Thus the FORTRAN 77 DO loop implementation first evaluates and saves all of the loop parameters, determines the iteration count, initializes the loop index, and executes the range of the loop (if the iteration count is not zero). After the terminal statement of the loop is executed, the loop index is incremented, the iteration count is decreased by 1, and control is transferred back to the head of the loop, where the iteration count is tested again and, if it is nonzero, the process (execute the range of the DO, alter loop index and iteration count, transfer back to test on iteration count) is repeated. Thus, we could show the parallel between a FORTRAN 77 DO loop and its equivalent implementation without using DO as follows:

DO 50 V = init, final, step	ITER = MAX(INT((final -
...	& init + step)/step), 0)
...	V = init
50 CONTINUE	FIN = final
	STEPS = step

```

10 IF (ITER.GT.0) THEN
    ...
    ...
    v = v + STEPS
    ITER = ITER - 1
    GO TO 10

```

Notice that the expressions for the final value and step size are evaluated and saved before the loop begins, so that even if they are variables changed during the execution of the loop, its original iteration count will not be altered by these changes.

The CONTINUE Statement

We saw that there must be a statement number on the last executable statement of the range of a DO loop, to indicate where the loop body ends. We pointed out the similarity to marking the range of statements controlled by a segment of a block IF by a terminating ELSE, ELSEIF, or ENDIF, but none of these are statements *within* the range—they merely signal the end of the range. Programmers in FORTRAN 77 have thought it desirable to have a similar statement to signal the end of the loop body range but not part of the range itself. To do this, they have utilized the neutral CONTINUE statement, which we have already met, and advise that such a statement be used to terminate all DO loops.

This is a matter of good style, not a restriction of the language. FORTRAN 77 says that the terminal statement of a DO loop may be *any* executable statement (with certain exceptions). To make such a statement a CONTINUE is designed to make the loop body stand out, indented with respect to the controlling initial DO statement and terminating CONTINUE. (Note: The CONTINUE also serves as an analogue to the "NEXT" statement at the end of a loop in BASIC, a useful similarity for those familiar with BASIC.) You could write a loop to print out all the even integers from 2 through 200 in any of the following ways:

$N = 2$ 6 PRINT*, N $N = N + 2$ IF (N .LE. 200) GO TO 6	DO 20 N = 2, 200, 2 20 PRINT*, N	DO 30 N = 2,200,2 PRINT*, N 30 CONTINUE
--	-------------------------------------	---

Clearly the simplicity of the DO loop is preferable, and using the CONTINUE statement to terminate the loop gives the program a better, more structured appearance. Such features all contribute to good style, which is a necessary ingredient in modern programs that must be easy to read and to integrate into larger systems.

Restrictions on the Terminal Statement of a DO Loop

The terminal statement of a DO loop, that is, the one with the statement number referenced in the DO statement, can be any *executable* FORTRAN statement except the following: GO TO, Arithmetic IF, Block IF, ELSE, ELSEIF, ENDIF, RETURN, STOP, END, or another DO. Also, it may not end on a nonexecutable statement. The only nonexecutable statements you have encountered so far are type statements, PARAMETER, and Comments. Another non-executable statement you will soon meet is the FORMAT statement, used for fine control of I/O. A FORMAT statement must have a statement number, but it cannot be the terminal statement of a DO loop.

Of course, the advantage of using the structured style of terminating every DO loop with a CONTINUE statement is that you will not have to worry about remembering the exceptions.

DO and END DO (Non-Standard; Included in Fortran 90)

This section briefly covers a feature that is not part of the FORTRAN 77 Standard, but may be found in a number of FORTRAN 77 compilers and is included in the Fortran 90 compiler. Check with your instructor or look in the system manual to see if it is available for your use now.

This alternate form of the DO statement, available on some compilers, omits the mention of a statement number to identify the terminal statement of the DO. Then, since some way of marking the end of the range of the loop body is needed, an END DO statement (analogous to an ENDIF) is placed after the last executable statement of the loop. The END DO statement may be written with or without the blank (END DO or ENDDO), whichever seems more suitable for readability. This eliminates the need to use a statement number to flag the end of the DO loop and, since statement numbers in FORTRAN generally imply GO TOs (and GO TOs are to be minimized), it gives an even more structured appearance to a program. Thus a loop might be written either:

```

DO 25 N = 3,75,4          or      DO N = 3, 75, 4
...                         ...
- 25 CONTINUE               END DO

```

if this feature is available on your compiler.

Since the DO/ENDDO is not part of the FORTRAN 77 Standard, we will not use it further in examples in this book, but if it is available to you, you may use it (but be well aware that it is currently not portable to all machines). It will be available to everyone with the introduction of Fortran 90.

Exits From DO Loops

One of the rules that was strongly suggested in earlier versions of FORTRAN, and is enforced in FORTRAN 77, is that you cannot jump *into* the range of a loop from outside of the loop. That is, there can be no transfers of control from IFs or GO TOs that have as a target some statement in the body of the loop. You should readily see why this is undesirable, since it would not allow the loop parameters to be properly set up by the execution of the DO statement. Thus, when the terminal statement is encountered and the loop index is to be incremented and tested, these values would not have been set correctly.

It is, however, legal (that is, within the rules) to transfer *out* of the range of a DO loop. Generally, this will be done when some premature exit condition occurs in the body of the loop that indicates the loop should not proceed through its normal full set of cycles. This situation usually occurs if you set the upper limit on the loop greater than anything you actually expect to execute, and intend that some other condition will determine termination of the loop. For example, let us say that you want to add up terms in the series

$$1 + 1/2 + 1/3 + 1/4 + \dots$$

but, instead of wanting a sum of a definite number of terms, such as 100 terms, you want to add terms in the series until the sum exceeds 3.0, and then see how many terms it took to do that. You do not know ahead of time how many terms this will take, but you are sure it will not be as many as 100 terms. Thus, you write a DO loop that potentially will add up 100 terms, but you put a test in the loop to exit the loop as soon as the sum exceeds 3.0. This is legal in FORTRAN, and is one way to solve the problem. We will examine a program which implements this strategy.

```
***** ADD TERMS OF 1 + 1/2 + 1/3 + ... UNTIL SUM EXCEEDS 3 *****
***** USE METHOD OF PREMATURE EXIT FROM DO LOOP *****

      REAL SUM, TERM
      INTEGER N
      SUM = 0.0
      DO 20 N = 1, 100
          TERM = 1.0/N
          SUM = SUM + TERM
          IF (SUM .GT. 3.0) GO TO 25
20    CONTINUE
25    PRINT*, 'IT TOOK ', N, ' TERMS TO GET A SUM OF ', SUM
      STOP
      END
```

This technique works and is perfectly legal as far as the syntax rules of FORTRAN go. You may sometimes find that such a use of a GO TO may be the

cleanest way out of a loop for some special exit condition. This situation is so likely to occur that Fortran 90 will include an EXIT feature for DO loops (see Appendix E); in our program the GO TO line would be replaced by

```
IF (SUM .GT. 3.0) EXIT {Fortran 90}
```

However, structured programming style strongly discourages the use of a GO TO exit if some other method is available. Even though the While and Repeat/Until do not occur as part of the FORTRAN 77 Standard, we have seen how to simulate their use, and the use of either of those structures would be superior in solving this problem. We will examine these alternate methods in even more detail in the next section.



NON-STANDARD REPETITION STRUCTURES

We have already constructed "While" and "Repeat/Until" types of structures using IFs and GO TOs, and we will review them here in comparison to the DO loop that we now have available. Since the DO loop does not provide the best structure for all kinds of repetitions we may encounter, it is good to have other options available. Not every repetition is to be performed a definite number of times, or for a particular fixed range of values, which is what a DO loop is best suited to accomplish. Often, as you have seen already, you need to *repeat* a loop *until* some condition is met, or continue its execution *while* some condition holds true.

Let us look again at the forms of Repeat/Until and While structures, and the ways we have found to simulate them in Standard FORTRAN 77. A Repeat/Until structure is a "posttest" loop which executes a set of instructions and then tests to see if it should repeat the set another time. Its general pseudocode form, and equivalent FORTRAN 77 code, are:

Pseudocode	FORTRAN 77 Implementation
Repeat Loop body Until (condition)	# CONTINUE Loop body IF (not-condition) GO TO #

The While structure, on the other hand, is a "pretest" loop, in which the condition is tested *before* the loop is executed. If the test condition is true, the loop body executes, and the program returns to the test condition again. The general pseudocode form of the While structure and its FORTRAN equivalent are:

Pseudocode	FORTRAN 77 Implementation
While (condition) Loop body EndWhile	# IF (condition) THEN Loop body GO TO # END IF

Several FORTRAN 77 compilers have incorporated a While-type structure, though it is not part of the Standard; however, a form of it is included in Fortran 90. The implementation of the While construct is not uniform among the current compilers that do include it. On several compilers it takes the following form, which parallels Pascal-like syntax:

<pre>WHILE (condition) DO Loop body END WHILE</pre>	{may be found on some compilers, but it is <i>non-Standard</i> }
---	--

However, other compilers (most notably on VAX machines) have adopted the form:

<pre>DO # WHILE (condition) (or) Loop body # CONTINUE</pre>	<pre>DO WHILE (condition) Loop body END DO</pre>
--	--

and this is the version that will be used in Fortran 90.

If these structures *are* available on your compiler, then you may make use of them if you find them helpful. You must keep in mind, however, that they are currently not standard; as a result, including them makes your program less portable. This should not be a major problem if you are prepared to convert them to the equivalent forms should the need for portability arise. In this text, however, we will not use WHILE constructs because they are not standard, and it is confusing to use constructs you may not have available. We will use the equivalent Standard FORTRAN structures we have developed to do the same job. We will in this text attempt to use the features of the FORTRAN 77 Standard as much as possible, and clearly indicate when we are digressing from the Standard. In this way, you will develop programming habits that should carry over well to any situation you may meet.

For comparison, we will implement the program we covered in the last section, that of adding terms until the sum exceeded 3.0, by using our "Repeat" and "While" structures. These will allow us to avoid the premature exit from the DO loop that we had to use in that section. Notice the subtle differences in these two implementations, one a posttest, one a pretest loop:

<pre>***** REPEAT/UNTIL STRUCTURE *** SUM = 0. N = 0 4 CONTINUE N = N + 1 SUM = SUM + 1.0/N IF (SUM .LE. 3.0) GO TO 4 PRINT*, N,' TERMS =',SUM</pre>	<p><i>(by 3)</i></p> <pre>***** WHILE STRUCTURE ***** SUM = 0. N = 0 5 IF (SUM .LE. 3.) THEN N = N + 1 SUM = SUM + 1.0/N GO TO 5 ENDIF PRINT*,N,' TERMS=',SUM</pre>
---	--

Notice that both tests are phrased the same way, as we have implemented the loops. However, if you stated the first in pseudocode form, you would say something like "Repeat the summation until (condition) the sum exceeds 3.0," and if you had a true "Repeat/Until" structure available, it would say:

```
SUM = 0.  
N = 0  
REPEAT  
    N = N + 1  
    SUM = SUM + 1.0/N  
UNTIL (SUM > 3.0)
```

{this structure is
not available on
any FORTRAN compiler
we know of}

Thus you see that in our Standard FORTRAN implementation we have used as our test condition for further repetitions the negation of this "Repeat/Until" termination condition, since we used

```
IF (SUM .LE. 3.0) GO TO 4
```

If you had a WHILE construct available on your compiler, you could write this problem solution as follows:

SUM = 0.	SUM = 0.	
N = 0	{or}	N = 0
WHILE (SUM .LE. 3.0) DO		DO WHILE (SUM.LE.3.0)
N = N + 1		N = N + 1
SUM = SUM + 1.0/N		SUM = SUM + 1.0/N
END WHILE		END DO

The version on the right will run under Fortran 90.

For example, imagine using these two structures to set up a loop that will read and process all legitimate (positive) values of a measurement made, add them up, and finally, average them. A "flag" negative value will be used at the end of the list to mark the end of the good data to be processed. It is possible that on some runs of the program there may be *no* measurement data to process, in which case this will be indicated by a negative value as the *only* entry in the list, so in both structures the operation of reading is performed once before the loop structure is even entered.

***** REPEAT/UNTIL *****	***** WHILE/DO *****
SUM = 0.	SUM = 0.
KOUNT = 0	KOUNT = 0
READ*, VALUE	READ*, VALUE
IF (VALUE.GE.0)THEN	9 IF (VALUE.GE.0) THEN
8 CONTINUE	SUM = SUM + VALUE
SUM = SUM + VALUE	KOUNT = KOUNT + 1

```

        KOUNT = KOUNT + 1           READ*, VALUE
        READ*, VALUE               GO TO 9
        IF (VALUE.GE.0) GO TO 8     ENDIF
        ENDIF
    
```

Both loops would then be followed by logic that calculated an average *if* there were any non-negative measurements, which is indicated by a positive value of KOUNT:

```

        IF (KOUNT .GT. 0) THEN
            AV = SUM/KOUNT
            PRINT*, 'THE AVERAGE MEASUREMENT IS', AV
        ELSE
            PRINT*, 'THERE WERE NO MEASUREMENTS THIS TIME'
        ENDIF
    
```

In this example, the simulated While structure seems a bit more simple and appropriate. If you were trying to implement this problem with a DO loop, it would be difficult, since you would not know how many times to execute the DO. You *could* set the upper limit to some high value that would never be reached, and test the input VALUE each time, exiting from the loop when a negative VALUE was read. But this is awkward when better structures are available.

Given the capabilities to perform repetitions, a definite number of times (DO loops), or until some condition is met, or while some condition continues to hold, you will be able to solve a great many interesting problems that involve loops. These are very important tools for your collection, and you should practice their use until it is second nature. It is also important, as we have seen, to choose the best tool for the job at hand. There are several exercises at the end of this chapter to help you sharpen your skills.



FORTRAN 90 DO STRUCTURES: VARIATIONS ON A THEME

All of the FORTRAN 77 loop forms will run under Fortran 90, and the new version will add several embellishments to the ordinary DO. We have already mentioned that the DO/ENDO form (which does not require specifying a statement number) is a feature of Fortran 90. Thus, a loop could be written either:

<u>FORTRAN 77 & Fortran 90</u>	<u>Fortran 90</u>
DO 25 I = 1, 100	DO I = 1, 100
...	...
25 CONTINUE	END DO

A DO loop may be given a name in Fortran 90; the name precedes the DO and is followed by a colon (:). Thus, you might write:

```
MYLOOP: DO J = 100, 1, -1
...
ENDDO MYLOOP      {must also have the loop name included}
```

Since the terminal statements of a DO loop no longer need statement numbers, this provides an alternate way for the programmer to keep track of which loop is which.

As we mentioned, Fortran 90 will implement an EXIT statement for DO loops. This statement will terminate execution of the loop in which it appears, and the program will transfer control to the next executable statement following the loop.

DO 50 J = 1, 5 ... IF (...) EXIT ... 50 CONTINUE	TESTER: DO J = 1, 5 ... IF (...) EXIT TESTER ... ENDDO TESTER
--	---

In the example on the right, the IF statement could have been written simply IF (...) EXIT. The execution of the EXIT instruction has the same effect as a transfer of control (a GO TO) to the first executable statement after the loop.

Fortran 90 also includes a CYCLE instruction. Execution of a CYCLE statement causes a DO loop to go into its next "cycle"—that is, the iteration count is decreased by 1 and the step size value is added to the DO variable. Examples of this follow:

DO 60 K = 10, 200, 10 ... IF...CYCLE ... 60 CONTINUE	SKIP: DO K = 10, 200, 10 ... IF...CYCLE {or CYCLE SKIP} ... ENDDO SKIP
--	--

The DO parameters (index variable, initial and final values, and step size) may be omitted in a DO in Fortran 90. This then becomes a sort of "DO indefinitely" loop structure, which should contain an EXIT statement to terminate its operation.

As we mentioned in the previous section, Fortran 90 also implements a form of the While construct:

DO WHILE (log.cond.) or DO WHILE (log. cond.)

and of course a DO statement could be given a name—for example:

MIST: DO WHILE (...)

It has been pointed out that the DO WHILE construct is not really necessary in Fortran 90, however, since the same thing could be accomplished by following the DO by an IF...EXIT:

```
DO WHILE (LEFT)    or    DO
                           IF (.NOT.LEFT) EXIT
```

NESTED LOOPS

Occasionally, you will encounter the need to have loops inside of loops. In such a case, for every value of the outer loop, a whole sequence of related values needs to be processed. A simple example would be a (big, outer) loop to process readings from a weather satellite every day; thus, there is a loop index value for each day. Additionally, for each day you must read in 12 measures of cloud-cover density and calculate the average of those values for the day. Thus, there must be an inner loop to read and add up the cloud density measures. The overall form of this problem would be:

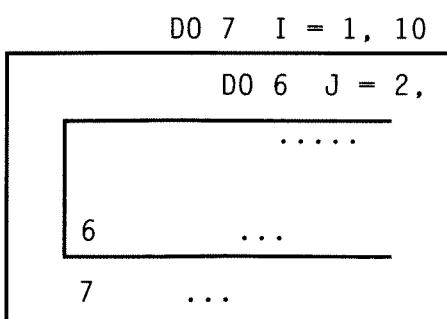
Process for each day (NDAY = 1, 365)

For each day read in and add up measures (I = 1, 12)

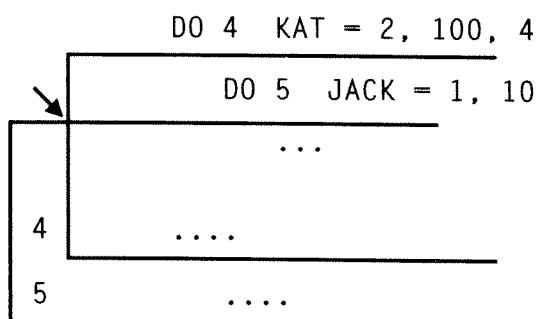
You will find many other occasions when such loops within loops are needed. First, we should attend to the details of how such structures must be set up to be legal in FORTRAN. Earlier versions of FORTRAN allowed nesting of DO loops up to fifty deep; there seem to be no limits specified in FORTRAN 77 or Fortran 90. However, for your purposes, you probably will not use loops nested more than three or four deep for some time to come.

When loops are *nested* within one another, they must be properly nested. This means that the range of the inner loop must be completely contained within the range of the outer loop. A good way to insure this is to draw the range of each loop on your program development sheet or on a program listing, to be sure that the ranges do not cross. Examples follow:

Legal Nesting



Illegal Nesting



If you can see how to nest a pair of loops properly, then you can extend this evaluation to any number of nested loops, simply looking at those which are adjacent, a pair at a time.

Generally, nested loops will terminate on different statements, and this is necessary if some action is performed between the end of one loop and the next, or if some "exit" must be taken from an inner loop to the end of an outer loop. However, FORTRAN allows two or more nested loops to terminate on the *same* statement if these conditions do not occur. Thus, the following nested loops would be legal in FORTRAN, though structured programming would advise against this. The loops are nested as indicated by the ranges drawn, and the output from the program is also given. Follow these nested loops carefully to see how they give the indicated output, to understand how nested loops work.

DO 10 I = 1,2	
DO 10 J = 4,6,2	I 4 7 I 4 8
DO 10 K = 7,8	I 6 7 I 6 8
PRINT*, I, J, K	2 4 7 2 4 8
10 CONTINUE	2 6 7 2 6 8

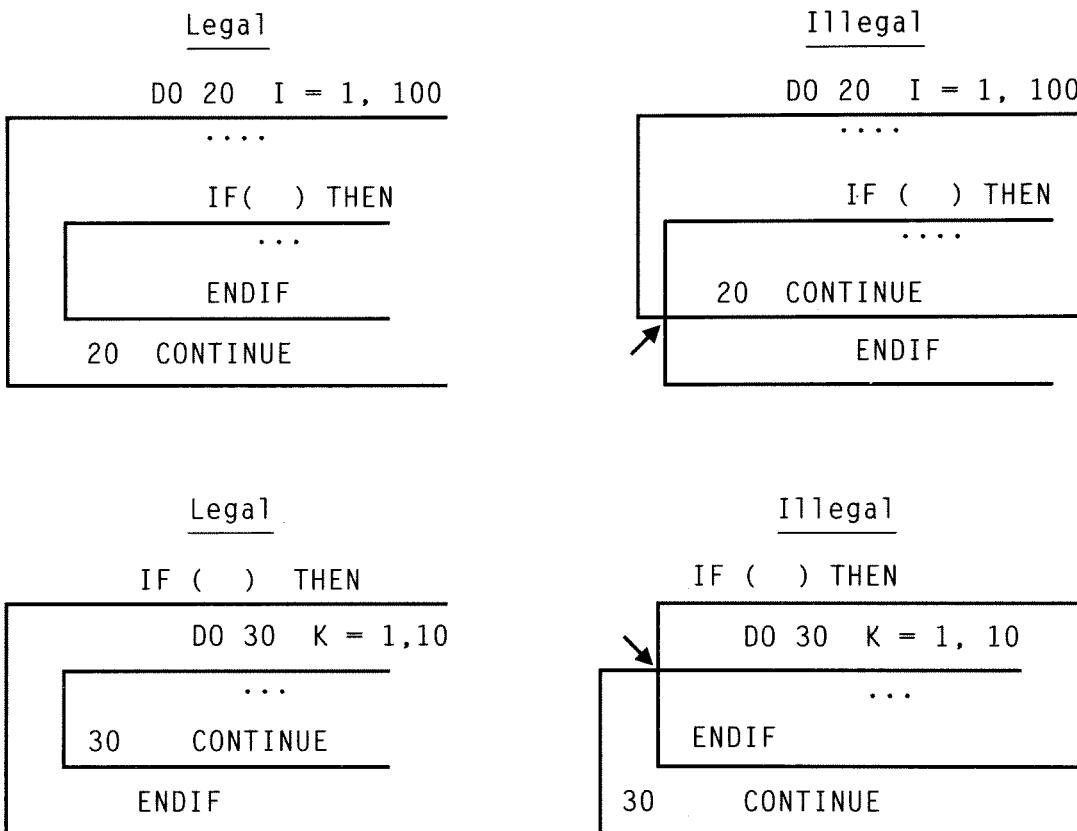
The "K loop" is the innermost loop, just as if it had ended on a different statement number, the "J loop" is the middle loop, and the "I loop" is the outermost loop. All patterns of the inner loops' values will be executed for each value of the outermost loop.

One additional restriction—if DO loops terminate on a common statement (as in our example), a branch to the terminal statement may only be made from the *innermost* loop. In FORTRAN 77, no such transfers from outer loops are permitted by the compiler. In earlier versions, strange results occurred if such branches were used, since loop indices would not be properly initialized, or the wrong loop counter incremented and tested.



PROPERLY NESTED CONTROL AND REPETITION STRUCTURES

Just as we have seen that DO loops must be properly nested, with ranges that do not cross, and we saw earlier that block-structured IFs should not have ranges which cross, it is also the case that the range of a DO loop should not cross that of an IF block and vice versa. If you do make such an error, and your program has DO and IF ranges that cross, it is often difficult to find. The error



message may say that you have "improperly nested DO loops," even if there is only one DO loop, and this can be confusing. The best protection against such errors, and the best way to track them down if they do occur, is to carefully draw the ranges of each DO loop and IF block on your program listing. This way you can see clearly if any ranges are crossed.

Thus, if an IF block is contained within a DO loop, it must be *completely contained* within the DO loop; and if a DO loop occurs within an IF block, it must be *completely contained* within the block. Since a DO loop cannot terminate on an ENDIF, there is no question of an IF block and a DO loop sharing the same terminal statement.

Some actual examples involving such nesting will be useful. Imagine that you are processing information on 100 Presidential candidates, and you have numeric measures on their honesty (a rating from 1 to 100), their drug abuse level (same scale), and their penchant for promiscuity (same scale). Your organization wants you to print out each candidate's name in an "OK" or a "NOT OK" column and, if they are not ok, the criteria on which they failed the standards you have set. Since a candidate might fail more than one criterion, you have to have a way of determining, by the time you get to the printout, which of the criteria they have failed. To do this, our program will assign a prime number to each level on which a candidate could fail: 3 for dishonesty, 5 for drug abuse, and 7 for promiscuity. These values will be summed up for each criterion they fail, and so we will be able to tell before we print out which criteria (if any) they failed.

Score	Rating/Failed Categories
0	O.K.
3	H
5	D
7	P
8	H&D
10	H&P
12	D&P
15	H,D,&P

Thus, the program to implement this process will be as follows. We have not included detailed comments about the scoring scheme in the program, since we have already explained it in the text. However, in a free-standing production program, it should be detailed in program comments. The DO loop processes the 100 candidates, and the IFs and block IF structures test them on the criteria, and determine how to categorize them.

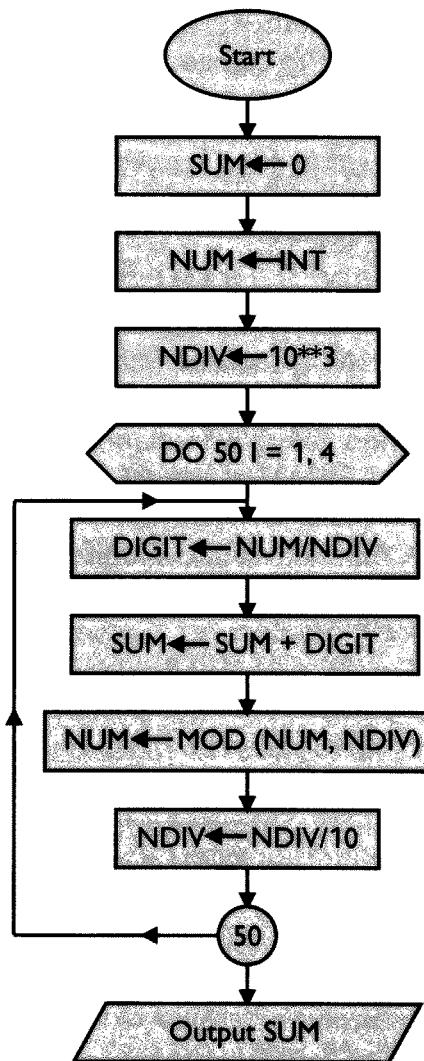
```
***** RATE THE PRESIDENTIAL CANDIDATES *****
INTEGER N, SCORE, HONEST, DRUGS, FOOLS
CHARACTER NAME*12
PRINT*, ' NAME O.K. FAILURES'
DO 10 N = 1, 100
  READ*, NAME, HONEST, DRUGS, FOOLS
  SCORE = 0
  IF (HONEST .LE. 70) SCORE = SCORE + 3
  IF (DRUGS .GT. 40) SCORE = SCORE + 5
  IF (FOOLS .GT. 40) SCORE = SCORE + 7
***** NOW TEST FOR APPROPRIATE CATEGORIZATION *****
  IF (SCORE .EQ. 0) THEN
    PRINT*, NAME, ' *'
  ELSEIF (SCORE .EQ. 3) THEN
    PRINT*, NAME, ' DISHONEST'
  ELSEIF (SCORE .EQ. 5) THEN
    PRINT*, NAME, ' DRUGS'
  ELSEIF (SCORE .EQ. 7) THEN
    PRINT*, NAME, ' PROMISCUOUS'
  ELSEIF (SCORE .EQ. 8) THEN
    PRINT*, NAME, ' DISHONEST & DRUGS'
  ELSEIF (SCORE .EQ. 10) THEN
    PRINT*, NAME, ' DISHONEST & PROMISCUOUS'
  ELSEIF (SCORE .EQ. 12) THEN
    PRINT*, NAME, ' DRUGS & PROMISCUOUS'
```

```

        ELSE
            PRINT*, NAME, ' DISHONEST, DRUGS, & PROMISCUOUS'
        ENDIF
10    CONTINUE
STOP
END

```

This was a somewhat detailed program, but not difficult. You should note the fact, however, that the programmer only had to write the set of instructions *once* and test them out; but once the program is tested and put into production, it can be used to rate hundreds or thousands of candidates, without any need for further human effort (except to enter the scores). Often programs involve detailed testing of a number of different alternatives, and you must be patient in writing them out.



A different example, one which involves nesting a DO loop within an IF block, would be the following. An integer value is read in. If it is a four-digit number, then the sum of its digits is computed and printed out. To extract the digits, we use integer division by successively smaller powers of 10 on the remainder after extracting the previous digit. Thus in the four-digit integer 1234, we see that division by 1000 gives us the most significant digit, 1. If we then take the remainder, 1234 mod 1000, we get 234; then division by 100 will give us the second significant digit, 2; and so on. Once the number has been established to lie in the desired range, the extraction of digits can be seen in the preceding flowchart.

```
***** READ IN AN INTEGER AND, IF 4-DIGIT, SUM ITS DIGITS *****
INTEGER INT, NUM, DIGIT, SUM, I, NDIV
READ*, INT
IF (INT .GE. 1000 .AND. INT .LE. 9999) THEN
    NUM = INT
    SUM = 0
    NDIV = 1000
    DO 50 I = 1, 4
        DIGIT = NUM/NDIV
        SUM = SUM + DIGIT
        NUM = NUM - DIGIT*NDIV
        NDIV = NDIV/10
50    CONTINUE
    PRINT*, 'THE SUM OF THE DIGITS OF ',INT,' IS ',SUM
ENDIF
```

You can readily see how this program could be extended to sum up the digits of any n-digit number, given n. An n-digit number must fall in the range from 10^{n-1} to $10^n - 1$:

```
IF (INT .GE. 10**(N-1) .AND. INT .LE. 10**N - 1) THEN
```

The divisor must begin at 10^{n-1} :

```
NDIV = 10**(N-1)
```

and the loop must execute n times:

```
DO 50 I = 1, N
```

Another related problem is to determine for any integer how many digits it has; this will be left as an exercise.

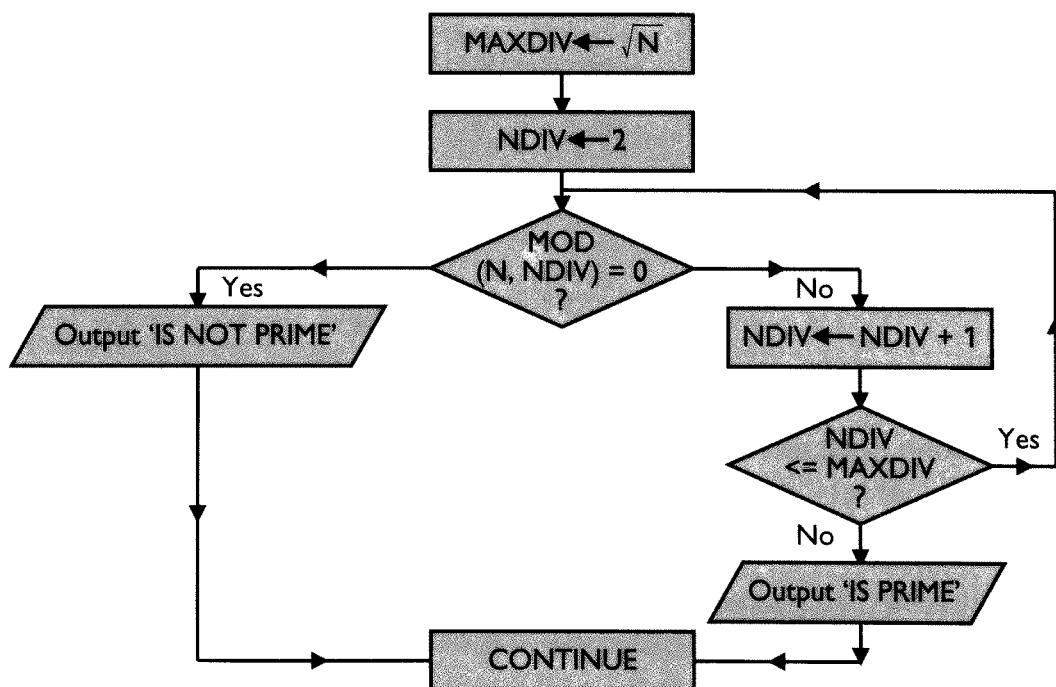
◆ EXAMPLE PROGRAMS

Prime Numbers

An interesting problem, which has applications in encrypted coding, is that of determining whether a given integer is *prime* or not. A prime number is one which has only 1 and itself as exact divisors. Prime numbers are useful in a variety of problems, such as the one where we coded the presidential candidates' flaws, or generating random numbers (to be discussed in a later chapter). If you want to print out the prime numbers in a given range of values, you can test each number in the range for "primality," and print out only those passing the test.

We will not test 1 as a divisor of the number, since all integers are divisible by 1. Thus, the divisors we need to test are those from 1 to the square root of the number. If the number has any divisors (factors) greater than the square root, any such factor had a coordinate factor (since factors come in pairs) less than the square root, so we will have already found it. Such initial problem analysis saves program execution time, since a hasty first look might test all divisors from 2 to $N-1$, thus wasting computer time to no useful advantage.

For a given number N , the test for its primality can be exhibited by the following flowchart. Note that as soon as a divisor of N is found, there is no need to test the other possible divisors, and it is simply declared not to be prime.



The program is as follows:

```
***** TEST FOR PRIME NUMBERS *****
      INTEGER FIRST, LAST, N, NDIV, MAXDIV
***** INPUT THE RANGE IN WHICH YOU WISH TO FIND PRIMES *****
      READ*, FIRST, LAST
      DO 50 N = FIRST, LAST
          MAXDIV = N**0.5
          NDIV = 2
***** REPEAT UNTIL DIVISOR RANGE IS EXHAUSTED (UNLESS) *****
      10  CONTINUE
          IF ( MOD(N,NDIV) .EQ. 0 ) THEN
***** IF THERE WAS AN EXACT DIVISOR, N IS NOT PRIME *****
***** TAKE PREMATURE EXIT *****
              PRINT*, N, ' IS NOT A PRIME'
              GO TO 50
          ENDIF
***** WHILE REMAINDER IS NOT ZERO, KEEP EXAMINING DIVISORS *****
          NDIV = NDIV + 1
          IF (NDIV .LE. MAXDIV) GO TO 10
***** NONE OF THE DIVISORS WORK, SO N IS A PRIME NUMBER *****
              PRINT*, N, ' IS A PRIME'
50   CONTINUE
      STOP
      END
```

When you have studied arrays, you will be assigned a different method of finding prime numbers—the “sieve of Eratosthenes.” In the preceding example, there are actually *two* conditions which control remaining in the loop—that the remainder not be zero and that $NDIV \leq MAXDIV$, and the results are different depending on which test is failed first. Since we have not yet covered logical variables, which you will meet in the next chapter, the simplest way to implement the program was with an early “exit” in case the divisor worked, and the number was seen not to be prime. This involved using a GO TO, within the Repeat/Until structure. We will see how to program this problem more elegantly in the next chapter.

Bouncing Tennis Ball

A tennis ball has a “rebound coefficient” r of 0.8, which means that each time it is dropped, it bounces to a height of 0.8 times the height from which it fell on that bounce. Our problem is to determine how many bounces it takes such that its height on the last bounce is only 1/4 that of the original height from

which it was dropped. Since our knowledge of FORTRAN is not yet sophisticated enough to perform symbolic manipulations to solve this problem algebraically, we will just use an arbitrary initial height of 6 feet (or we could read in any height we liked), and have our program calculate the number of bounces. We will keep the rebound coefficient R as a variable, so that the same program could be run for different rebound factors (newer balls, old beat-up balls). We will maintain one variable for the initial height and another for the bounce height, so that we can compare the two on each bounce.

```
***** BOUNCING TENNIS BALL *****
*      HEIGHT IS THE ORIGINAL HEIGHT FROM WHICH BALL DROPPED
*      BOUNCE IS THE HEIGHT ON EACH NEW BOUNCE OF THE BALL
*      R IS THE REBOUND COEFFICIENT ON EACH BOUNCE
*      NTIMES KEEPS TRACK OF HOW MANY BOUNCES IT TAKES
REAL HEIGHT, QUARTR, R, BOUNCE
HEIGHT = 6.0                      {or READ*, HEIGHT}
QUARTR = HEIGHT/4.0
R = 0.8                           {or READ*, R}
NTIMES = 0
BOUNCE = HEIGHT
***** PROBLEM STATED IN REPEAT/UNTIL FORM *****
55 CONTINUE
      BOUNCE = BOUNCE*R
      NTIMES = NTIMES + 1
      IF (BOUNCE .GT. QUARTR) GO TO 55
      PRINT*, 'IT TAKES ',NTIMES,' BOUNCES TO REACH HEIGHT/4'
      STOP
      END
```

Fibonacci Sequence

The "Fibonacci sequence," discovered by the medieval scholar Fibonacci, or Leonardo of Pisa, in 1202, purportedly traced the growth of pairs of breeding rabbits over successive time periods, assuming that none of the rabbits died. The sequence begins:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

The sequence begins with the pair of values 1 and 1; from there on, each new term is the sum of the two previous terms. Our program is to read in an integer value N, and print out the Nth Fibonacci number. We will solve this problem by direct repetitions, but later we will see it can also be done by recursion, a technique not yet standardly available in FORTRAN.

A brief problem analysis is in order to begin with. If the value of N is 1 or

2, we have no need for a loop, and the result to be printed is simply 1. For $N > 2$, we can construct a DO loop to successively determine the terms in the sequence, up to the N th. We will use the (integer) variables ONE and TWO to stand for the two most recent terms in the sequence. Of course, each time a new term of the sequence (NEW) is computed, the values of ONE and TWO will change as well.

```
***** FIBONACCI SEQUENCE *****
***** COMPUTE NTH TERM OF: 1, 1, 2, 3, 5, 8, 13,.. *****
      INTEGER N, ONE, TWO, NEW, I
***** INPUT VALUE FOR N, WHERE NTH TERM IS WANTED *****
      PRINT*, 'INPUT WHICH TERM OF FIBONACCI SEQUENCE YOU WANT'
      READ*, N
      IF (N .LE. 2) THEN
          PRINT*, 1, ' IS THE ', N, 'TH FIBONACCI NUMBER'
      ELSE
          ONE = 1
          TWO = 1
          DO 50 I = 3, N
              NEW = ONE + TWO
              ONE = TWO
              TWO = NEW
50      CONTINUE
          PRINT*, NEW, ' IS THE ', N, 'TH FIBONACCI NUMBER'
      ENDIF
      STOP
      END
```

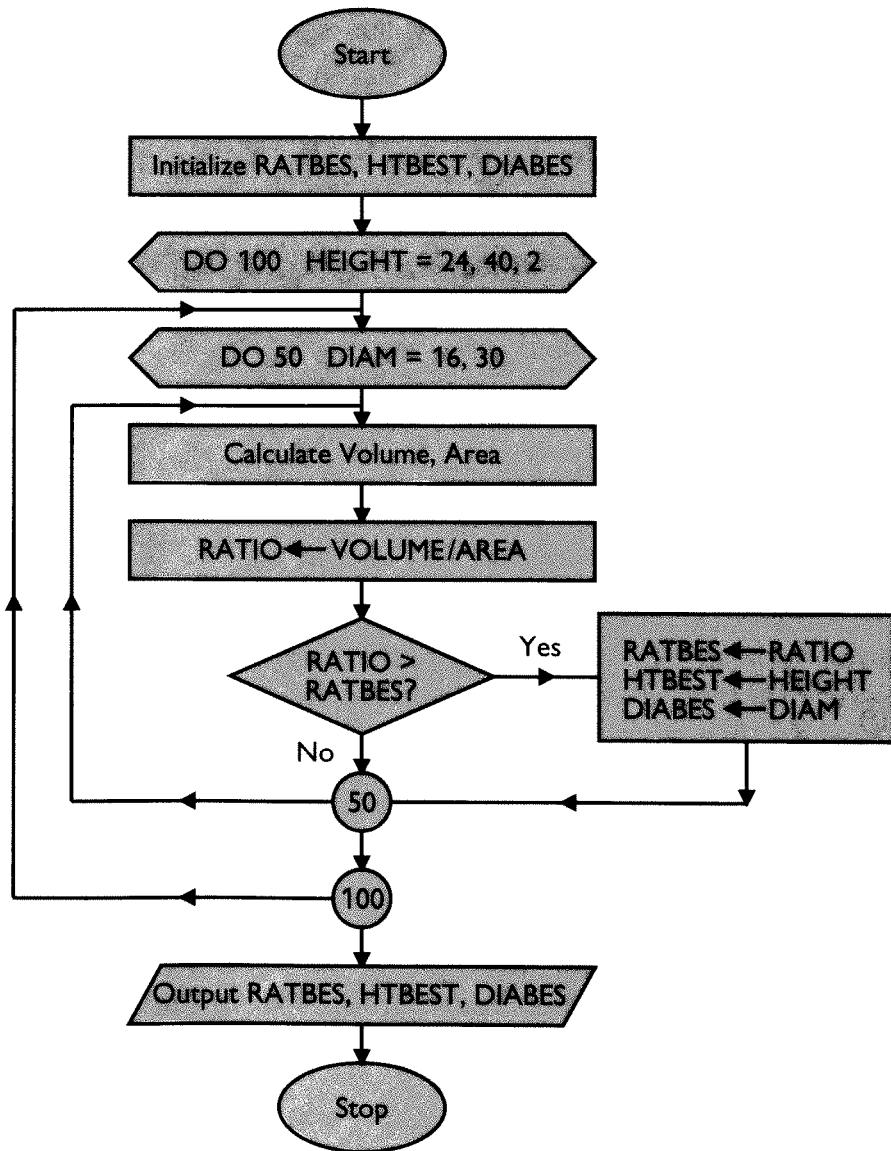
Optimum Wine Cask Dimensions

The volume that a wine cask can hold is a function of its height and its circumference. Though an actual wine cask is somewhat “fuller” at its midway point vertically, for the purpose of this problem, we will assume that it has a perfect cylindrical shape—that is, that its diameter, parallel to the ground, is the same throughout. Now, given certain limitations on possible heights and diameters of the barrels, we would like to determine the optimum dimensions for our wine barrels—that is, the most favorable ratio of volume to surface area (since our cost in constructing the casks will be proportional to the amount of material used, that is, to the surface area).

A problem of just this kind led Johannes Kepler to advances toward the foundations of the calculus. But today we will take a more brute-force approach, and use our computer to help solve the problem for us. Let us assume that our wine cask heights can vary from 24 inches to 40 inches, and

that the diameter of the cask can vary from 16 inches to 30 inches. Let us write a FORTRAN program which will examine all possible combinations of these dimensions, ranging in one-inch differences for the diameters and 2-inch differences for the heights (since the slats we can buy for the sides standardly come in even lengths). We will set up a doubly nested loop to try all these combinations of dimensions. For each combination, we will calculate the volume and the surface area, and then take the ratio. Since we want to maximize this ratio, we will test it each time against the maximum so far; if it is greater, we want to save the new, higher ratio *and* the height and diameter that gave rise to it. By the time we have finished the loops, we will have found the optimum size, given our integer restrictions on height and diameter.

We have already solved a similar problem of finding a largest value in a collection of data, and have even drawn a flowchart for it, so much of our work is already done. Let us just take a quick look at the overview flowchart for our doubly nested loops on heights and diameters, and then we can attack the FORTRAN code itself.



The program itself follows readily from this.

```
***** DESIGN OPTIMUM WINE CASK *****
***** HEIGHT IS CASK HEIGHT, DIAM ITS DIAMETER *****
***** RATBES, HTBEST, AND DIABES KEEP TRACK OF BEST SO FAR *****
      INTEGER HEIGHT, DIAM, HTBEST, DIABES
      REAL RATIO, VOLUME, AREA, RATBES, RADIUS, PI
      PARAMETER (PI = 3.14159)
***** INITIALIZE BEST COUNTERS *****
***** FOR RATIO, HEIGHT, AND DIAMETER TO ZERO *****
      RATBES = 0.0
      HTBEST = 0
      DIAM = 0
      DO 100 HEIGHT = 24, 40, 2
         DO 50 DIAM = 16, 30
            RADIUS = DIAM/2.0
            BASE = PI*RADIUS**2
            VOLUME = BASE*HEIGHT
            AREA = 2*BASE + PI*DIAM*HEIGHT
            RATIO = VOLUME/AREA
            IF (RATIO.GT.RATBES) THEN
               RATBES = RATIO
               HTBEST = HEIGHT
               DIABES = DIAM
            ENDIF
   50      CONTINUE
100     CONTINUE
      PRINT*, 'THE OPTIMUM HEIGHT IS ', HTBEST, ' INCHES AND'
      PRINT*, 'THE OPTIMUM DIAMETER IS ', DIAM, 'INCHES'
      PRINT*, 'YIELDING A VOLUME/AREA RATIO OF ', RATBES
      STOP
      END
```

Notice that it was not enough to keep track of just the best ratio; we also needed to save the height and diameter that gave rise to it.



PREVIEW TO SUBROUTINES

Another form of repetition can be gained through the use of *subprograms*—that is, functions which will perform a single task such as taking a square root, as often as you like, and *subroutines*, which will perform a group of operations, and can be executed many times. Though the full use of subroutines and

functions is rather complex, it will be valuable to give you a preview of the power they can provide your programs at this point. We will take a very simple introduction to subroutines, to encourage you to begin thinking in terms of *modularizing* your programs.

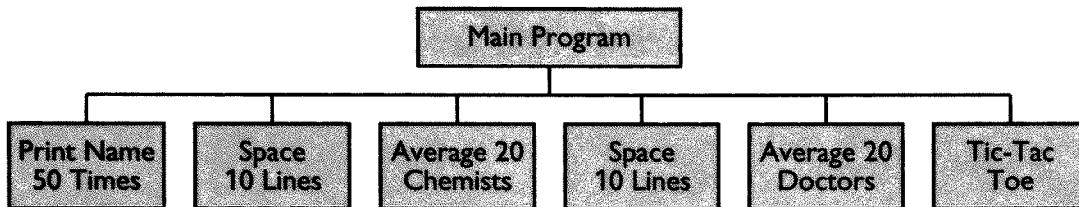
We have already stressed the importance of top-down design of your programs, using a divide-and-conquer approach to break your larger problems up into manageable parts. The structure charts, or HIPO diagrams, we looked at in Chapter 1 took a block-structured approach to such problem decomposition. If these program “blocks” can be developed relatively independent of one another, they can be designed as separate subprograms, which are then activated and tied together by the “main program” (the block at the top of the structure chart).

The simplest case we can begin with is one in which the action of each subprogram is totally independent of any information in the other blocks (including the main block). Thus each can be designed as a totally independent unit, tested, and “called” by the main program at the appropriate point. The structure of such a *subroutine* subprogram (we will later look at functions as another type of subprogram) is, first, that it begins with an identifying line, of the form:

SUBROUTINE name [arguments]

The *name* of the subroutine must follow the standard naming conventions in FORTRAN, and should not be the same as the name of any other “global” entity—that is, it should not be the same as any other subprogram name, or any name of a labelled area of COMMON (to be discussed in Chapter 9), and it should not be the same as the name of any variable appearing in this subprogram. The subroutine program unit may then contain a set of FORTRAN instructions of the kind we have been writing all along (it is a kind of “mini-program”). It should contain at least one RETURN statement, and the last statement in the program unit must be an END statement. The END statement is a flag to the compiler that this is the end of this program unit; since all program units get compiled separately and independently, the compiler must have a way to know where each one begins and ends.

Imagine you want to write a simple program that (1) first prints out your name 50 times down the page, (2) then skips 10 lines, (3) then reads in 20 salaries of chemists and calculates and prints the average chemist salary, (4) then skips 10 more lines, (5) then reads in the salaries of 20 doctors and calculates the average doctor salary, and finally (6) prints out a tic-tac-toe board. This is an odd assortment of tasks, but for some weird reason you want them all run as one job (and, given that we want to keep the sections totally independent of one another, and that we have not yet covered formatted I/O or arrays, we need a relatively simple-minded example here). Note the hierarchical structure of the job:



Note that the second block and the fourth block are identical, and the third and fifth blocks are closely related; we can take advantage of these similarities in our programming. Let us begin writing subprograms to do the jobs. First, printing your name:

```

SUBROUTINE NAMES
DO 60 I = 1, 50
      PRINT*, '          MARIE CURIE'
60 CONTINUE
RETURN
END
  
```

Next, a subroutine to space ten lines:

```

SUBROUTINE SPACES
DO 60 I = 1, 10
      PRINT*
60 CONTINUE
RETURN
END
  
```

Now we can write a subprogram that will read in 20 salaries and calculate the average:

```

SUBROUTINE SALARY
REAL SUM, SAL, AVER
SUM = 0.0
DO 75 I = 1, 20
      PRINT*, 'ENTER A SALARY, AS A REAL NUMBER'
      READ*, SAL
      SUM = SUM + SAL
75 CONTINUE
AVER = SUM/20.
PRINT*, '$', AVER
RETURN
END
  
```

Now we can tackle the tic-tac-toe board:

```

SUBROUTINE TICTAC
CHARACTER A, B*17, SPACES*5
SPACES = ' '
A = '|'
B = '_'
DO 8 I = 1, 2
    DO 5 J = 1, 3
        PRINT*, SPACES, A, SPACES, A
5     CONTINUE
        PRINT*, B
8     CONTINUE
    DO 10 I = 1, 3
        PRINT*, SPACES, A, SPACES, A
10    CONTINUE
    RETURN
END

```

Now that we have our subprograms, we can put them together into one job; they will be tied together by a main program which will "call" each subroutine as it needs it. The way a subroutine is referenced is by a statement of the form:

CALL name

where *name* is the name you have given the subroutine. Thus we can now write our main program. For now, "trust me" that the RETURN in each subroutine will bring you back to the calling program at the point *right after* the CALL; this will be explained in greater detail in Chapter 9 on subprograms in general.

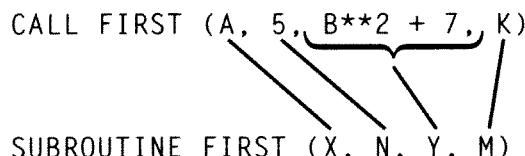
```

***** MAIN PROGRAM - SUBROUTINE DISPLAY *****
CHARACTER PROMPT*43
CALL NAMES
CALL SPACES
PROMPT = 'YOU WILL BE PROMPTED TO ENTER SALARIES FOR '
PRINT*, PROMPT, 'CHEMISTS'
PRINT*
PRINT*, ' THE AVERAGE CHEMIST'S SALARY IS'
CALL SALARY
CALL SPACES
PRINT*, PROMPT, 'DOCTORS'
PRINT*
PRINT*, ' THE AVERAGE DOCTOR'S SALARY IS'
CALL SALARY
CALL TICTAC
STOP
END

```

All of these program units should be entered as part of one workfile (in older batch systems, these were all part of one card deck submitted as a unit, between job control cards), then submitted for compilation and execution. Usually, the main program is entered first, but this is not necessary. The order in which the subroutines are entered does not have to match the order in which they are called. Notice that we were able to use two of the subroutines we wrote twice. This is one of the advantages of subroutines; complex code to do a job need only be written once, but it can be executed many times, from different parts of your program, or else it can ever be "ported" for use in another program (yours or that of a friend).

One additional note, to widen your knowledge regarding subroutines a little more before we leave this chapter. The subroutines we wrote were completely independent of one another and of the main program (except that they had to be called by the main program in order to execute; they cannot initiate their own execution). What if we wanted to share some information with the subroutine? For example, the SPACES subroutine could be more general if we allowed it to print a variable number of blank lines. One way to do this is to give the subroutine one or more *arguments*, which appear in a list following the subroutine name. On the SUBROUTINE line, this list must be made up of "dummy variables," that is, variables which will take their identities from the values (constants, variables, or expressions) on the *actual argument* list in the subroutine CALL. This coordination of arguments works as follows:



Thus the dummy variables in the subroutine get their identities from the actual argument list in this CALL as follows: X is identified with the variable A, N gets the constant value 5, Y gets the value of $B^{**}2 + 7$, and M is identified with the variable K in the calling program. These lists have to agree in length, and the types of the dummy arguments must agree with the types of the actual arguments they are lined up with.

Thus we could modify our SPACES subroutine to print *any* number of blank lines, N, by the following changes:

```

SUBROUTINE SPACES (N)
DO 60 I = 1, N
    PRINT*
60 CONTINUE
RETURN
END
  
```

and calls to this subroutine could take the form:

```
CALL SPACES (15)
CALL SPACES (MANY)
CALL SPACES (MINK*2 + MINE)
```

(as long as the integer variables MANY, MINK, and MINE had been defined in the calling program prior to the subroutine calls).

As one more useful subroutine example, write a subroutine which will interchange the contents of any two integer variables:

```
SUBROUTINE SWITCH (M, N)
MTEMP = M
M = N
N = MTEMP
RETURN
END
```

This could then be called from another program unit by:

```
CALL SWITCH (MAX, MAD)
CALL SWITCH (MUTT, JEFF)
```



FORTRAN 90 FEATURES

The relevant innovations in Fortran 90 for this chapter concern the DO loop. First, it should be noted that the use of real and double precision DO parameters will be continued in Fortran 90, but is considered *obsolete*, and discouraged.

DO loops may omit the reference to a statement number after the DO, and be of the form DO . . . END DO (or ENDDO).

A DO loop may have a name; if it does have such a name, the END DO statement should also include the name; for example:

```
NAME: DO V = I, F . . . END DO NAME
```

An EXIT statement may be used within a DO loop to terminate execution of the loop and transfer control out of the loop.

A CYCLE statement may be used in a DO loop to prompt the execution of the next stage (cycle) of the loop at that point.

A DO statement may be used which specifies no loop parameters; this is an "indefinite" loop, which then should include an EXIT statement to insure termination.

A DO WHILE form is available, which sets up a pretest loop:

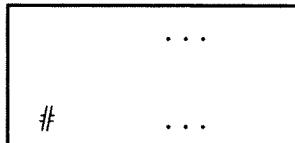
```
DO WHILE (log. cond.) . . . END DO
```



SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

FORTRAN has one basic built-in repetition structure, the *DO loop*. It has the general form:

```
DO # var = init, final [,step]
```



The set of statements following the DO through the numbered statement (#) referenced in the DO is called the *range* of the loop. The variable *var* in the DO statement can be any integer, real, or double precision variable, and it is the *index* that controls the repetition of the loop: it begins at the value of the DO parameter *init*, and proceeds in steps of *step* until it exceeds the upper limit specified by *final* for the loop. The DO parameters *init*, *final*, and *step* may be integer, real, or double precision constants, variables, or expressions. If a value for *step* is not specified, the default value is +1. The *terminal statement* of the DO loop (usually marked by the statement number #) can be any executable statement except a GO TO, arithmetic IF, block IF, ELSE, ELSEIF, ENDIF, RETURN, STOP, END, or another DO. It is generally desirable to terminate a DO on a neutral CONTINUE statement, and indent the block of instructions in the DO loop.

You may transfer out of the range of a DO loop, but not into the range. The value of the DO loop index *var* must not be changed during the range of the DO, and the other parameters should not be changed. If DO loops are *nested*—that is, if one [outer] loop controls the execution of another [inner] loop, they must be properly nested, such that the range of the inner loop is entirely within the range of the outer loop. Similarly, if a DO is used in conjunction with a block IF-style control structure, the outer structure must completely contain the range of the inner structure—that is, their ranges must not overlap.

The DO loop structure in FORTRAN 77 is a *prechecked* loop—that is, when the DO is encountered, the number of iterations of the loop is calculated, using the initial and final values and the step size; if the number of iterations is positive, the loop is executed that many times, but if it is negative or zero, the loop is skipped entirely. The *While* structure we simulated is another example of a *pretest* loop; the *Repeat Until* structure is a *posttest* loop. All three of these structures are useful in FORTRAN, and a good programmer is alert to making the best choice of loop structure for the problem at hand.

A *subprogram*, which may be either a *subroutine* or a *function* in FORTRAN, is a way to modularize your program so that certain repetitive sequences of instructions need only to be written once, and then properly referenced, and they also aid in developing your program in a top-down fashion, solving smaller problems one at a time, and then finally tying them together. Subprograms are discussed fully in Chapter 9. To begin writing “mini-programs,” or subroutines, though, all that is needed is the way to structure one (entry, body, return), and then reference it by a CALL statement.

A *subroutine* has the following structure in FORTRAN:

```
SUBROUTINE name [(dummy variable list)]
    {block of instructions composing the subroutine}
    RETURN
    END
```

and it is *called*, from the main program or another subprogram, by a statement of the form:

```
CALL name [(list of actual arguments, matching dummy list)]
```

Note that the list of calling arguments, and the matching dummy variable list in the SUBROUTINE line, are optional; they are used if it is necessary to transfer information back and forth between the calling program and the subroutine.



EXERCISES

1. Write a FORTRAN program which will read in an integer value of N (up to 12) and print out N factorial ($N!$). You drew a flowchart for this problem earlier. Use a DO loop. We limited the value to 12, since that should work on any of our large machines. However, on a 32-bit machine, $13!$ or larger would cause an integer *overflow* (a value too large to fit in one word in memory). Experimentally try a sequence of N's for which you want to determine factorials, to find the first value which causes overflow on your machine.
2. Write a FORTRAN program which adds the odd integers from 301 through 999 and prints out the sum. Use a DO loop.
3. Write a FORTRAN program which adds up the sums of the squares of the first 50 integers. Write a program which sums the squares of the integers until the sum exceeds 1200; print out the sum and the number of terms added.

- **4.** Are any of the following sequences illegal in Standard FORTRAN 77? If so, indicate why they are illegal.

```

1) DO 3 I = 1, 5      2) DO 7 K = 5, 2      3) IF(J)20,20,30
   DO 3 J = 2,6,2          7      PRINT*, K      DO 20 I = 1,10
   K = I**J-2              20      PRINT*,I,I**2
   PRINT*, K              30      PRINT*,'OVER'

3 CONTINUE

```

- 5.** What value will be stored in location K at the end of each of the following sequences? Assume default typing.

```

1) K = 0           • 2) DO 30 J = 1, 4   3) DO 7 K=1,100,8
   DO 10 J=1,3           K = 0           7      PRINT*, 'I AM LYING'
   DO 10 N=J+1,4       DO 30 N=1,5      PRINT*, K
10      K = K + J*N   30      K = K + N
   PRINT*, K           PRINT*, K

```

- 6.** Write a FORTRAN program which will print out the three-digit sets from 0,0,0 through 9,9,9 in order. Use triply nested DO loops.

7. Use the loops you set up in problem 6 to determine which three-digit integers are equal to the sums of the cubes of their digits, and print them out. Include 0 (000) and 1 (001), since they both satisfy the criterion. The next number you should find and print out is 153 ($153 = 1 \times 1 \times 1 + 5 \times 5 \times 5 + 3 \times 3 \times 3$).

- **8.** Modify the “bouncing ball” program so that the “rebound coefficient” is only 98% of its last value on each succeeding bounce (after all, the ball gets less resilient as it bounces).

9. Write a program which will test for “perfect numbers” in the range from 5 to 500. A perfect number is one that is equal to the sum of all its factors, including 1 but not including the number itself. Thus 6 is a perfect number because its three factors, 1, 2, and 3, add up to 6.

10. Your bank is advertising a 5.5% yearly rate on money you deposit with them. Compare how much money you would earn on a \$1000 deposit if this interest were paid yearly, monthly, weekly, daily, or “continuously.” Note that if the interest is compounded over a period shorter than a year, the interest for that period is correspondingly less; for example, if compounded

monthly, each month's interest is $(5.5/12)\%$. In general, for a rate r (expressed as a decimal equivalent of percent) per time period for a principal P , compounded over n time periods, the total money accumulated is:

$$M = P(1 + r)^n$$

If r is computed from a yearly rate y , as the number of time periods increase, your multiplication factor:

$$(1 + y/n)^n$$

also increases, but has an upper limit. That is, the limit of the expression:

$$(1 + y/n)^n \text{ as } n \text{ approaches infinity}$$

is e^y , where e is the base of the natural logarithms ($e = 2.71828\dots$). Thus, if your investment is compounded *continuously* at a yearly rate y (expressed as decimal equivalent of %), your total investment at the end of the year is:

$$M = Pe^y$$

Note: to take e^x in FORTRAN, use the system function EXP(X).

11. The formula given in the last problem for interest compounded continuously also applies to any exponential growth (or decay) situation. In general, the new population N (of rabbits, humans, bacteria, or whatever is being studied) of an original population P where the growth rate is *exponential*—that is, the growth rate at any instant is proportional to the population at that instant, with a growth rate k over a time period t is

$$N = Pe^{kt}$$

The world population in 1985 was 4845 million, with a growth rate of 1.7%. How much will it be in the year 2000? In 2020? How long will it take to double? What if the annual rate of increase is 1.8%?

- **12.** Jimmy has an account with an organization which guarantees to double his money every two years. Write a program to determine how long, if these promises held true, it would take Jimmy to become a millionaire if he started with \$10.

- 13.** You are given two job offers on graduation. One offers you a starting salary of \$20,000 and a guaranteed 8% raise every year. The other firm offers you a starting salary of \$23,000 and a raise of \$1500 every year. After 6 years,

at which firm will you be making a higher salary? Also compute your accumulated gross earnings over those 6 years and compare.

- ◆ **14.** The horizontal range of a projectile fired at an angle A, since the force of gravity tends to pull it down, is:

$$\text{range} = v^2 \sin(2A/g)$$

where v is the velocity of the projectile and g is the acceleration of gravity, 9.8 meters/second². Assuming a projectile velocity of 90 meters/second, write a DO loop that will print out a table of distances achieved by the projectile for angles of inclination A from 10° to 80°. Use the sine function on the system—i.e., SIN(X), where X is an angle expressed in *radians*. To convert degrees to radians, remember that 2 pi radians = 360°.

- **15.** Billy Batson has invented a machine he hopes will make him a millionaire; it is a super-duper popcorn popper. His company can produce 1000 units a day. On the first day he is in business, he receives 24 orders; the next day, he receives three times as many orders; and on each succeeding day (including weekends), the order rate triples. How many days does it take for his production capacity and backup to be matched (or exceeded) by the orders? Print out a table day by day of the day number, the total number of orders, and the production to that point. Choose your loop structure wisely.

- 16.** Write a FORTRAN program to compute

$$1 - 2 + 3 - 4 + 5 - 6 + \dots - 1000$$

Do this in a loop, without using any input.

- 17.** Recall the “monthly payment” formula discussed earlier. Use it to print out payment amounts for loans from \$4000 to \$12,000 in steps of \$2000, at rates of interest from 7% to 12% (in steps of 1%), for three different time periods: 3, 4, and 5 years. Make your output informative. This will require triply nested loops. Also calculate, for a given loan and interest combination, the different totals paid out over 3-, 4-, and 5-year loan periods. Which is most desirable?

- 18.** Assume that a spherical shape is the ideal in nature. Calculate a table of the ratio of surface area to volume for spherical biological creatures with radii ranging from 1 unit to 20 units, in steps of 1 unit.

- **19.** A population of rabbits increases at a rate of 5% a year, and a population of foxes increases at a rate of 2% a year. If you assume that the sizes of the two populations are independent, then examine the comparative populations

over a 10-year period (output in a table), if you begin with a population of 100 rabbits and a population of 300 foxes.

Now assume that there is attrition in the rabbit population which is a function of the size of the fox population for that year. This attrition decreases the rabbit population by N rabbits a year, where N is calculated as $.07 * \text{current fox population}$ (truncate or round fractional values to nearest integer). Factor this in, and redo your original table.

20. If the world population in 1977 was roughly 4083 million people and it was increasing at a rate of 1.8% a year, write a FORTRAN program to calculate how many years it will take, at this rate, for the world population to double. If the land surface of the planet is 57,821,000 square miles (this estimate includes, unfortunately for our purposes, inland water and Antarctica), given the rate of growth predicted and the population in 1977, by what year will the population of the planet be such that each person has only 20 square feet of land surface to stand on? By what year will there be only one square foot of land surface per person?

21. It has been suggested that it might be desirable to have tunnels straight through the earth between major cities, instead of having to travel on the (curved) earth's surface. Assume that the earth is (roughly) a sphere, with a radius of about 6.368×10^6 meters (a kilometer = 0.6214 miles). Set up a program to compare the straight-line distance through the earth between cities whose arc surface distance is 500, 1000, 1500, 2000, up to 5000 miles apart (in steps of 500 miles). At what point might a tunnel seem worth the effort involved in the distance saved?

• **22.** For any integer value read in, determine how many significant digits it contains.

23. To exercise your newly developed subroutine skills, write a subroutine which will calculate and print out the gravitational force between any two bodies, given their masses (in kilograms) and the distance between them (in meters). Refer to the example program near the end of Chapter 2 for this.

24. Due to the properties of light in this situation, the angle between your eye level and the highest point of the arch of a rainbow is $42\frac{1}{3}$ degrees. Thus if you know how far away you are (horizontally) from the rainbow, you can calculate its height. Use the system TAN function, and remember that it takes the tangent of an angle expressed in *radians*.

25. *Bode's Law* in astronomy (which he formulated in 1772) is obtained by taking the numbers 0, 3, 6, 12, 24, 48, 96, 192, 384 (each number after 3 is

double the preceding number), adding 4 to each of these numbers, and then dividing the result by 10. The resulting values represent mean distances from the sun of various bodies (mostly planets), expressed in *astronomical units* (where 1 astronomical unit is the earth's mean distance from the sun, that is, 93 million miles). Generate these values, and then use them to print out a table of the Bode's Law values compared with the following actual astronomical distances (note that Ceres, an asteroid accidentally discovered by Piazzi in 1801, fills in at the distance of 2.8 units, and the average distance of hundreds of asteroids discovered since then is roughly the same 2.8 units).

Planet Distances from the Sun in Astronomical Units

Mercury – 0.39	Venus – 0.72	Earth – 1.0	Mars – 1.52
Ceres – 2.77	Jupiter – 5.2	Saturn – 9.54	
Uranus – 19.19	Neptune – 30.07	Pluto – 39.46	

Note: Pluto may have been a satellite of Neptune which escaped.

26. *Line Spectra (Quantum Mechanics).* Examination of the spectrum of atomic hydrogen shows the existence of spectral series as indicated, named after their discoverers:

Lyman series	$f = cR (1/1^2 - 1/n^2)$ n = 2,3,4,... ultraviolet
Balmer series	$f = cR (1/2^2 - 1/n^2)$ n = 3,4,5,... visible
Paschen series	$f = cR (1/3^2 - 1/n^2)$ n = 4,5,6,... infrared
Brackett series	$f = cR (1/4^2 - 1/n^2)$ n = 5,6,7,... infrared
Pfund series	$f = cR (1/5^2 - 1/n^2)$ n = 6,7,8,... infrared
Humphreys series	$f = cR (1/6^2 - 1/n^2)$ n = 7,8,9,... far infrared

In these formulas, c is the speed of light (3.0×10^8 m/sec), and R is the *Rydberg constant* (1.0967758×10^7 /m). Write a FORTRAN program to generate a specified number of the frequencies of the line spectra for any one of these series (specified by name).

27. The value of "machine epsilon" for a particular computer is the smallest (real) value greater than zero that it can represent. Though one could

determine this from a knowledge of the word size on the machine and its convention for representing reals, let us determine it experimentally instead. Begin with a real value of 1.0, and continue dividing it by 2, printing out every 10th value, until the value goes to 0.0. Keep track of the point at which it went to zero, so that you can print out the *last* value before this point, which should be the machine's epsilon. There is another epsilon value, *e*, which is the smallest value which when added to 1.0 gives a value greater than 1.0 (it is larger than machine epsilon; do you know why?). Beginning with the value of machine epsilon, increase it by successive multiplications of 2 until when added to 1.0 it gives a value greater than 1.0; print out this value for *e*.

28. A bobsled run for the 1992 winter Olympics is 150 meters high. Assuming that there is no friction, and using the law of conservation of energy, calculate and print out the speed of the bobsled at 10-meter intervals (vertically) as it speeds down the run. [Note: its total energy will be conserved; its potential energy is mgh , where h is its height above the lowest point, and its kinetic energy is $1/2 m v^2$.]

29. The "golden mean" is said to be the ratio of lengths of the sides of the most aesthetically pleasing rectangle, and was used by the ancient Greeks in their architecture. It is

$$1/2 (\sqrt{5} - 1)$$

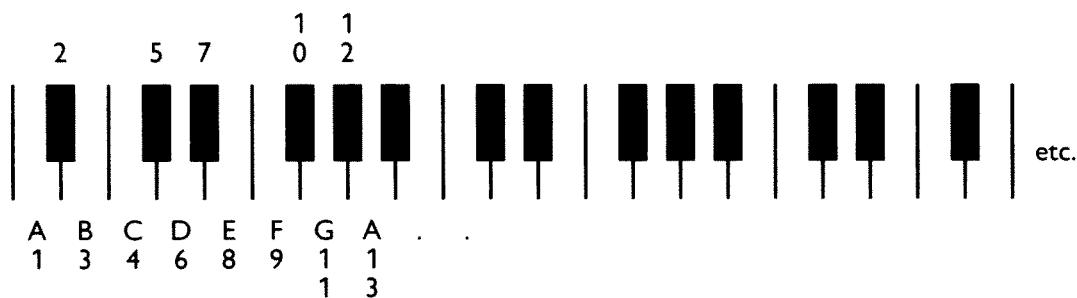
and is also the limit (as n approaches infinity) of two successive Fibonacci numbers, F_n and F_{n+1}):

$$\text{golden mean} = \lim (F_n/F_{n+1}) = 1/2 (\sqrt{5} - 1)$$

Test out how the ratio of successive Fibonacci numbers converges by generating the first 100 numbers of the sequence (refer to the example in the chapter), printing out the ratios of each successive pair, and comparing to the value of the golden mean.

30. A piano keyboard has 88 keys—seven repeating patterns of 12 keys (7 white keys—A, B, C, D, E, F, and G—and 5 black keys, as shown in the picture) and 4 additional keys at the right (A, A# {A sharp, or B flat}). The lowest key (at the left) is an A, and produces a frequency of 27.5 cycles per second when its hammer hits the string. Two closest keys of the same name are said to be an *octave* apart, and the frequency of the higher is twice that of the nearest lower. The octave is divided equally into twelve tonal steps, so the frequency of any key (except the first) is $\sqrt[12]{2}$ times that of the lower adjacent key. Thus,

if you know the frequency of the lowest key, you can determine the frequency of each key on the keyboard. Write a DO loop to print out the frequencies of each key, numbered 1 to 88; to be more elegant, give each key a name as well (B, C#, etc.).



CHAPTER 5



DATA TYPES IN GENERAL

What *kinds* of values can your FORTRAN programs handle? This is a very important question, because it will determine what kinds of problems you are able to solve in FORTRAN. The simple, first answer is that FORTRAN has basically six data types, which include integers, reals, and character strings (the three you will use most often). This chapter introduces all six FORTRAN data types in detail, so that you will be comfortable using them, and points toward Chapter 13, which will show you ways to create additional kinds of data structures as you need them.

IBM, Burlington, Vermont.
Computer generated plot on
paper of Dynamic Random-
Access Memory Chip
(DRAM). 1982.

"Characters of the great Apocalypse, The types and symbols of Eternity, Of first, and last, and midst, and without end."

- William Wordsworth, "The Simplon Pass"

There are several different kinds of entities that can be represented in the computer under FORTRAN. There are various numeric values, character strings, and Boolean True and False values. These represent the standardly available *data types* in the language, and the programs you construct will—in general—operate on these types of information. Since a location in memory contains only a string of bits, the program must know, in each case, how it is to interpret these bits. Thus it is crucial for the program to have identified the type of each variable location in which it will store a value. Such identification is done in FORTRAN by using a type statement, or by default.



TYPE STATEMENTS

We have already briefly mentioned type statements. These declare the type of a variable or list of variables in your program. They are of the general form:

type listofvariables

where type may be INTEGER, REAL, LOGICAL, CHARACTER, COMPLEX, or DOUBLE PRECISION. Type declaration statements must appear at the beginning of your program, before any executable statements. If a variable used in your program is not declared in a type statement, it is given its implicit type according to the built-in default typing rules in FORTRAN: variables beginning with I through N (I-N) are type INteger; the rest are, by default, reals.

Thus the use of type statements in FORTRAN does not guarantee that any variable name *not* declared in a type statement will be identified to be in error, since it will simply be assigned its default type. Most entities you use in your programs will be either integers or reals, and they may either be declared explicitly or you may use the default naming convention. Any other types of variables *must* be explicitly declared.

We will briefly examine the various data types in FORTRAN, to see how they differ, and the values they can represent.



INTEGERS AND REALS

As already indicated, most values you process in your programs will be simple numerics, either integer or “real” (floating-point) values. Since there may be

many of these involved in one program, it was for this reason that the original FORTRAN designers allowed a default naming convention for these commonly used variables, to save programmer time and effort. However, more recent programming practice encourages the explicit typing of *every* program variable at the beginning of a program, as part of good development. Since several generations of FORTRAN programmers are accustomed to the "I-N for INteger, the rest for Real" rule, it might be a good idea to adopt these conventions whenever reasonable in naming integer and real variables, even if you have explicitly typed them, to avoid confusion. However, this should not be carried to the point of giving awkward or non-meaningful names to variables.

The magnitude of the numeric values you can deal with is a function of the word size on the computer you are using. For your reference, the following limitations on integers are common:

INTEGERS

Computer Word Size	Integer Range
16-bit micro	-32768 to 32767
32-bit machine (IBM, VAX, etc.)	-2147483648 to 2147483647
48-bit (Burroughs)	± 549755813887
60-bit machine (CDC)	± 576460752303423487
64-bit supercomputer (Cray)	± 9223372036854775807

When you must deal with *very* large integers, which exceed the memory word capacity of your machine, you must adopt more sophisticated techniques for dealing with them, such as storing them in several locations (in an array); we will discuss such techniques at a later time. The size and precision of *real* values you can store also is dependent on the word size of your computer, and how it is utilized. Examples of these limitations were discussed in Chapter 2.

You must carefully distinguish integer from real values in your program, since a real is stored in multiplier and exponent form, and so is a very different representation from an integer bit pattern. For example, on most machines, the bit pattern for the integer value 5 (101 in binary) is radically different from the representation of the real value 5.0, so you must be clear which value you intend to deal with. This is particularly a problem when you are passing arguments to subprograms, since a variable passed from the calling program may be of one type, and be treated as another type in the subprogram, causing errors. Generally, values for *counters* are integers. On many machines, integer arithmetic is performed more efficiently than real arithmetic, so if your problem can be represented in terms of dealing with integer values, it should be. Reals are used to represent measurements, or many constants of nature, such as the acceleration due to gravity on this planet (32.174 feet per second²), or Avogadro's number (6.02×10^{23} molecules in a mole—an amount of a substance whose weight in grams is the same as its atomic/molecular formula

weight), or pi (3.14159...), and the like. The proper choice of type of variable is important. We have seen that only real values will be stored in an assignment to a real location, and similarly for integers, so that storing in a real may involve "floating" an integer value, and storing in an integer variable may involve "truncating" a real value before storage. This conversion takes computer time, so if it can be avoided, do so (e.g., write N = 6 instead of N = 6.0).



LOGICALS

You have already become acquainted with logical relations, such as (A .GT. B), which are used in various IF control structures. Such a logical relation has a *truth value*, that is, a value of either True or False. If the value is True, one path is taken in the program logic; if it is False, a different path is taken. These truth values are clearly binary in nature, and as such could be represented in a single bit. More often, a logical value is stored in a byte, or even in a word, in memory. You can store a logical value in a location, just as you can save the result of a numerical calculation. The only thing is that the location must have been designated type LOGICAL before you can store a logical value in it.

- Once a location has been declared type LOGICAL, it can store a .TRUE. or .FALSE. through an assignment or a DATA statement, it can have a value read into it, or it can be assigned the truth value of some logical relation. The following are all legal:

```

— LOGICAL X, Y, Z
  READ*, A, B
  X = .TRUE.
  READ*, Y
  Z = A .LE. B

```

The most common use of a LOGICAL variable is to contain a truth value which will be used in the program to control conditional operations. For instance, you may want certain PRINT statements in your program to be executed only if the user asks for them. Thus the user could be asked if printed output of a certain kind was desired, the response used to set a logical variable, and that logical variable used to control the PRINT statements, as sketched in the following program segment:

```

LOGICAL OUT
CHARACTER ANSWER*3
PRINT*, 'DO YOU WANT PRINTED OUTPUT OF THE PLAYS?'
READ*, ANSWER
OUT = .FALSE.

```

```
IF (ANSWER .EQ. 'YES') OUT = .TRUE.
IF (OUT) PRINT*, . . .
```

We can now use a LOGICAL variable to get rid of one of the annoying GO TOs in our prime number program of the previous chapter (we will, however, have to keep the other to retain the "Repeat/Until" structure for the problem). As you recall, the program tests divisors of a candidate number until either a divisor works (in which case the number is *not* a prime), or we have tried all the possible divisors (in which case it *is* a prime), whichever comes first. Since we do not want to continue testing divisors after we have found one that does divide exactly, we will have to use a compound test in the loop—we continue it as long as the next divisor is less than or equal to the maximum divisor *and* no divisor has worked. If this were used as a pretest condition, in a while-type structure, or as a posttest condition in a FORTRAN-simulated repeat/until structure, we would express it as:

```
IF (NDIV .LE. MAXDIV .AND. PRIME) ...
```

where PRIME is a logical value which is .TRUE. as long as the number has still not been disproved as a prime, .FALSE. otherwise. As another way of looking at it, in pseudocode, we would write it as

Repeat {body} Until (NDIV > MAXDIV or not-PRIME)	{Representation is in <i>pseudocode</i> }
--	--

that is, the form of the condition would be

```
{Until}       (NDIV .GT. MAXDIV .OR. .NOT. PRIME)
```

But we have seen that, in our FORTRAN simulations of Repeat/Until structures, the IF at the foot of the loop tests the *opposite* of the Repeat/Until condition. By DeMorgan's Laws, our test becomes

```
IF (NDIV .LE. MAXDIV .AND. PRIME) GO TO 10
```

which verifies the form of the test we arrived at earlier. Incorporating this into our earlier program, we get:

```
***** TEST FOR PRIME NUMBERS (REVISED) *****
INTEGER FIRST, LAST, N, NDIV, MAXDIV
LOGICAL PRIME
***** INPUT THE RANGE IN WHICH YOU WISH TO FIND PRIMES *****
READ*, FIRST, LAST
DO 50 N = FIRST, LAST
  MAXDIV = N**0.5
```

```

NDIV = 2
PRIME = .TRUE.

***** REPEAT UNTIL DIVISOR RANGE IS EXHAUSTED OR NON-PRIME *****
10    CONTINUE
***** IF THERE WAS AN EXACT DIVISOR, N IS NOT PRIME *****
***** SET INDICATOR FALSE *****
        IF ( MOD(N, NDIV) .EQ. 0 ) PRIME = .FALSE.

***** WHILE DIVISORS ARE IN RANGE, KEEP EXAMINING DIVISORS *****
***** UNLESS IT HAS ALREADY BEEN SHOWN TO BE NON-PRIME *****
        NDIV = NDIV + 1
        IF (NDIV .LE. MAXDIV .AND. PRIME) GO TO 10
***** IF NONE OF THE DIVISORS WORKED, N IS A PRIME NUMBER *****
        IF (PRIME) THEN
            PRINT*, N, ' IS A PRIME'
        ELSE
            PRINT*, N, ' IS NOT PRIME'
        ENDIF
50    CONTINUE
END

```

This program, though no shorter than the earlier version, is much more structured in appearance.

CHARACTERS

FORTRAN 77 has incorporated the CHARACTER data type into its compiler. Prior to that, character strings in FORTRAN might be stored into integer, or sometimes real, locations, and the length of the string that could be stored was a function of the combination of the byte size needed to store a character (6 or 8 bits) and the word size of the particular machine. This led to great incompatibilities among FORTRAN programs that used characters, and so the matter was remedied in FORTRAN 77 by standardizing the handling of character strings at the programmer's level. The actual implementation of character storage is machine-dependent, but it is transparent to the FORTRAN programmer.

A variable to contain a character string must be declared in a CHARACTER type statement at the beginning of the program, along with its maximum length. This may be done in one of two ways:

- CHARACTER*n listofvars
- {or} CHARACTER var₁*n₁, var₂*n₂, ...

where the n's represent the lengths of character variables. In the first example, all of the variables on the list will be of length n (unless specific exceptions are made); in the second case, each variable on the list is given its own length, and if the length designator is omitted, it is assumed to be 1.

```
CHARACTER*5 A, B, C
CHARACTER D*4, E, F*7
CHARACTER*6 X, Y*3, Z
```

In the preceding declaration statements, variables A, B, and C will contain 5 characters, D will be of length 4, E will contain 1 character, F will be 7 characters long, X and Z will both be 6 characters long, and Y will contain 3 characters. There is generally no specific limit to the length of a character string, though some systems may have a limit of 256 characters.

A CHARACTER type variable may be given a value in the usual ways: in an assignment statement, an input (READ) statement, or a DATA statement. Character strings can be tested for equality or inequality and manipulated in various other ways.

The order in which symbols are coded varies from one implementation to another; this order is referred to as the *collating sequence*. This sequence may be an ASCII or EBCDIC ordering, as shown in Appendix D. The relational operators, when applied to character expressions, return .TRUE. or .FALSE. depending on their order in this collating sequence. Fortunately, the order within certain subsequences is fixed, no matter what the particular collating sequence of your machine. The order of the capital letters, the small letters, and the digits is always the normal ascending sequence you would expect:

```
'A' < 'B' < 'C' < 'D' < ... 'Y' < 'Z'
'a' < 'b' < 'c' < 'd' < ... 'y' < 'z'
'0' < '1' < '2' < '3' < '4' < '5' < '6' < '7' < '8' < '9'
```

Thus, for most strings that you will be interested in comparing, such as English words or names, the relational operator .LT. will give you alphabetical ordering.

The relational operators .EQ. and .NE. are clearly independent of any collating sequence; either two strings are identical or they are not. You may even compare strings of different lengths; if you do, the shorter string is considered filled out with blanks to the same length as the longer string. For example, if we have two character variables A and B of unequal length, and they are assigned different strings, the following results will occur if they are tested for equality:

```
CHARACTER A*3, B*6
```

<u>A</u>	<u>B</u>	<u>A .EQ. B</u>
CAT	HORSES	.FALSE.
BAT	BATMAN	.FALSE.
RAT	RAT	.TRUE.

A list of character string variables can be sorted into ascending order, thus putting it into alphabetical order. Character strings can also be pulled apart into substrings, or concatenated together to form longer strings. The variety of character string manipulation techniques available in FORTRAN 77 gives the language a new dimension in its capabilities for nonnumeric problem solving.

* Substrings

When a character variable has been declared to be of a length longer than 1, it then may be thought of as having substrings. For example, the character variable A of length 4 has been assigned the value 'CATS'. It may also be thought of as containing the substrings 'C', 'CA', 'CAT', 'CATS', 'A', 'AT', 'ATS', 'T', 'TS', and 'S'. To access the substrings of a character variable, we may use a substring notation, which indicates in parentheses following the variable name the positions of the first and last characters of the substring, separated by a colon.

var	A	'CATS'
var(first:last)	A(2:3)	'AT'
var(:last)	A(:3)	'CAT'
var(first:)	A(3:)	'TS'
var(:)	A(:)	'CATS'

If both positions are specified, the substring is taken as that from position *first* through position *last*. If the first position is omitted, it is assumed to be 1, and if the last position is omitted, it is assumed to be equal to the length of the string.

Thus, we could use nested loops to print out all of the substrings of a particular string of length N in the following manner (use our 'CATS' example as a model in running through this code to see that it does in fact do what it claims):

```

CHARACTER B*...
READ*, B
DO 20 J = 1, N
  DO 10 K = J, N
    PRINT*, B(J:K)
10   CONTINUE
20   CONTINUE

```

Note that the program would print out the substrings of 'CATS' in precisely the order: 'C', 'CA', 'CAT', 'CATS', 'A', 'AT', 'ATS', 'T', 'TS', 'S'. For a string of length N, the number of substrings will be the sum of the integers from 1 to N, or $(N)(N+1)/2$ (in our case, there were 10).

*Concatenation

Two character strings, either character variables or string constants, may be *concatenated*, or appended, together, by the operator // . The resulting string will have a length equal to the sum of the lengths of the two original strings. Of course, if the result of the concatenation is to be stored in another character variable, the latter should be long enough to hold the entire string; if it is not long enough, only the number of leftmost characters equal to the length of the string variable will be stored. Thus, in the following program segment,

```
CHARACTER A*4, B*6
A = 'GOOD'
PRINT*, A, ' ', A//'BYE'
B = A//'BYE'
PRINT*, B
```

the printed results will be:

```
GOOD GOODBYE
GOODBY
```

List-Directed I/O of Character Data

We have already looked at some PRINT* statements for character strings, and noted that they print the string at the next position in the output, leaving no spaces between. Thus, if you want spacing, you must include character blanks between items as we did in the example at the end of the last section. Character constants are enclosed in single quotes (apostrophes) in the PRINT list, and what is between the quotes is exactly what gets printed (except if there is a single quote or an apostrophe *in* the string you want printed, the single quote character must be entered twice to print out correctly). Thus,

```
PRINT*, 'IT''S A LONG, LONG WAY TO TIPPERARY'
```

will print out

```
IT'S A LONG, LONG WAY TO TIPPERARY
```

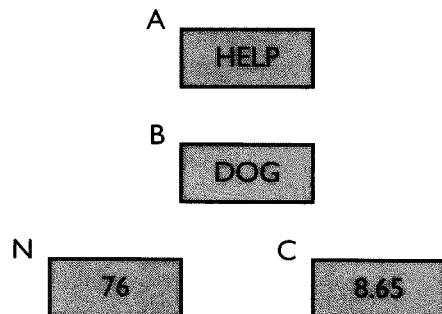
Note: Fortran 90 provides the option of using either single quotes or double quotes to enclose character constants or literal strings. Thus another way to handle our "Tipperary" problem in Fortran 90 would be to enclose the output string in double quotes and leave the apostrophe ("IT'S A LONG, LONG WAY TO TIPPERARY").

On list-directed input, a character string value must be enclosed in single quotes in the input record, and separated from other items on the list by one or more blanks and/or commas. If the string read into a character variable on a list-directed READ is shorter than the length of the variable, it is stored *left-adjusted* and blank-filled to the right; if the string is longer than the assigned variable length, only the *leftmost* characters of the string constant are stored, as many as will fit (this differs from the handling of such a case in a formatted READ, which we will examine in the next chapter). The following example would read values into character variables and other variables, as indicated.

```
INTEGER N
CHARACTER A*4, B*5
REAL C
READ*, A, N, B, C
```

reads the input record:

```
'HELPFUL' 76, 'DOG' 8.65
```



Useful Character Functions: LEN, INDEX, CHAR, and ICHAR

There are several built-in system functions which are helpful in performing character manipulations. The first is one which determines the *length* of a character string. The function is of the form:

LEN (C)

where C is a character string argument. The argument can be a character constant (such as 'KING KONG', which would have the length 9), character variable, or character expression. This is particularly useful when character strings have been passed as arguments to subprograms, as we shall see.

- The system function INDEX (C1, C2) takes two character string arguments, and returns the numeric position of substring C2 in string C1. If C2 does not occur in C1, it returns 0. Thus

```
CHARACTER A*5, B*2
A = 'HORSE'
B = 'OR'
```

```
N1 = INDEX (A, B)
N2 = INDEX (A, 'CAT')
```

stores a 2 in N1 and a 0 in N2.

There is a coordinated pair of character manipulation functions which can be used in a variety of interesting ways:

CHAR(n) and ICHAR(c)

The function CHAR(n) returns the character that is in integer position n of the system's *collating sequence* (the ordered set of symbols for your machine); the function ICHAR(c) returns the integer position of any character c in the collation sequence. Thus, you can see that these two functions have the relations:

$$\begin{aligned} \text{ICHAR (CHAR(n)) } & \text{ is } n \\ \text{CHAR (ICHAR(c)) } & \text{ is } c \end{aligned}$$

- ~ These functions are particularly useful in converting character representations of digits to their numeric equivalents, and in generating sequences of characters from the collating sequence. Let us say you have the problem of converting some character representation of a digit (some value '0' through '9') that is stored in a variable into its equivalent numeric value; how can this be done? For example, if the character variable D contains a '3', you want to convert it to a numeric 3 (and so on). If you use the position of the digit '0' in the collating sequence as a reference point, then the difference between the position of D (that is, ICHAR(D)) and that of '0' is equal to the numeric value of D. Thus, the numeric value N of the character digit D is determined as follows:

$$N = \text{ICHAR}(D) - \text{ICHAR}('0')$$

Let us examine a typical (ASCII) collating sequence for the digits and the capital letters, to see how our technique works:

Digit	Coll. Seq. Posn.	Letter	Coll. Seq. Posn.
'0'	48	'A'	65
'1'	49	'B'	66
'2'	50	'C'	67
'3'	51	'D'	68
'4'	52	
'5'	53	
'6'	54	'W'	87
'7'	55	'X'	88
'8'	56	'Y'	89
'9'	57	'Z'	90

You can easily see from this table how our conversion from character representation of a digit to its numeric equivalent works. If our digit D was '3', as we suggested, then ICHAR(D) is 51; ICHAR('0') is 48, and ICHAR(D) – ICHAR('0') is 3. This method works for any digit you select.

You can also use the fact that the letters of the alphabet appear sequentially in the collating sequence to easily generate a set of these characters, for output, storage, and so on. For example, if your problem requires the output of the letters of the alphabet 'A' through 'Z', one per line, the simplest way to do this would be to make use of the CHAR and ICHAR functions:

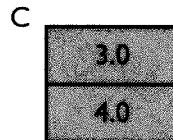
```
NPOS = ICHAR('A')
DO 20 I = 1, 26
    PRINT*, CHAR(NPOS)
    NPOS = NPOS + 1
20 CONTINUE
```



COMPLEX VARIABLES

A *complex* value is one that has two parts, a *real* part and an *imaginary* part. If a variable is designated complex in a type statement, it is assigned two consecutive locations in memory, and the first is used to store the real part of the complex value, the second the imaginary part. Both are stored as real constant values. An assignment statement to a complex variable has the following form:

```
COMPLEX C
C = (3.0, 4.0)
```



If a complex variable is output using a list-directed PRINT* statement, it is printed with the real and imaginary parts in parentheses, separated by a comma. This is also the form the input of a complex value to a list-directed READ* must take. Thus, the output of complex value with a PRINT* to some permanent device such as tape, disk, or the card punch could later be read by a list-directed READ*.

```
COMPLEX D
READ*, D
PRINT*, D + 5.0
```

will read the following record

(4.0, 6.0)

and print the output record

(9.0, 6.0)

Complex values can be combined with real or integer values in arithmetic operations, as indicated. A real is simply taken as a complex value with a 0.0 imaginary part, and an integer is floated and then combined with the complex value in the same way as a real. Complex values may not be combined with double precision values in FORTRAN 77 arithmetic operations.

There are system functions which will separate a complex value into its component parts. If C is a complex variable, REAL(C) will give the real part of the value, and AIMAG(C) will give the complex part. The function CMPLX will take two real arguments and combine them into a complex representation; for example, $C = \text{CMPLX}(A, B)$ will take two real arguments A and B and store the complex value $A + Bi$ into C. If two complex values, A ($a + bi$) and C ($c + di$) are combined using the normal arithmetic operators, the results are the following:

$$\begin{aligned} A + C &= (a + bi) + (c + di) = (a + c) + (b + d)i \\ A - C &= (a + bi) - (c + di) = (a - c) + (b - d)i \\ A * C &= (a + bi) * (c + di) = (ac - bd) + (ad + bc)i \\ A/C &= (a + bi)/(c + di) = (ac+bd)/(c^2+d^2)+(bc-ad)/(c^2+d^2)i \end{aligned}$$

You may recall that a negative quantity cannot be raised to a real power in FORTRAN, since this might require the taking of imaginary roots. However, you *can* express imaginary quantities directly if you use complex variables or constants.

You remember the formula for finding the roots of a quadratic equation of the form $ax^2 + bx + c = 0$:

$$\text{root} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Up until now, we have had to restrict ourselves to handling this only when the value under the radical (that is, $b^2 - 4ac$) is *positive*. However, with our addition of complex values to our toolbox, we can now solve the equation for any values of a, b, and c (recognizing that it may have complex roots). For example,

$$x^2 + x + 1 = 0$$

will have complex roots: $x_1 = (-1 + 3i)/2$, $x_2 = (-1 - 3i)/2$. To write a general-purpose program which will solve this for any coefficients in the equation, we must employ complex quantities. The system function CSQRT will take a square root of a complex quantity and return a complex result. Since our coefficients a, b, and c will be real to begin with, we will make the value given to the square root function complex before applying the function.

194 DATA TYPES IN GENERAL

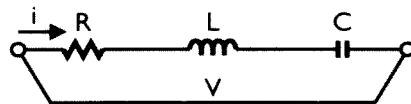
```
***** ROOTS OF QUADRATIC EQUATION *****
***** ALLOWING FOR THE POSSIBILITY OF COMPLEX ROOTS *****
COMPLEX RAD, RESULT, BC, TWOA, ROOT1, ROOT2
REAL A, B, C
PRINT*, 'ENTER THE REAL COEFFICIENTS FOR A QUAD. EQN.'
READ*, A, B, C
IF (A .EQ. 0) THEN
    ROOT1 = -C/B
    PRINT*, 'THE ROOT IS ', ROOT1
ELSE
    RAD = CMPLX( B**2 - 4*A*C, 0.0)
    RESULT = CSQRT(RAD)
    TWOA = CMPLX (2.0*A, 0.0)
    BC = CMPLX (B, 0.0)
    ROOT1 = (-BC + RESULT)/TWOA
    ROOT2 = (-BC - RESULT)/TWOA
    PRINT*, 'THE ROOTS ARE ', ROOT1, ROOT2
ENDIF
END
```

We have converted all quantities to complex before they are involved in complex operations. However, you can do arithmetic combining reals and complex values, and FORTRAN will do the conversions for you. For example, we could have written:

$$\text{ROOT1} = (-B + \text{RESULT})/(2.0*A)$$

Also be aware that our program would work equally well for an equation whose roots are *not* complex. The answers would simply be expressed as complex values with zero imaginary parts.

Complex arithmetic is also used in electrical circuit analysis (an electrical engineering application). For example, to calculate the impedance for a circuit with a resistance R, an inductance L, and a capacitance C in series, as shown:



the total impedance Z for the circuit is calculated from the formula:

$$Z = R + j(\omega L - 1/(\omega C))$$

where electrical engineering notation for the square root of -1, which we have been calling i, is j (since i is used in electrical engineering to represent current),

and ω represents the *radian frequency* of the current, or $2\pi f$ (where f is the frequency in cycles per second). Thus, if we are given values for R (for example, 8.0 ohms), L (for example, 2.38 millihenrys), and C (for example, 14.14 microfarads), we can compute the complex impedance Z for this circuit at a specified frequency f (for example, 500 cycles per second). We can also determine the magnitude of this impedance (absolute value of Z) and its "phase angle" (as the arctangent of the imaginary part divided by the real part):

```

COMPLEX Z
REAL R, C, L, F, ZABS, ANGLE, TWOPI, IMAGIN
PARAMETER (TWOPI = 2*3.14159)
READ*, R, C, L, F
IMAGIN = TWOPI*F*L - 1.0/(TWOPI*F*C)
Z = CMPLX ( R, IMAGIN )
ZABS = CABS(Z)
ANGLE = ARCTAN (IMAGIN, R)
PRINT*, 'IMPEDANCE IS ', Z, '
PRINT*, 'MAGNITUDE OF ', ZABS, ' OHMS'
PRINT*, 'PHASE ANGLE OF', TWOPI*ANGLE, ' DEGREES'
STOP
END

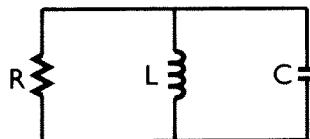
```

We would find for our example that the impedance is $(8.0, -15.0)$, with a magnitude of 17.0 ohms and a phase angle of -61.9° .

Many other computations can be performed using complex values in electrical circuit design. For example, voltage V is

$$V = IZ$$

where V is voltage and I is current (all may be complex). If we had three elements connected in parallel instead of in series:



the complex impedance Z of this circuit, at frequency f cycles per second ($\omega = 2\pi f$), would be:

$$1/Z = 1/R + j(\omega C - 1/(\omega L))$$

In general, if several impedances are combined in series, the overall impedance Z is:

$$Z = Z_1 + Z_2 + Z_3 + \dots \quad \{ \text{series} \}$$

and if they are combined in parallel, the overall impedance Z can be calculated from:

$$\frac{1}{Z} = \frac{1}{Z_1} + \frac{1}{Z_2} + \frac{1}{Z_3} \dots \quad \text{(parallel)}$$

Current and voltage can also be related by a quantity called the *admittance* Y, where $Y = 1/Z$; this also can be a complex quantity. An RLC circuit is said to be in *series resonance* when the imaginary part of Z (the impedance) is a minimum, that is:

$$\omega L - 1/(\omega C) = 0$$

and this occurs at $\omega_0 = 1/\sqrt{LC}$. Similarly, a parallel RLC circuit is said to be in parallel resonance when admittance is a minimum, and impedance is a maximum, which also occurs when:

$$\omega = 1/\sqrt{LC}.$$



DOUBLE-PRECISION VARIABLES

Double-precision variables are also assigned two consecutive locations in memory. This allows a larger exponent to be stored for a real value, or more significant places to be stored for the multiplier, or both. The problems of limited precision in real values are discussed in Chapter 8. Precisely how this additional storage space is utilized differs from one machine implementation to another. Thus, using double precision may allow us to store *larger* (or smaller—with a negative exponent of greater magnitude) values than we could with single-precision reals, or it allows us to carry out calculations to more digits of precision, or some combination of the two. The range of the size of values that can be stored using double precision will depend on your system, as will the number of significant digits it gives you; check it out.

A double-precision constant in an assignment statement or input may simply be represented as a decimal value with many places, or it may be represented in “scientific notation,” but using a D instead of the E to designate the exponent; for example, 0.5498643 D 287 would be a large double-precision value. Double-precision variables must be declared in a DOUBLE PRECISION type statement at the beginning of a program. They may be read into or printed out using list-directed I/O (or formatted I/O).

Note that simply storing a single-precision constant into a double-precision variable does not increase its number of significant digits. Thus, if we have declared the variable DOUBLE to be double precision, the assignment statement

```
DOUBLE = 0.1
```

will still only store the single-precision representation of 0.1 into DOUBLE. However, if you write

```
DOUBLE = 1.0 D -1
```

a double-precision representation will be stored.

If you wanted a greater accuracy in adding up the terms of a series such as:

$$1 + 1/2 + 1/4 + 1/8 + 1/16 + \dots$$

you could use DOUBLE PRECISION variables in the program. To make it more interesting, print out the sum after every 100 terms added, for a total of 1000 terms:

```
***** ZENO'S PROGRESSION *****
***** ADD UP THE TERMS 1 + 1/2 + 1/4 + 1/8 ... *****
***** FOR 1000 TERMS, PRINTING OUT EVERY 100TH PARTIAL SUM *****
      DOUBLE PRECISION TERM, SUM
      TERM = 1.0
      SUM = 0.0
      DO 100 I = 1, 10
          DO 50 J = 1, 100
              SUM = SUM + TERM
              TERM = TERM/2.0
50      CONTINUE
      PRINT*, 'THE SUM AFTER ',I*100,' TERMS IS ', SUM
100    CONTINUE
      STOP
      END
```

Since double-precision arithmetic usually takes much more computer time than comparable single-precision operations, you should be sure that you want to use double precision before you incorporate it into a particular problem. Double-precision arithmetic is more time-consuming because there are more bits which must be added, multiplied, and so on, or taken into consideration in the exponents. However, in some cases it may be necessary to use double precision to get anything close to reasonable results, if the values you are dealing with are very large or very small, or if they differ greatly in magnitude from one another. Please refer to the chapter on errors (Chapter 8) for more detail on these problems. The accuracy of your information may be sufficiently high that you want to retain as many significant digits as possible in the results. For example, if you have calculations accurate to 15 decimal places, and they further involve the use of pi, you will want to define that value to 15 places as well, and double precision will allow you to do this.

```
DOUBLE PRECISION PI
PI = 3.14159265358979 D 0
```

To make all variables and other symbolic names in your program integer (which might be desirable for certain programs), you can write either:

```
IMPLICIT INTEGER (A - H, O - Z)
```

or else

```
IMPLICIT INTEGER (A - Z)
```

since it does no damage to reiterate the default typing. The second form is simpler, and more straightforward.

Note: Fortran 90 provides a new IMPLICIT statement form, which can specify IMPLICIT NONE. The inclusion of this statement would then require that *all* symbolic names used in the program be explicitly typed, and those that were not typed would be flagged as an error by the compiler. This new feature, when it becomes available, will provide the FORTRAN programmer with the option of forcing declaration of all symbolic names, and the advantage of then having the compiler do a careful type-checking of all symbolic names in the program (as is a required feature in Pascal). The IMPLICIT NONE feature is currently available on several FORTRAN 77 compilers. Though its use is now nonstandard, it does have the advantage of providing a compiler type and definition check for all of your program variables. If IMPLICIT NONE is not available, the programmer can rely on a compiler-generated symbolic reference map to check the names of all variables used in the program, and a careful scan of such a list should bring to light any maverick variable names.



DATA STATEMENT INITIALIZATION OF VARIABLES

FORTRAN also provides a nice feature, not available in most other high-level languages, of initially defining the values of a number of variables in a compact manner, using a DATA statement. The DATA statement is of the form:

```
DATA listofvars1/listofconsts1/[,]listv2/listc2/...]
```

The list of variables may contain any kinds of variables, including array elements or array names (which will be discussed in Chapter 7), as many as you like. The number of constants in the following, corresponding list must agree exactly with the number of variables, and they are matched up, item-by-item, with the variables on the list. There may be multiple DATA statements in a program. Variables of type CHARACTER or LOGICAL must be given constants of the same type; numeric constants will be converted, if necessary, to the appropriate type of the variable on the list, as they would be

in assignment statements. [Note: This conversion was *not* performed in earlier versions of FORTRAN, and generally the best rule to follow is that the type of the constant in the data list should agree with the type of the variable into which it will be stored.]

The DATA statement puts the constant values into the variables on the list at *compile time*, so that these values are ready to be accessed at the time the program begins execution. The greatest advantage of the DATA statement is that it is much more compact than a number of corresponding assignment statements at the beginning of a program. Also, we will later see that it has special advantages when used in subprograms. A value set in a DATA statement may be changed later on in the program, and then it is never reset by the DATA statement.

There are further restrictions on the use of DATA statements with respect to items in COMMON, which will be discussed at the appropriate time. DATA statements can also be used to initialize values in an array efficiently, a method which will be discussed in Chapter 7. If a sequence of the same constant value occurs in a DATA statement constant list, it may be expressed using a repetition factor r , as $r*constant$. Thus the following are equivalent DATA statements:

```
DATA A, B, C, D / 3.0, 4.0, 4.0, 4.0 /
DATA A, B, C, D / 3.0, 3*4.0 /
```

Variable types may be mixed in the variable list of a DATA statement (though it would be better program style to keep the variables in a list all of the same type). Thus several more examples of DATA statements, all of which are legal, follow.

```
CHARACTER A*4, B*6
LOGICAL X
COMPLEX C
DATA A, N, Y / 'HOPE', 5, 2.6 / X, K / .TRUE., 7 /
DATA B, C, D / 'GOOD', (3.0, 4.0), 3.14159 /
```

The DATA statement provides a very compact, efficient means of initializing variable values for a program, and it is also readily changed to accommodate changes in program parameters.

Fortran 90 introduces an alternate technique for declaring variable types, and for the related PARAMETER statements. The declared type in the type statement may optionally be followed by a double colon (::) preceding the list of variables given that type; for example,

```
INTEGER :: A, BACK, WAY
COMPLEX:: CURRNT, IMPED
```

It is also possible to give a variable an initial value when it is declared; for example,

```
REAL A=3.789, X = 75.09
INTEGER :: HELP = 99, NOHELP = 0
```

Further, a KIND= parameter may be specified which refers to a system-dependent feature regarding the magnitude and precision of allowable values; consult your system manual on this.

In Fortran 90, a PARAMETER statement may combine typing and the defining of the symbolic constant value in one statement; e.g.,

```
REAL, PARAMETER :: ONE = 1.0, TWO = 2.0
```

The ability to initialize values in a type statement largely obviates the need for the DATA statement in Fortran 90, except in special cases where part of an array (see Chapter 7) is to be initialized, or in setting variables to binary, octal, or hexadecimal constant values (see Appendix A); for example,

```
INTEGER A, B, HEX
DATA A, B / B'1001', O"7752"/, HEX / Z'AC3'/
```



USER-DEFINED DATA TYPES

We have discussed six types of data available in FORTRAN: integer, real, double precision, complex, logical, and character. This covers a wide range, but occasionally you may think of other “data types” you wish to deal with, such as days of the week (Sunday, Monday, Tuesday, and so on), months of the year (January, February, and the rest), elements from the periodic chemical table (hydrogen, helium, lithium, and so on), or other ordered collections of data elements. These are not naturally built into any computer representation for constants, but some languages (such as Pascal) provide a capability for the programmer to establish lists of elements as *user-defined data types*, and even then to define operations on these data elements. There are ways to incorporate such user-defined data structures into your programs by using arrays (which we will discuss in Chapter 7), and even define operations on them using subprograms such as functions (discussed in Chapter 9). In these chapters, we will cover the tools needed to structure your own data types and define manipulations on them; in Chapter 13, in the section on implementing other data structures, we will see how to implement these data types, as well as sets, stacks, and records. In Chapter 13, we will also look at the facility available in Fortran 90 for creating derived data types.



FORTRAN 90 FEATURES

Character strings (appearing as constants or as literal strings for Input/Output) may be enclosed in either single quotes ('') or double quotes ("").

The IMPLICIT NONE statement may be used at the beginning of a program to prevent any implicit typing of variables (including the Fortran default typing), thus forcing the programmer to explicitly type all variables used in the program.

Type statements may have the type optionally followed by a double colon (::), and they may be used to initialize values of variables. A type may be declared as part of a PARAMETER statement, and the parentheses may be omitted.



SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

A *type statement* is used to declare the types of values that can be stored in different variables. A type statement is of the form:

type listofvariables

where *type* may be one of six possible types available in FORTRAN:

INTEGER—values of whole numbers, negative, zero, or positive, within the storage limitations of your machine

REAL—numeric values containing a decimal point, with an optional exponent (power of ten), within machine limitations

LOGICAL—values representing .TRUE. or .FALSE.

CHARACTER—character strings, where the *length* of the string must be specified as part of the type declaration; e.g.,

CHARACTER A*3, B*4 {or} CHARACTER*6 X, Y, Z

COMPLEX—two-part representations, containing two real values, representing the real and imaginary part, respectively

DOUBLE PRECISION—reals with more significant digits of precision, and possibly with larger magnitude (larger exponents) than normal reals, because they occupy *two* storage locations

An IMPLICIT type statement can declare all the variables beginning with a letter, or a range of letters, to be of a certain type; for example,

IMPLICIT INTEGER (A, C), LOGICAL(X-Z)

declares all variables in the program beginning with A or C to be of type integer, and all variables beginning with X, Y, or Z to be type logical. Exceptions can be made with new type statements.

A DATA statement can be used to initialize variable values.

Character operations to take substrings, concatenate strings, and the functions LEN, CHAR, and ICHAR were introduced.



EXERCISES

- Using LOGICALs, show that DeMorgan's Laws are correct. That is, print out "truth tables" such as those in the text for AND, OR, and the other connectives, using the FORTRAN expressions which are appropriate, for the following pairs of expressions which DeMorgan claims to be equivalent:

$$\begin{aligned} \text{.NOT. (X .AND. Y)} &= (\text{.NOT. X}) \text{.OR. (.NOT. Y)} \\ \text{.NOT. (X .OR. Y)} &= (\text{.NOT. X}) \text{.AND. (.NOT. Y)} \end{aligned}$$

Logical values for X and Y (which must be declared to be type LOGICAL) can be set to .TRUE. or .FALSE. using assignment, input, or DATA statements. Determine which approach you find most efficient. When you print out your truth table values for the expressions, for the time being you can use PRINT*. In the next chapter, you will learn how to use *formatted* input and output.

- Write FORTRAN code which will store a value .TRUE. in the logical location L if a variable X is greater than 5.0, and store a .FALSE. otherwise. Make your code as simple as possible.
- You can use logical values to represent membership or non-membership in *sets*. For example, the sequence of positive integers from 18 to 150 forms a *set*, perhaps the set of those people of legal voting age in a city. Then for any new person who enters the city, you can determine whether that person is of legal voting age (whether in the set, a value of .TRUE.) or not (.FALSE.). This could then be used to look at the *union* of two sets, A (our legal voters) and B, the set of tall people in the city (those 6 feet tall or taller). The union of A and B represents all of those people in the city who are either of legal voting age *or* who are tall (or both). Read in data on a set of ten people in the city (their names, ages, and heights in inches), and print out those in the *union* set we have described.

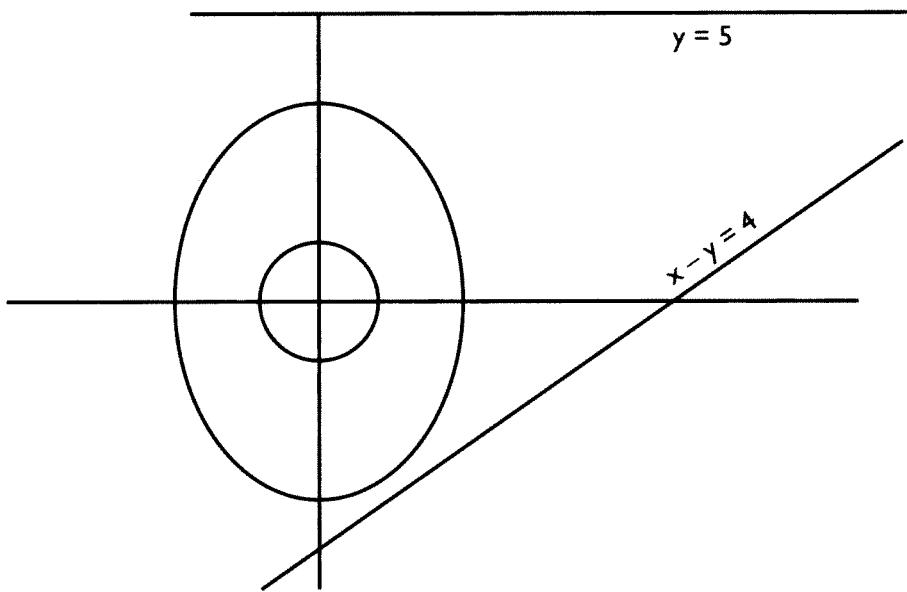
Now, with the same data, print out those who are in the *intersection* of sets A and B, that is, those who qualify as members of *both* sets, or are in A *and* B. These will be all of the tall legal voters in the city.

- You can use this notion of sets and logicals for many applications. For instance, imagine that you will be reading in a set of data points (x and y coordinates in a two-dimensional space), and these data points may cluster in various ways. Those points which fall on or in a circle of unit radius represent one set (set A—let us say these are points of close correlation with the target that we want to hit). Another set (B) represents those points falling within an

ellipse whose equation is:

$$\frac{x^2}{4} + \frac{y^2}{9} = 1$$

The points in the ellipse represent a “bombing pattern” that includes the target and a related area of interest. Another set (C) represents points too far north of the target ($y > 5$), and a final set (D) represents points too far southeast of the target (below the line $x-y = 4$).



Set up a program that will read in a pair of coordinates (x,y) for a data point, and determine its membership (or non-membership) in each of the four sets. Then determine if the point lies in:

B but not in A (that is, B.AND..NOT.A)
not too far north or southeast (.NOT.C. .AND. .NOT.D)

Notice that you can play many variations on this problem by varying the spaces the different sets describe. In each case, draw a picture of the spaces defined to guide you. You can also use the program to test many points, not just one.

- 5. Write a program to input an 8-letter word (you choose the word), and print out all of its substrings of lengths 1, 2, 3, and 4. The one-character substrings will, of course, be the eight letters in the word. The two-character strings will be its “digrams,” or letter pairs (there will be seven of them), the three-character strings its trigrams, and so on. This can form the basis of a study of letter frequencies, digram frequencies, and so on, in a text, which might then be used to determine its authorship. We will be able to tackle such a problem

when we have covered arrays in Chapter 7. Since chemical compounds are also represented by strings of characters, we might use similar techniques to analyze various compounds.

6. Use concatenation to look at all of the possible combinations of the three strings 'BIO', 'CHEM', and 'PHYSICS'.

7. A technique similar to that in exercise 5 can be used to play "anagrams." Take an input word, no longer than five characters, break it up into its individual letters, and then print out all of the rearrangements of those letters. There are $n!$ (n factorial) permutations, or possible arrangements, of n letters. Thus a five-letter word will have $5!$ (or 120) possible rearrangements. Setting up the logic for this problem will take some thought.

8. A complex value can be expressed as $x + yi$. It can also be expressed in *polar coordinates*, as r at an angle of θ , or as

$$r \cos \theta + r \sin \theta i$$

where $r = \sqrt{x^2 + y^2}$ and $\theta = \tan^{-1} y/x$

The complex value can also be expressed in *exponential form*, following Euler's Theorem, as

$$r e^{i\theta}$$

Using a list-directed READ, input a complex value (as two real numbers separated by a comma and enclosed in parentheses) and convert it to polar coordinates; print it out in its three equivalent forms.

Using your knowledge of these equivalent forms, write a short FORTRAN program that demonstrates the equivalence:

$$e^{i\pi} + 1 = 0$$

9. The roots of a quadratic equation,

$$ax^2 + bx + c = 0$$

can, as you know, be computed from the equations:

$$x_1 = (-b + \sqrt{b^2 - 4ac})/2a$$

$$x_2 = (-b - \sqrt{b^2 - 4ac})/2a$$

If the square root term in the equations has a negative argument, the roots will turn out to be *complex*. Using your facility with complex numbers, write a program to find the roots of this:

$$x^2 + 2x + 5 = 0$$

Print out the roots, then use your program to plug the complex values you have calculated back into the equation to verify that they actually are correct solutions.

- 10. To compare the results you can obtain with single and double precision, set up loops to approximate the value of pi from the following equation, derived by Leibniz:

$$\pi = 4 (1 - 1/3 + 1/5 - 1/7 + \dots)$$

by adding up 100, 200, 300, up to 1000 terms of the series. Use two different loops, one with single-precision variables and the other using double precision. Print out the results, both single and double precision, after each 100 terms added, and compare the results. Also compare the results to the actual value of pi to as many decimal places as appropriate. The following value of pi should be adequate for comparisons:

$$\pi = 3.14159265358979323846264338327950288419716939937510 \dots$$

Determine whether it may be worthwhile for you to extend the program to include more than 1000 terms in the sum.

11. A circuit has a resistance of 5 ohms and a capacitive reactance of -5 ohms in series. The current can be represented as (8.0, 0.0). Write a program to calculate and print out the impedance and the voltage in this circuit. Generalize your program so that it will do this job for any series RLC circuit.

12. Write a program which will calculate the resonance frequency of any RLC circuit (series or parallel) given the inductance L (in millihenrys) and the capacitance C (given in microfarads).

13. Write a program which will accept an argument N followed by N impedance values and a specification of either 'SERIES' or 'PARALLEL', and compute and print out the effective impedance of these N impedances connected as specified. Allow that the impedances may be complex, so each is to be input as a complex value [in a READ* statement, this means that they should be expressed (x,y)].

14. Write a program that will print out, separately, the real and imaginary components of the product of any two complex values that are entered.

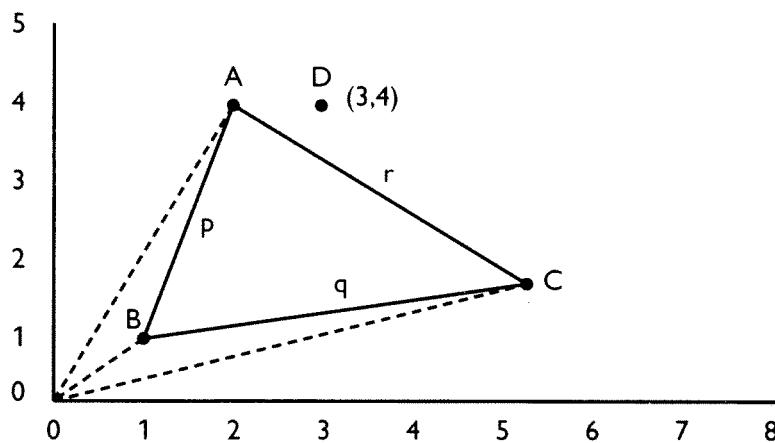
15. If you count complex roots, every complex number has three cube roots, four fourth roots, and so on. Thus the cube roots of 1 are $1, (-1/2 + \sqrt{3}/2 i)$, and $(-1/2 - \sqrt{3}/2 i)$; expressed as magnitude and phase angle, these are $1 \angle 0^\circ$, $1 \angle 120^\circ$, and $1 \angle -120^\circ$. The formula to calculate the nth root of any complex number, expressed as $r e^{Ai}$, or $r (\cos A + \sin A i)$, is

$$\sqrt[n]{r e^{Ai}} = \sqrt[n]{r} e^{A/n i}$$

Use this information (or its equivalent form, where $e^{Ai/n} = \cos A/n + \sin A/n i$) to calculate the nth root of any value (real or complex) given to your program.

Express your answers both in magnitude and angle form, and as complex numbers.

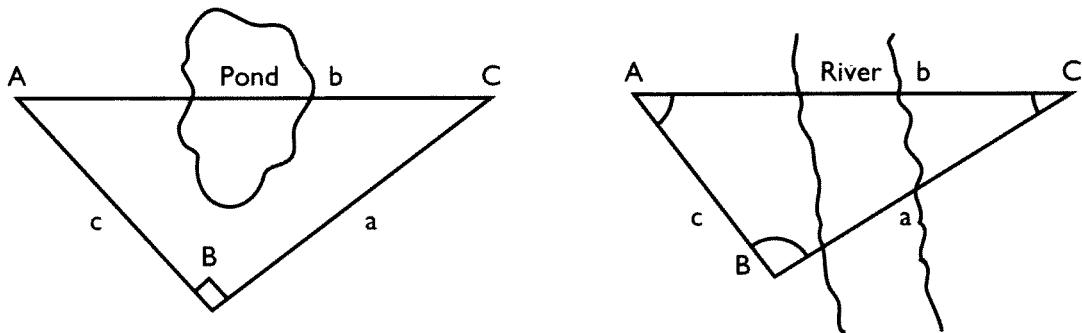
- 16.** Complex numbers can also be considered to represent points in the x,y-plane, where the real part of the number is the x-coordinate and the imaginary part is the y-coordinate. Thus the complex number (3.0, 4.0) is the following point D shown in the x,y-plane



Thus if you define three distinct points (A, B, and C) in the x,y-plane by three complex values, these can then be viewed as the vertices of a triangle. The three sides of the triangle can be represented as having lengths $|A-B|$, $|B-C|$, $|C-A|$, and so you can calculate the area of the triangle (if p, q, and r are the lengths of the sides of any triangle, then $s = (p + q + r)/2$, and the area of the triangle is $\text{Area} = \sqrt{s(s-p)(s-q)(s-r)}$). The point which is the center of gravity of the triangle is just $(A + B + C)/3$. Write a program to read in any three complex values as points in the plane, print out the lengths of the sides, the area, and the position of the center of gravity of the triangle.

- ◆ • 17. You are asked to read in data on a set of 800 measurements that will fall in the range from 101. to 200. grams. Set up ten different counters K0, K1, K2, ..., K9 which you initialize to zero, and use them to count how many of the measurements fall in the range greater than 101. but less than or equal to 110. (use K0 to count this), greater than 110. but less than or equal to 120. (use K1), and so on. You then want to output your results as a count n followed by n asterisks ('*'), to give a histogram of the frequency distribution of the values. *Hint:* begin with a character string A of length 120 (CHARACTER A*120) that is set to all blanks initially (A = '' will do it, since the rest of the string will be blank-filled). Then fill all of its one-character substrings from A(1:1) through A(n:n), using a loop, with '*'s; if you then use PRINT*, A, it should print out n asterisks followed by blanks. [Note: You will find the part of the program using the counters much easier after we introduce arrays in Chapter 7.]

- 18.** In civil engineering, the problem often arises of measuring inaccessible distances (across ponds or rivers, or the like, as illustrated in the diagrams).



In the pond case on the left, the engineer can make a right angle (or some other angle, which will also work) at B between the two measuring lines AB and BC, and then determine the distance AC using the Pythagorean theorem. Write a FORTRAN program to accept the distances AB and BC, assuming a right angle, and determine and print out distance AC across the pond. Modify the program to accept AB, BC, and the angle measured at B, and calculate AC. It will be $(b^2 = a^2 + c^2 - 2ac \cos B)$.

In the river problem on the right, the distance AB can be measured, plus the angles (line of sight) at A and B when sighting to point C across the river. (The angle at C can also be measured as a check, but C should equal $180^\circ - A - B$.) The sides of the triangle, AC or BC, can then be determined using geometry. Write a program to accept the distance AB and the two angles at A and B, and calculate and print out AC (or BC). You might want to use complex variables for this problem.

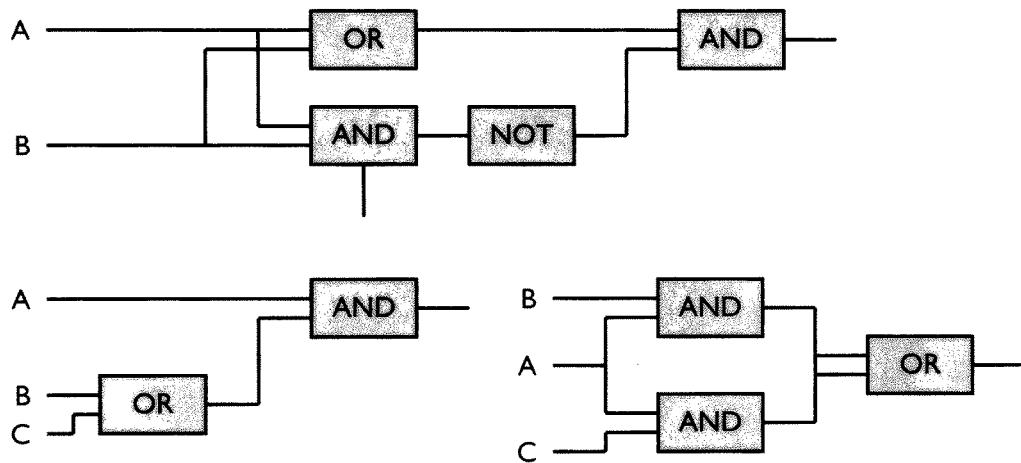
- 19.** You might need to calculate the absolute value of a complex number, $a + bi$. This can be done using either the system function CABS (see Appendix B), or by the following formulas:

$$\begin{aligned} \text{absolute value} &= (a^2 + b^2)^{1/2} \\ \text{or} &= x(1 + (y/x)^2)^{1/2} \\ \text{or} &= 2x(1/4 + (y/(2x))^2)^{1/2} \end{aligned}$$

where x is the maximum of the absolute values of a and b, and y is the minimum of the two absolute values. Write a program which will accept a complex number and calculate its absolute value using the system function CABS and using the three different formulas suggested here. Print out the results for comparison.

- 20.** A computer can be used to design other computers (such as the circuit board shown at the beginning of this chapter). As a modest beginning to such

a complicated project, write a FORTRAN program which will define a logical value (.TRUE. or .FALSE., 1 or 0) for values A, B, and C, and show the output results from the logical circuits shown. (If you have access to a system, or Fortran 90, that allows defining Boolean strings using DATA statements, such as DATA A, B /B'1001', B'1010'/ , use those.)



CHAPTER 6



FORMATTED INPUT/OUTPUT

In order to create really effective output, you need to learn the formatting controls in FORTRAN.

Using these, you will be able to output well-organized tables of results, graphic data, and other interesting symbolic information. Formatted control of input will allow you to handle already existing data in any form, not just that tailored to suit the list-directed READ. These techniques will greatly extend your creative capabilities on the computer.

This IBM POWERstation 530 system allows the user to create sophisticated three-dimensional output such as engineering designs, molecular models, and full-color animation.

"Form ever follows function."

- Louis Henri Sullivan, The Tall Office Building Artistically Considered,
Lippincott's Magazine (March 1896)

◆ WHY BOTHER?

So far we have been using list-directed I/O, and it has served us well—or has it? We can get values in and out, but on input we must follow certain definite restrictive rules, such as separating values by blanks or commas and enclosing character strings in single quotes, and on output the control of how it looks is largely out of our hands and up to the computer. If we have large real values that represent money amounts, the computer will print them out in scientific notation, with an E and an exponent, instead of looking like dollars and cents. It would be desirable, at least upon occasion, for the programmer to have more control over the form of the input and output of a program.

Even though it is slightly more complex than formatted input, we will begin with a discussion of formatted output, since *all* of our programs will need some output. The transition to understanding input formats will then be relatively easy.

PRINT OR WRITE (OUTPUT)

We will begin by covering the elements that can appear in a FORMAT statement to describe an output record to the printer or other output device.

Such a statement is utilized in the following manner:

PRINT n, list {or WRITE (unit, n) list }
n FORMAT(format edit descriptors)

where n represents a statement number identifying the FORMAT statement to be used to output the indicated list. What follows the word FORMAT in statement n is a complete description of what the output line will look like. This may include a “carriage control” character (for vertical spacing of output to printer or terminal screen), and may include spacing, tabbing controls, literal strings, controls to skip lines, and format descriptors for any items on the print list. We will be discussing all of these Format description elements.

Carriage Control, Spacing (X)

When program output is to go to the printer or the screen (the two most common cases), the first consideration in the output of a line to a visual

medium is where it should be positioned relative to the previous line. Thus every output line to a printer or screen should begin with a carriage control character, which indicates where the line will be positioned vertically. Since this is the first output consideration, and since its controlling indicator is the first thing to appear in our FORMAT statement, it is only appropriate that it should be the first item we discuss in output formatting. We should note here, however, that output to external devices other than printer or screen (such as tape or disk, for example) does not require this vertical positioning information, and the screen does not respond to a *top-of-a-new-page* carriage control.

There are ways to skip additional lines, using the slash (/) indicator, which we will discuss in a later section, but aside from that, there are basically four choices for carriage control in the FORTRAN Standard. The carriage referred to is that of the printer (analogous to the carriage on a typewriter), and it can be controlled to position the next line displayed. Since the printer outputs to special connected reams of paper that are perforated at the page breaks, to allow for easy separation, it can also identify the beginning of a new page (by the perforation). The printer can be instructed to print on the very next line, skip a line, print at the top of a new printer page, or print on the *same* line (potentially dangerous, only to be used with care). These vertical spacing control characters are:

→ Carriage Control Character	Result (Vertical Spacing)
' ' (blank)	print on next line
'0' (zero)	skip a line
'1' (one)	top of a new page
'+' (plus)	print on same line

Note: The '1' carriage control does not have the "top of a new page" effect on the terminal screen, since the screen does not have pages. It generally acts like a blank carriage control when used for output to the screen.

The '+' carriage control can be used (carefully) to achieve interesting overprint output effects on the printer. On the screen, however, it generally causes the output from the existing line to be wiped out as the new line is displayed, thus destroying the effect in that medium (though it can be used to create other interesting screen effects). [This screen reaction to the '+' carriage control may vary from one system to another.] Note that we are using carriage control *characters*; this is because the entire output record will be in terms of characters. Formatted I/O on the computer involves transmitting strings of characters, converted to or from the appropriate data types. What are transmitted are "text files," those most readily comprehended by human users. There is another option, to be discussed later, where output from a program is in binary, for fast readability by another computer; this is called *unformatted I/O*.

If the programmer does not specifically choose to include a carriage control character at the beginning of the Format record, the first character in the

output record defined by the FORMAT, whatever it is, will be taken as carriage control. The first, vertical spacing control, character does not appear as part of the output record in printing; it just determines line positioning. Most other characters that could accidentally get grabbed off for carriage control if you do not deliberately specify one will have the same effect as a blank—that is, to print on the very next line. Often a few characters will have unusual effects on a system, such as skipping halfway down a page; you have to check out any such special cases in your system manuals if you want to use them (but they won't be portable).

The simplest case of a formatted output statement would be one without a list of items to be printed; it would only space (skip one or more lines), or print some message included in the Format. To put blank spacing into an output

- Format, use the edit descriptor X. One X indicates one blank to appear in the record, XX (or 2X) indicates two blanks, and so on. You can always use a repetition factor n before the edit descriptor to be repeated, such as 5X. A literal string (of characters), enclosed in single quotes, will appear in the output record at the position where it is described. The output record is described, after the carriage control character, beginning in column 1, as far as is desired, and the rest of the line will be blanks. Thus, the following PRINT/FORMAT combination:

```
PRINT 66
66 FORMAT('0',10X,'THIS IS MY FIRST PROGRAM')
```

will skip a line, space in 10 spaces, and print out:

```
THIS IS MY FIRST PROGRAM
```

Your system may use only capital letters in FORTRAN, in which case the only characters which can appear in a literal string for output will be capital letters, digits, and the special symbols you have available. However, many systems also include lowercase letters of the alphabet and, if the printer will accommodate them, you can use them in literal strings as well. Some systems allow you to enter FORTRAN instructions in either uppercase or lowercase, or a mixture; this is something you will have to check out. In this book, we have used the standard convention of writing all FORTRAN instructions in uppercase letters.

Notice that the FORMAT is merely a way of introducing a character description of how the line is to be output. Another option is to put this literal description in the place of the FORMAT statement number; you may even use a character variable to store the string, and refer to the character variable in the PRINT statement. Thus, our earlier example could alternately be expressed in either of the two following ways:

```
PRINT ('''0'',10X,''THIS IS MY FIRST PROGRAM'')
```

or else

```
CHARACTER OUT*40
OUT = '(''0'',10X,''THIS IS MY FIRST PROGRAM'')
PRINT OUT
```

Note that the single quotes of the message, when placed within the single quotes that set off the character string, had to be doubled (as in "0") in order to be understood by the compiler.

These have been very simple examples, just to introduce carriage control, spacing, and literal strings in output. Most output statements you use, however, will have a print list of variables and expressions whose values you want output as part of the record. Each of these items on the print list will require its own edit *format descriptor* to indicate how you want it output—how much space it is to take up in the record, how many places you want to see displayed to the right of the decimal point in a real, and so on. We will now cover the forms available for the various data types we will want to output.

Integers (I Format)

Integer values and variables are used very frequently, and they have a relatively simple format descriptor, so we will begin our discussion of edit format descriptors with that for integer I/O. If an integer variable or expression appears in your I/O list, it must have a corresponding edit format descriptor in the FORMAT statement that indicates how it is to be handled. On output, this descriptor indicates how many columns wide (w) the output field containing this integer value will be. This is done by using an Iw edit descriptor, where the 'I' indicates that the value is Integer, and the 'w' is an integer value indicating its field width, that is, how wide a field it will be written in. Thus a simple example would be:

```
-      NICK = 5**2
      PRINT 88, NICK
88  FORMAT(' ', 5X, I4)
```

In this example, the value (25) of NICK will be printed on the next line (blank carriage control), after 5 spaces, in a field which is 4 columns wide. When, as in this case, the field is wider than the value to be output, the value is output *right-adjusted* in the field. Thus in our example, there will be 7 blanks on the line followed by the value 25. Note that if the integer value to be output is negative, the '-' sign will also occupy a column, and must be accounted for. We will output:

In FORMAT 88 of the previous example, we could have incorporated the blank carriage control into the initial spaces in the format, and have written instead:

```
88 FORMAT(6X, I4)
```

and it would have had the same effect. If the field width specified for an integer value is not wide enough to accommodate the value, all stars (*) will be output in the field. Thus, if we had the following output statement in our program:

```
MACK = 10**8
PRINT 88, MACK
88 FORMAT(' ', 5X, I4)
```

the output would be:

Instead of printing *part* of the value, which the user might then interpret as a correct answer, the stars in the field serve as a flag that the value is too big for the field provided. The program could then be rerun with a larger output field width.

- There is one other variation on the I format. You may write an edit descriptor of the form Iw.m, which indicates a field width w and guarantees that a *minimum* of at least m digits will be output in the field. The value of m must not be larger than w. If we printed the earlier value of NICK (25) with an I4.4 descriptor, it would print leading zeros in the field (0025). If m is zero (e.g., I4.0) and the value the descriptor is used to output is zero, all blanks will be output in the field.

A repetition factor can be used if two or more edit descriptors in a FORMAT are the same. For example, in a loop that is printing the values of N, N**2, and N**3, to indicate side length, area of a square of that side length, and volume of a cube of the same length side, for a range of values, the table might be output in uniform fields, each eight columns wide:

```
PRINT 35
35 FORMAT ('1', 7X, 'SIDE', 3X, 'SQUARE', 3X, 'CUBE')
      PRINT 45
45 FORMAT(' ', 6X, 'LENGTH', 3X, 'AREA', 3X, 'VOLUME')
      DO 50 N = 1, 10
50      PRINT 55, N, N**2, N**3
55      FORMAT('0', 2X, 3I8)
```

The output from this program would look as follows:

SIDE LENGTH	SQUARE AREA	CUBE VOLUME
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

Reals

A real, or floating-point, value must be output with an appropriate edit descriptor, which may be F, E, or G format. The most commonly used is the F (for Floating-point) descriptor, which is of the form Fw.d, where w indicates the width of the field the value will occupy, and d indicates the number of decimal places you wish to see displayed to the right of the decimal point. Whereas a list-directed output statement will probably give you as many digits in output as are significantly stored on the machine, and output large or small values in E notation, you can force the output of a definite number of places, and keep the value expressed simply in whole number and fractional part notation, without the use of exponents, if you use F format. You must be sure, however, to make your field wide enough to hold all of the digits in the number, in the form you have requested, or else stars will be output in the field.

For example, you may be dealing with a problem of how much money is accumulated at a certain rate of interest over a given number of years when the interest is being compounded. For a principal P, a rate (decimal) of interest R, and a number of years N (if compounded annually), the formula is:

$$P (1 + R)^N$$

If the rate is reasonably high (say, 7% or more), and the time period is long, these values can get very big. For example, a classic beginning computer problem which you may have encountered is that of determining how much money the Indians who sold Manhattan Island to the Dutch in 1626 would have in the bank today if they had deposited their \$24 at rates of interest differing from 1% to 10%. If you run this program using list-directed I/O, you will always get the same number of significant digits printed out, and by the time you get to 4%, the output may have switched to E notation, which does

not look much like dollars and cents. You can correct this by using an F format for the output, being careful to use a very wide field.

```
***** MANHATTAN MAD MONEY *****
***** $24 FROM 1626 TO 1992 AT 1% TO 10% INTEREST *****
      N = 1992 - 1626
      PRINT 8
8   FORMAT('1', 5X, 'INTEREST RATE', 5X, 'ACCUMULATED TOTAL')
      DO 20 I = 1, 10
          R = I/100.0
          TOTAL = 24.0*(1 + R)**N
          PRINT 12, I, TOTAL
12   FORMAT('0',10X,I2,'%',8X,'$',F20.2)
20   CONTINUE
      STOP
      END
```

This program will create output, beginning at the top of a new page, something like (the layout will be the same on all machines, but the number of significant digits may differ):

INTEREST RATE	ACCUMULATED TOTAL
1%	\$ 915.87
2%	\$ 33718.27
3%	\$ 1198490.25
4%	\$ 41153752.00
5%	\$ 1366109312.00
6%	\$ 43868135424.00
7%	\$ 1363583172608.00
8%	\$ 41049779077120.00
9%	\$ 119713244219392.00
10%	\$ 33879580351660032.00

This program was run on a machine which has only about 7 or 8 significant decimal digits for single-precision values (so digits displayed beyond that point are meaningless). A machine with a larger word size, or the use of double precision, would give values significant to more places. But the output looks like dollars-and-cents amounts, and gives a sense of the magnitude of the large amounts at higher interest rates. You can thus begin to see the advantages of having control over your output format.

Note that this problem has natural extensions to problems of population growth, bacteria growth, spread of infectious disease, and so on, all of which follow similar growth patterns.

No matter how many significant decimal digits can be represented on the machine you are using, your Fw.d edit format descriptor indicates exactly how

many places you want to see displayed to the right of the decimal point, and this may have to fill in zeroes (as in our Indians' Interest example) or cut off trailing digits to do so. If trailing digits must be cut off in output of a real value, the last digit output is rounded up or down as appropriate. Thus, in the following program segment:

```
A = 1.0/3.0
B = 2.0*A
C = 5.0 + 2.5
PRINT 7, A, `B, C
7 FORMAT(6X, 3F6.3)
```

the output will be:

.333 .667 7.500

Notice that w must be larger than d, large enough to accommodate d decimal places, a decimal point, any significant whole number part digits, and possibly a sign. Generally, w must be *at least* d+2. Again, if the field given is not wide enough, stars will be printed in the field. Thus, if we had tried to output the value of C (7.5) in the previous example with an F4.3 field (which would have worked for A and B), it would have output **** (that is, it fills the field—here 4 columns—with *s).

Single-precision real values can also be output in scientific or exponential notation, using an Ew.d format. This asks for the value to be output with d decimal places to the right of the decimal point, in a field w columns wide, and that it be expressed as a multiplier times ten to some power. The output result is interpreted in the same way as the real constants we have seen expressed in scientific notation. In this notation, the field width must be made large enough to accommodate the number of places to the right of the decimal point (d), the decimal point, the 'E', the exponent (a signed 2-, or possibly, on some systems, 3-digit number), and a possible sign of the number. Thus a good rule of thumb is that w should be at least d + 6. On most systems, the multiplier will be output as .xyz..., or as 0.xyz... if space permits (in which case the field width should be at least d + 7), but some systems will output the multiplier as as x.yz...

Exponential (E) format will always output a value (not asterisks), as long as you have selected a field width (w) wide enough for your d and the range of exponents available on your system. However, values in this notation are not very accessible to the normal human reader. A value output using F format is more [human-]readable, but it is easier to run the risk in this format of not choosing a field wide enough to contain the value. If you do not know much about the magnitude of values you might expect to have to output, this can be chancy, and you may often end up with stars in your output fields. E format is safer, but less satisfying. There is an available compromise in FORTRAN, the G format.

A format descriptor of the form Gw.d will output a real value in F format if it can be done in the field (that is, if the value is between 0.1 and 10^{**d}), and in E format otherwise. This allows the possibility of using the more readable (F) format if it is possible in the space provided, but the facility to switch to E format if it cannot be done in F. In this way, you are saved from "seeing stars" in your output field, and the value is output in the best way possible.

· Characters (A Format)

-Character strings (constants or variables) may be input or output using an Aw format, or simply with an A format. As in the case of integers, Aw format indicates a field width of w columns for the datum. If only an A edit descriptor is specified, the length of the field is determined by the length of the character entity involved. On output, if the field width w specified is less than the length of the value to be output, only the *leftmost* characters of the string will be output. If the field width w is greater than the length of the character entity to be output, it will appear in the rightmost portion of the field, with blanks filled in to the left in the field.

Thus, for example, in the following program segment involving character variables, the output will be as indicated:

```
CHARACTER C*5, D*8
C = 'HELPS'
D = 'GOODNEWS'
PRINT 6, C, D
6 FORMAT(5X, 2A4)
PRINT 7, D, C
7 FORMAT(5X, 2A9)
```

The output from this will be:

```
HELPGOOD
GOODNEWS      HELPS
```

Logical, Complex, and Double-Precision Values

Logical values are .TRUE. or .FALSE., or .T. or .F., and they may be output with an Lw format. As usual, w is the field width in which the value will be output. The value to be output must be of type LOGICAL. The output value will be a T or F, right-adjusted in the w-column field.

Complex values have two components, a real and an imaginary part. These two values can be output using any of the available floating formats, as a pair.

Thus a complex value might be output with a 2F8.2, or a 2E10.2, or a 2G10.2 format. Though list-directed output encloses the components of a complex value in parentheses, you would have to add this feature yourself in formatted output—for example, FORMAT (3X,'(,F5.2,',',F5.2,)').

Double-precision values should be output with a Dw.d format, where w and d have the established meanings. They will be output in an exponential notation, with the difference being that the exponent follows a 'D' rather than an 'E'. There is, as we have discussed, the possibility of larger exponents and/or more significant digits in a double-precision value than in a normal real, so your formats for such values should reflect this.

Table of Various Outputs

The following table indicates the various output results created by the use of different formats we have discussed.

Internal Value	Format	Output	Format	Output
357	I3	357	I8	357
1000000	I3	***	I8	1000000
1.0/3.0	F12.10	0.3333333433	E12.6	0.333333E+00
1.0/3.0	G12.6	0.333333	F5.5	*****
12345.67	F12.10	*****	E12.6	0.123457E+05
12345.67	G12.6	1234.57	F12.6	*****
3.1415926	F12.10	3.1215925026	E12.6	0.314159E+01
6.03 E-23	F12.10	0.0000000000	E12.6	0.603000E-22
'CAT'	A8	CAT	A	CAT
'ABCDEFGH'	A8	ABCDEFGH	A4	ABCD
.TRUE.	L1	T	L3	T
(4.57, 2.16) {complex}	2F4.1	4.6 2.2	F5.2,',',F5.2	4.57, 2.16
1.0D0/3.0D0 {double precision}	F12.10	0.3333333333	D12.6	0.333333D+00

Slash (/) in Format—Skipped Records

We have seen the standard carriage controls for output, which print on the next line, skip a line, print on the top of a new page, or print on the same line. Sometimes these four options do not handle the particular situation you might be interested in representing. A slash (/) in a FORMAT statement indicates a *skipped record*. On output, this means a skipped line. Thus, five slashes, represented as ///// or 5(/), indicate moving down five lines in output (that is, the output device—printer or terminal screen—is positioned 5 lines lower than it was before). If you utilize such indicators to skip lines *before* a line to be output, the line itself still needs its own carriage control. A set of n slashes at the beginning or the end of a FORMAT indicate n skipped records; n slashes in the middle of a FORMAT indicate n-1 skipped records, since the first slash is merely taken to indicate the termination of the previous record being defined. Thus a FORMAT

```
PRINT 6
6 FORMAT(///'0',30X,'HELLO, SAILOR'//31X, 'HOW'S TRICKS?'//)
```

will move down three lines (for the three slashes) from the last line output, skip another line (for the '0' carriage control), move over 30 spaces, print HELLO, SAILOR, end the record, skip one line, take a blank for carriage control, and so print on the next line, space over 30 spaces, and print out HOW'S TRICKS?, and skip two more lines.

```
HELLO, SAILOR
HOW'S TRICKS?
```

Note: This example was chosen because it was an instance of a computer "bomb" once planted by a disgruntled employee; he had programmed the system so that, a few months after he had been fired, the system froze up and would only print this message.

Example Program Simulating Gasoline Pump

We can use the '+' carriage control to create interesting effects on the terminal screen, because it causes the previous contents of a line to be "wiped out" by the new line. Thus, we can use this, for example, to simulate the face of a gasoline pump as gas is drawn and total gallons taken and corresponding total cost is registered. We set up a fixed price per gallon (PPG), as would be set for the pump, and then have the number of gallons increase in "snapshot" form, say, every 0.2 gallons, up to a total of 12 gallons. There are two "windows" for our display: one for GALLONS and the other for TOTAL (cost). We have

included a long "do-nothing" loop between steps on the gas pump, because otherwise the output would zip by too fast for us to see. Notice that we have used the *slash* control at the end of the header line, to skip two lines *after* the header, before the gas amount and total cost will be written and overwritten.

```
***** GASOLINE PUMP DISPLAY *****
PRINT 7
7 FORMAT('1', 2X, 'GALLONS', 2X, ' TOTAL'//)
    PPG = 1.09          {OR USE READ*, PPG }
    DO 40 G = 0.0, 12.0, 0.2
        PRICE = G*PPG
        PRINT 8, G, PRICE
8 FORMAT('+', 2X, F5.1, 4X, '$', F5.2)
    DO 30 I = 1, 1000000
30      K = I*2**2 + 9*2**3*2
40 CONTINUE
END
```

Note: If this program were run to the **printer**, all 72 lines of gas amount and cost would overprint in the same place!

Repeated Format Patterns

We have seen that each new PRINT statement creates a new record (line). However, what if the list to be printed is longer than the format will accommodate?

We have already seen that you can use a repetition factor to precede a simple format descriptor, such as 5I4 or 4F6.2. It is also possible to indicate repetition of more complex format patterns by using a repetition factor and enclosing the pattern in parentheses. Thus, the following FORMATS are equivalent:

— 65 FORMAT(['0', 2X, I5, 2X, I5, 2X, I5, 2X, I5])

and

65 FORMAT('0', 4(2X, I5))

75 FORMAT('1', I4,3X,F5.1,2X, I4,3X,F5.1,2X, I4,3X,F5.1)

and

75 FORMAT('1', 3(I4, 3X, F5.1, 2X))

* There is only one difficulty with repeated format descriptions that include interior parentheses. If you have an output list that is longer than your FORMAT will accommodate, the FORMAT is repeated as many times as necessary to output the list. Each time the format is reused, it is as if a brand new PRINT command had been entered, and a new record is created. Thus, if you have a print list with 10 items on it, for example:

```
PRINT 85, I, J, K, L, M, N, INK, JACK, KAT, MACK
85 FORMAT('0', 4I6)
```

and the FORMAT will only print 4 items per line, the PRINT statement will reuse the FORMAT, three times in this case, to skip a line, print 4 items (I, J, K, and L) on the next line (each with an I6 format), skip a line, 4 items on the next line (M, N, INK, and JACK), skip a line, and two items on the last line (KAT and MACK). This is a useful feature which eliminates the need for writing many FORMAT statements of the same structure. However, if the FORMAT which is reused contains interior parentheses, matters may become complicated. When the format is reused, it is only repeated from the left parenthesis that matches the rightmost inner right parenthesis. Thus in our example, if the format used to print the list of ten items were:

```
85 FORMAT('0',4(2X,I4) )
```

the first time the format is used a line will be skipped and the first four items printed, but after that the only part of the FORMAT that is repeated will be the 4(2X,I4), and the '0' carriage control will have been lost, the first blank taken as carriage control, and the values on the next lines will not line up with those on the first line.

In this example, if the values on the print list, of I, J, K, ..., MACK had been 100, 200, 300, ..., 1000, let us compare the effects of the two FORMAT statements by placing the two outputs side by side:

FORMAT				<u>('0', 4I6)</u>				<u>('0', 4(2X, I4))</u>			
100	200	300	400					100	200	300	400
								500	600	700	800
500	600	700	800					900	1000		
900	1000										

Just as with integer division truncation, once you are aware of this rule in FORTRAN, you can use it to your advantage. For example, if you had the same print list of 10 items, but you wanted to start it off on the top of a new page, you would not just replace the '0' carriage control with a '1', because that would end up printing one line on the tops of three successive pages. You

could, however, make use of repeated format with interior parentheses by making your format:

```
85 FORMAT('1'/(0', 4I6) )
```

In this way, the '1' carriage control would only be used once, and the ('0', 4I6) format would be reused three times to output the list of ten integer items at the top of a new page.

Recap

We have now covered the output format descriptors for all the available data types in FORTRAN, carriage controls, and spacing horizontally (X) and vertically (/). Combining these features should give you great control of the form of your output. A few more advanced features of I/O, such as Tabs, will be discussed in a later chapter. The important thing to remember is that if you have a list of values to be output, each of them needs its own edit descriptor in the FORMAT statement, and they are aligned with the descriptors in the same order they appear on the output list. Thus in the following short program segment, the indicated correspondences apply between the print list and the edit descriptors in the format statement:

```
INTEGER INK, LATE
REAL B, C, D
CHARACTER A*5
...
PRINT 77, B, INK, LATE, A, C, D
77 FORMAT('0',F5.2, 2I5, 2X, A6, 2X, 2F7.2)
```

We have been using the simpler PRINT statement, assuming that for the time being, most of your output will go to the printer or to your terminal screen. However, the more general form of the output statement is:

```
WRITE (u, n) list
```

where u represents a *unit number*, and n is the statement number of the FORMAT to be used in output. The unit number identifies the device to which you will be writing on output. You may use the unit number of the default output device (the printer on batch systems, your remote terminal on interactive systems), and your output statement will then be equivalent to a PRINT. The default output device unit number on many systems is 6, but yours may be different; check it out. If the default output device is unit 6, then the following output statements are equivalent:

```
PRINT 5, list      WRITE (6,5) list      WRITE (*,5) list
```

Putting an asterisk (*) in the unit number position of the WRITE will automatically give you the default output device.

If you specify some unit number other than that of the default output device, you must include an OPEN statement in your program that *attaches* that device to your program, and then you may write to tape or disk, for example, for a more permanent record of your program output. The OPEN statement and handling of external files will be discussed in detail in Chapter 11.



INPUT FORMATS

On input, you have a list of variables to be filled with values from some external source. This source may be your terminal keyboard, the card reader, or tape or disk. As in output, you may have simple READ statements without a specified unit number, which read from the default input device (the terminal keyboard on today's interactive systems), or you can have a unit number u mentioned in the READ control list:

`READ (u, n) list`

On many systems, the default input device unit number is 5 (check your system conventions). If the default input unit is 5, then

`READ 7, list` `READ (5, 7) list` `READ (*, 7) list`

are all equivalent input statements.

END= and ERR= Clauses

An input (READ) statement can be modified to include two useful statements to handle error conditions, or end-of-data signifiers, that may arise. They are the ERR= and the END= clauses, and they may be included in the READ as follows:

☛ `READ (u, f, END = n, ERR = m) list`

In this statement, the u stands for the unit number to read from, f is the format statement number, END = n identifies statement number n as the one to branch to if an end-of-file identifier is encountered (sort of like using a "flag" value), and ERR = m establishes statement number m as the statement to branch to if an error occurs on the read (in this way, you may be able to recover gracefully from the read error and have your program continue, making your

program *robust*; otherwise, the program will just abruptly terminate upon encountering the error).

Note: You may also include an IOSTAT = *ivar* clause in the READ, which will store an integer value into *ivar* which indicates *which* type of error has been committed. See Chapter 11 for more details.

Thus, if a sequential file of information had been written out to disk using list-directed writes and stored as the file named 'INFO', and had been terminated by a special End-of-File record, we could write a program to read from it until the file of data was exhausted, using END=. Suppose that the data written to the file was a list of tensile strengths of various materials, and you wanted to read the data and calculate the average tensile strength and determine the greatest value in the list. Since we have already done separate problems similar to this, it should not be necessary to draw flowcharts for the procedure. The program illustrates the use of the END= clause.

```
***** READ TENSILE STRENGTHS FROM FILE 'INFO' *****
***** UNTIL DATA FILE IS EXHAUSTED *****
***** FIND AVERAGE AND GREATEST TENSILE STRENGTH *****
      REAL TENSIL, BIG, AVERAG, SUM
      INTEGER COUNT
      OPEN (3, FILE = 'INFO', STATUS = 'OLD')
      SUM = 0.0
      BIG = - 100.0
      COUNT = 0
 50  CONTINUE
      READ (3, *, END = 100) TENSIL
      SUM = SUM + TENSIL
      COUNT = COUNT + 1
      IF (BIG .LT. TENSIL) BIG = TENSIL
      GO TO 50
100 CONTINUE
      AVERAG = SUM/COUNT
      PRINT 111, AVERAG, BIG
111 FORMAT('0', 2X, 'THE AVERAGE TENSILE STRENGTH IS ', F6.1,
& ' AND THE GREATEST TENSILE STRENGTH IS ', F7.1)
      STOP
      END
```

Input READ Descriptors

The FORMAT statement describes the record to be read, beginning at the leftmost position (column 1), for as many columns as are of interest. Each variable on the read list must have a corresponding format descriptor in the

FORMAT statement, which indicates the type of the variable and how many columns the value occupies. Because input format edit descriptors parallel the output descriptors, we will cover them briefly here. There is no carriage control for input, since that only deals with positioning on an output device relative to the previous line that was output.

The format for integer input is Iw , where w indicates the width of the field from which the value will be read. Your system will have a built-in convention regarding how blanks are interpreted in a numeric field. On older versions of FORTRAN, they were always interpreted as zeros, but recent systems more often tend to ignore them. Check out the convention on your system. If you want to control the interpretation of blanks in numeric fields with your FORMAT, you can include a BZ editor, which indicates that they should be interpreted as zeros, or a BN editor, which indicates that blanks should be ignored. The interpretation of blanks can also be set in an OPEN statement, as will be discussed in Chapter 13. If the two FORMATS indicated are used to read the numeric field containing blanks (represented as b's), the results will differ as follows:

```
READ 4, INK
4  FORMAT(BZ, I5)
```

```
READ 5, INK
5  FORMAT(BN, I5)
```

and the input line is

bb7bb {the b's represent blanks}

the values stored will be 700 in the first case (FORMAT 4) and 7 in the second case (FORMAT 5, where blanks are ignored).

Columns to be skipped are indicated by X's, as in output, and repetition factors may be used (such as $6X$). Even if there is nonblank data in the field covered by the X's, it will be skipped. Occasionally this is useful, in cases where you *want* to skip over some parts of the input line. For example, if you were going to input dates in the form mm/dd/yy (month/day/year) and you wanted to allow the person entering the data to include the slashes as part of the date (for readability), you could use the following READ/INPUT duo:

```
INTEGER MONTH, DAY, YEAR
READ 44, MONTH, DAY, YEAR
44  FORMAT(I2,1X,I2,1X,I2)
```

This would read in the integers from the I fields and skip over the slashes (which would be illegal and cause an untimely program termination, if they appeared in a numeric field). The indicated READ and FORMAT would thus properly read the input:

and store a 12 in MONTH, a 25 in DAY, and a 99 in YEAR.

Any characters in the input record that are not in a field described by some input descriptor will simply be ignored. Thus an input format can simply "pick out" a value from a whole record filled with values. For instance, if you have an 80-column record filled with 80 one-column single-digit answers to a questionnaire, and you only want to pick out the 12th answer (which is in column 12), you could use the following:

```
READ 7, IT
7  FORMAT(11X,I1)
```

Note that the 11X is needed to skip over the first 11 one-digit answers, but the remaining 68 are simply ignored, since no input field covers them. Note that each new READ goes to a new record, even if there is data remaining on the record just read.

Real, or floating-point, values are input with an Fw.d format descriptor. If there is a decimal point in the field, the d specifier is ignored. However, if there is no decimal point in the field, the d specifier is used to place the decimal point before storing the value. Thus if the following is used

```
READ 9, B
9  FORMAT(2X,F6.2)
```

to read the input line

876543

the value stored in B will be 8765.43, with the 'd' (of the w.d field descriptor) placing the decimal point. However, if the same format (9) is used to read this input line:

12.345

the decimal point in the field will override the 'd' in the format descriptor, and the value 12.345 will be stored in B.

Real values may also be input using Ew.d or Dw.d format, in which case an exponent following an E or D, respectively, will be expected in the input field.

Complex values should simply be input as a pair of real values, read in F or E format. If you include a comma or parentheses in the input record, you should use X's to skip over these columns on the formatted READ.

Logical values are input with an Lw format descriptor, indicating to read from a field w columns wide. The value input should be a T or F, optionally followed by additional characters, and optionally flanked by dots; there may be blanks in the field.

Character values are read with an Aw or an A descriptor. If just A is used, the length of the field is taken to be the specified length of the character variable into which the value is to be read. If w is less than the length of the character variable, the input string is stored left-justified and blank-filled in the variable. If w is greater than the length n of the character variable, the *rightmost* n characters from the input field will be stored (notice that this is different than the way it is handled in a list-directed READ, as discussed earlier).

Thus in the following example:

```
CHARACTER A*3, B*4, C*5
READ 33, A, B, C
33 FORMAT(A, 1X, A3, 1X, A8)
```

if the following input line is read:

CAT AND TITMOUSE

the character string CAT will be stored in A (it will have been read as if by an A3 format, since the length of variable A was designated as 3), the string 'AND' will be stored in B (the 3-character string stored left-adjusted and filled with a blank), and the string 'MOUSE' will be stored in C (since C is only set up as a 5-character string, the rightmost 5 characters of the 8-character field read are stored).

The following READ statement fills the variables as shown:

INTEGER MAN, MUCH	MAN	754
REAL TALK	TALK	3.14
CHARACTER A*5, B*4	MUCH	15
READ 77, MAN, TALK, MUCH, A, B	A	HORSE
77 FORMAT(I3,3X,F5.1,I2,2A5)	B	LOBE
{input line:}		
754621 3.1415HORSEGLOBE		

Slash (/) in Input Record

In an input format, n slashes at the beginning or end of a format description indicate n records to be skipped. In the middle of the description, n slashes indicate n-1 records to be skipped. A skipped input record means that it simply will be passed over and not read from. For example, a skipped input record from the card reader is a card that is passed through the reader and its contents ignored; on tape or disk, it is simply a record (one written out as a single "line") that is passed by. In the following example:

```
READ 5, A, B, D
5 FORMAT(//5X, 2F5.1//10X, F5.1///)
12      3      4      5      678
```

a total of *eight* records are processed. The first two records are skipped by the leading two slashes; two real values are read from record 3; record 4 is then skipped (the two slashes in the middle indicate, respectively, the completion of record 3 and the skipping of record 4); one real value is read from record 5; and then three records (6, 7, and 8) are skipped. By the time all of this is completed, the input device is positioned ready to read from (or skip) the next (9th) record.

Repeated Formats on Input

If a READ list is too long for the FORMAT it is identified with, the format description must be reused until the list is filled (presuming that the input device does not run out of data). If there are interior parentheses in the format, after the first use, the description will only be repeated from the left parenthesis matching the innermost right parenthesis. Thus, for example, in the following case:

```
READ 6, {list of 50 integer variables}
6 FORMAT(10I8)
```

a total of 5 records will be read, 10 values from each record, each from 10 adjacent 8-column fields in that record. However, in the following example:

```
READ 7, {list of 50 integer variables}
7 FORMAT(I5, 2(2X, I3), 5(3X,I5))
```

eight values will be read from the first record, in just the layout the format description indicates, *but* after that, only five values will be read from each succeeding record, using the format 5(3X,I5), that is, 5 repetitions of 3 spaces followed by an I5 field for each of these records. Thus, this READ/FORMAT combination will try to read a total of ten records. (Do you see this? How many values will be read from the last record?)

Sample Program

You have been given the problem of displaying, in a visual form, integer data from an experiment in the order in which it occurs. You know that the data falls in the range from 1 to 100, and you are to output a line of asterisks (*)'s for each data value, with as many stars as the value of the data point. Data points will be read in, one value per record (written in an I3 format) until an end-of-file condition is encountered. The loop to continue reading is not difficult (we have done something of this kind before); the real difficulty appears to be in printing out a number of asterisks depending on the value read

in. We have not yet covered arrays, or implied lists, which will later give us two other approaches to the problem. But today we must write the program making do with what we know.

The string of asterisks will have to be output as characters, so we must think about what we have learned about character data. A single asterisk will not do, because we have no way of controlling how many repetitions of it will be output. But if we created a character string A containing 100 asterisks, then if we wanted N of them printed, we could output a subset of the asterisks of length N by printing A(1:N). We will have to make the field width of the output descriptor A100, to handle the largest possible output.

There is one drawback that we will have to live with for the time being. Perhaps after completing the next chapter we can come up with a more elegant solution. We learned that if a character value is output to a field wider than the length of the string, it will be printed *right-adjusted* in the field. Thus, our stars will be printed right-adjusted, when we really would have preferred them to appear left-adjusted. However, we still will get a visual display of the pattern of the data as sampled, which is really what we were aiming for. Thus we will create our string of stars, write the loop to read from the file, and output our distribution right-adjusted.

```
***** VISUAL DISPLAY OF DATA POINTS *****
CHARACTER A*100
DATA A/*****'/
&*****
INTEGER N
PRINT 44
44 FORMAT('1', 55X, 'DISPLAY OF DATA POINTS FROM EXPERIMENT')
OPEN (2, FILE = 'DATAPT', STATUS = 'OLD')
100 CONTINUE
      READ (2, 155, END = 200) N
155  FORMAT(I3)
      PRINT 188, A(1:N), N
188  FORMAT(5X, A100, I6)
      GO TO 100
200 STOP
END
```

The output data would look something like the following—acceptable, if not ideal.

*****	5
*****	10
*****	19
***	3

etc.

Notice that a PRINT* statement would print the stars aligned with the left margin, if you did not print the number (N) first, since that might take up different widths for different values, and thus throw off the asterisks' alignment. Can you think of a different way to approach this problem, using a character variable, which would print the histogram beginning at the left margin instead of the right? (*Hint:* begin with the character string containing blanks, and then fill its substrings as needed.)



SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

A FORMAT statement may be used to describe the layout of input or output records for your program. This statement is tied to an I/O statement by a statement number, as in:

PRINT 7, list	READ 8, list	WRITE(6,9) list
7 FORMAT(.....)	8 FORMAT(.....)	9 FORMAT(.....)

A *record* is simply the collection of information (usually in character form) that is output in one group, as for example, to one printer or terminal output line; or that is read in one group from, say, one punched card or one line entered at the terminal. The contents of the FORMAT statement *describe* how the input record is to be interpreted, or how the output record is to look. This can be done by using a variety of *edit descriptors* such as the following:

Carriage Control Characters—used only on output to the printer or terminal; there are four standard control characters:

'1' — top of a new page	'0' — skip a line
' ' — write to the next line	'+' — output to the same line

X (spaces)—for example, 3X means 3 blank columns on output, or 3 skipped columns on input

Iw—for integer values; on output, they will be written right-adjusted in a w-column field; on input, an integer value will be read from the next w columns of the input record (*Note:* Iw.m on output guarantees that at least m digits will be output)

Fw.d—for real (floating-point) values; on output, this will place a real right-adjusted in a w-column field, displaying d digits to the right of the decimal point; on input, a value will be read from the next w columns and, if there is no decimal point in the input field, a decimal point will be placed on storage which puts the d rightmost decimal places which were read in at the right of the decimal point

Ew.d—for real values; these will be output in a w-column field (if possible), with d digits to the right of the decimal point, a fractional multiplier, and an exponent (power of 10) written after an E in the field; on input, a real value (which may be expressed in Exponential notation) is read from w columns

Gw.d—for real values; on output, if the value is between 0.1 and 10^d , it will be written in F format, otherwise it will be displayed in E format; on input, behaves the same as *Ew.d*

Dw.d—like *Ew.d*, except used for double-precision values

Aw (or A)—for character values; on output, a character string will be output right-adjusted in a w-column field; if w is less than the length of the character string, the *leftmost* w characters of the string will be written (if w is not specified, and just A is used, the width of the field is the same as that of the character string); on input, a string of characters will be read from the next w columns of the input record—if w is less than the size of the character variable into which the string is to be stored, it will be stored *left-adjusted* and blank-filled; if w is greater than the length n of the variable, the *rightmost* n characters from the field will be stored

Lw—for *logical* values; on output, a T or an F, as appropriate, will be output right-adjusted in a w-column field; on input, a logical value (which must contain either a T or an F, optionally flanked by .’s and/or other characters) is read from the next w columns of the input record

/ (slash)—for skipped input or output records; n slashes at the *beginning* or *end* of a format description indicate n skipped records; n slashes in the interior of a format description indicate n-1 skipped records

Repeated formats—if an input or output list is *longer* than the format description will accommodate, the format is *repeated* until the list is exhausted. If there are interior parentheses in the format description, after the first use, it will only be repeated from the left parenthesis matching the first interior right parenthesis

If a numeric value is *too large* for the specified output field, all asterisks (*’s) will be written in the field.

Other format specifications—since a format merely gives a character string specifying the I/O form, such a string may be specified directly through an alternate method:

PRINT 7, list	CHARACTER A*50
PRINT '(...)', list	A = '(...)'
7 FORMAT(...)	PRINT A, list

END= and *ERR=*—in an input READ statement, the control list may optionally include these specifications:

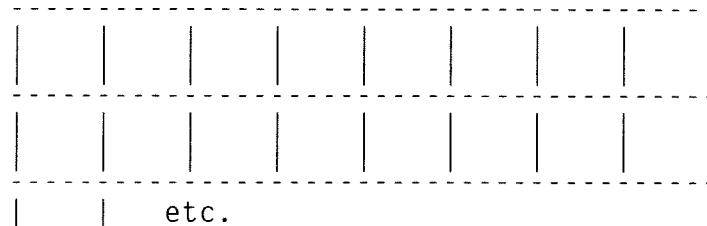
READ (u, f, END = n, ERR = m) list

where u is the unit number read from, f is the format statement number, n is the number of a statement to branch to if the END-OF-FILE record is encountered, and m is the number of a statement to branch to if an error is encountered on the READ.



EXERCISES

1. Try out the '+' carriage control (*carefully*). Print out your name (using A format) on a line, and then use another print statement to output your birthday to the right of your name on the same line. Remember that you have to space past the name already written on the line before beginning printing.
2. Use the slash format to print a message in the *middle* of the page (both horizontally and vertically).
- ♦ • 3. Use your knowledge of formatting techniques to print the pattern that would represent a chessboard—8 x 8 squares, looking something like this:



4. Experiment to determine how many significant digits your computer stores. Take a familiar fraction such as 1/3, compute it as a real and store in a real variable, then print it out to about 20 decimal places; how many are significant? Then try the same experiment in double precision.
5. Given the following input record, which begins with a 3 in column 1, what will the indicated READ statements store in the values on their read lists? Indicate any cases in which an execution error would occur because of format mismatch.

```

34567891233.333 765 321005551212FAREWELL
CHARACTER A*4, B*4, C*10
INTEGER K, L, M, MAD, MAX
REAL X, Y, Z
READ 22, K, L, M, X, Y, MAD, MAX, A, B

```

236 FORMATTED INPUT/OUTPUT

```

22 FORMAT (3I3, F6.1, F4.1, I5,I8, 2A4)
  READ 33, K, L, M, MAD, MAX, C
33 FORMAT (2I5, 3X, I8, 7X, I4, A)
  READ 44, K, L, M, MAD, MAX
44 FORMAT (5I8)
  READ 55, A
55 FORMAT(33X, A3)
  READ 66, K, L, M, MAD, MAX, X, C
66 FORMAT(5I1, 6X, F3.2, 10X, A8)

```

6. Practice, as a paper-and-pencil exercise (unless your instructor will create such a file for you, or you wish to read ahead in Chapter 11 to create one yourself), writing a loop which will terminate when a special end-of-file record is encountered, and then go on and process the data. Imagine that someone has written out a file of names and GPAs to tape, and placed an end-of-file mark at the end of the data. Write a program which will read in data until the end-of-file is encountered, then print out who had the highest GPA and go on to calculate the average GPA. After you have done this, REWIND the file (issue the command REWIND u for the unit u you are reading from), reread all the data, and calculate the standard deviation of the GPAs, now that you have calculated the average A. The standard deviation can be calculated from the formula:

$$\text{S.D.} = \sqrt{\sum_{i=1}^N \frac{(G_i - A)^2}{N}}$$

where N is the number of GPAs processed.

Also practice a pencil-and-paper program that will branch to a part of your program that can go on without the data if an error occurs on the READ.

7. What will the following statements print out?

```

CHARACTER A*5, B*3
INTEGER MAD, MAX, MUTT, JEFF
REAL X, Y, Z
DATA A, B/'HORSE','COW'/ MAD, MAX / 100, 12345 /
X = 1.0/6.0
Y = .234 E 4
Z = 0.3 E -4
MUTT = - MAD
JEFF = 10**7
PRINT 6, A, A, A, B

```

```

6 FORMAT('0',5X, A2, 1X, A2, 1X, A6, A)
   PRINT 7, X, Y, Z
7 FORMAT(6X, F6.5, 2X, 2F9.4)
   PRINT 8, MAD, MUTT, MAX, JEFF
8 FORMAT(4X, 2I3, 2X, 2I5)
   PRINT 9, Y, Z, JEFF
9 FORMAT(6X, F10.0, 2X, F8.6, 2X, I9)
   PRINT 10, MAD, MAX, MUTT, JEFF
10 FORMAT('0', 2X, I7)

```

- 8. You are given data to process which has already been written out to disk, in a file named 'CASH'. The disk contains answers to 1000 questionnaires, and each person's answers take up four 80-column records. You only want to read *one* answer from each person, a one-digit integer appearing in column 45 of the second record for that person, *and* you want to read each person's name, which is in columns 61-80 of the last (fourth) record. Write a program which will process all 1000 questionnaires, print out the name of anyone whose answer to the question was 8, and *count* how many of the answers were 2, and how many were 6.
- 9. A galaxy has been observed to be moving away from the earth at a speed of 21,600 km/sec, and it is currently 1.4×10^9 light-years from us. Work backward, indicating its distance from earth 100,000 years ago, 200,000 years ago, and so on, up to a million years ago. Then calculate its distance from the earth 5 million, 50 million, 500 million, and 5 billion years ago. Assume that it has been travelling at the same velocity all along. The speed of light is roughly 300,000,000 m/sec. Make your output well-labelled and informative. Choose an output format for the distances that is meaningful and easy to read. Indicate the number of years as an integer, though you may have to switch to a real representation for 5 billion years, depending on the word capacity of your machine.
- 10. Using the technique developed in the problem at the end of the chapter, write a program which will print the pattern:

```

      X
      XX
      XXX
      XXXX
      XXXXX
      XXXXXX
      XXXXXXX
      XXXXXX
      XXXXX
      XXXX
      XXX
      XX
      X

```

Allow the length of the longest section to be controlled by an input value up to 40.

- 11. Making use of the CHAR and ICHAR functions, write a program which will read in text all in uppercase letters, and convert the entire text to lowercase and print it out.

Modify this program so that it tests to be sure the character is in the uppercase letter class ('A' through 'Z') before changing it to lowercase; thus your program should leave other symbols in the text (such as '.', ',', ':', etc.) unchanged.

- 12. The distance travelled in time t by an object starting at initial velocity v and undergoing a constant acceleration a is:

$$x = vt + \frac{1}{2} at^2$$

The velocity achieved at the end of this time period is:

$$v' = v + at$$

Write a program which will accept an initial velocity and an acceleration, and output the distance covered and the velocity for values of time t from 1 to 50 seconds.

- 13. Rewrite the "gasoline-pump" output example from this chapter so that it continues filling the tank, displaying cost every 20-cent increment, until the total cost is \$15.00. This will give you practice using the special effects of '+' to the terminal screen.

- 14. To convince yourself of the difference between *formatted* and *list-directed* READS, experiment entering the following character strings: 'BIO', 'EGO', 'QUARK', and 'CHARM' to the character variables A, B, C, and D, respectively, which are:

CHARACTER A, B*2, C*3, D*4

First read them in using READ* (in which case each character string must be enclosed in single quotes on input), and then read them in (without quotes) using an A5 format for each. Print out the different results stored in your variables in each case.

- 15. Print out the positive integers from 1 through 100, ten to a line, such that there is exactly one blank column between each of the consecutive integers on a line.
- 16. Scientific handbooks contain tables of trigonometric functions. Using the system functions SIN, COS, and TAN, print out an informatively labelled table of the sine, cosine, tangent, and cotangent (l/tangent) of angles from 0° through 90° , in steps of 5° (*Note*: the trig functions take angles expressed in radians as their arguments. There are 2 pi radians in 360° .)

17. The ENIAC was designed to create ballistics tables. Write out a table of maximum range (no air resistance) for "Big Bertha" with muzzle velocity of 4800 ft/sec for angles 10° – 60° .

18. The following table indicates the compatibility of blood donors and recipients, based on their blood types. From this chart you can see that O is the universal donor, and AB is the universal recipient (can accept any blood type). Write a FORTRAN program that will read in the blood type of the person who needs blood, and print out the type of blood needed from a donor.

Recipient	Donor
AB	Any type
A	O or A
B	O or B
O	O

19. If you want to see all of the printable characters available on your system, write a loop (from 0 to 127 if your system is ASCII, from 0 to 255 if your system is EBCDIC—see Appendix D) and, using these values as n in the CHAR(n) function, return and print out the character for each n (using A format). There will be several cases where there is no printable character. If you can figure out how to do so, have your table include not only the printable character and its position in the *collating sequence*, but its equivalent octal and hexadecimal representations as well, as they are displayed in Appendix D. Use Appendix A on Number Systems to figure out the octal and hexadecimal equivalents.

CHAPTER 7



ARRAYS—SUBSCRIPTED VARIABLES

Arrays allow you to handle groups of information of the same class in a very efficient manner. Without arrays, there are many interesting problems you simply could not solve in FORTRAN. Once you have mastered arrays, you can learn techniques of sorting and searching, which are essential to much of the work done on computers.

The Very Large Array (VLA)
radio telescope scans the sky
in Socorro, New Mexico.

"A place for everything, and everything in its place."

- Samuel Smiles, Thrift, Chapter 5

◆ WHY ARRAYS?

We have developed the capability to write a number of interesting programs with the tools acquired so far, but our ways of representing data are still rather limited. We can store and manipulate values in single named locations of the six available data types in FORTRAN, but that is all. What if we have a large group of related data, such as 100 weights of precipitate, from a chemical experiment, to process? Further, we want to be able to store all of the weights in memory at once, because after we have read them in and added them up to take the average, we want to go on and calculate the standard deviation of the weights (this is similar to what we did in Exercise 6 of the previous chapter, and for which we need all of the GPAs *and* the average), and then later *sort* them into descending order to examine the weight distribution. How are we to do this effectively? To store each weight in memory requires that each one have a unique name—shall we name them WA, WB, WC, and so on? But how can we take advantage of the fact that we want to do the *same* thing to each member of the group, such as add it into the sum of weights, look at it to calculate the standard deviation, and so forth?

The answer is the *array*. We need to give each weight in the group a unique name, but also take cognizance of the fact that they are members of the same group. Thus, we will give the group a name, and the elements of the group will be distinguished according to their position in the group (first, second, etc.). For example, we can call the grade group WEIGHT, and the elements of the group (array) WEIGHT(1), WEIGHT(2), WEIGHT(3), and so on.

◆ ONE-DIMENSIONAL ARRAYS (LISTS, VECTORS)

If we were to use mathematical notation to indicate that we wanted to add up all n elements of a group, called X, we would write that we wanted the summation:

$$\sum_{i=1}^N x_i = x_1 + x_2 + \dots + x_n$$

where each element of the group, or vector, or list, was distinguished by its *subscript*, a number which represented its position in the collection. We will use a similar notation for arrays in FORTRAN, but since we do not have the capability of writing subscripts below the line, we will instead put the subscript

in parentheses following the group name. Instead of writing X_1 , X_2 , we will write $X(1)$, $X(2)$. The “subscript” here, the value appearing in parentheses after the name of the group or array, can be any integer expression (constant, variable, or arithmetic expression) and its value will determine the position of the value in the array. There is one other thing we must do, however, before referring to subscripted array variables in our program—we must *dimension* the array, that is, set aside consecutive locations in memory for it to occupy. This may be done either with a DIMENSION statement or in a type statement.

To alert the compiler that our program will be dealing with groups of values, or arrays, we must tell it at the beginning of the program unit that there will be arrays of certain sizes in the program. A DIMENSION statement or a type statement can specify a certain number of locations to be set aside. Since modern programming practice encourages explicit typing of all program variables, the DIMENSION statement is becoming less useful, and is deemed outmoded in the proposed Fortran 90 revision; this means that it may be removed in the next revision, which may occur near the end of the decade/century. However, in the meantime, it is still around, and so we will introduce you to both means of *declaring* the size of an array. It is either:

INTEGER MAD	or	INTEGER MAD(500)
DIMENSION MAD(500)		

Both of these statement combinations declare to the compiler that the program will use an integer array called MAD which will contain 500 locations. The compiler then sets aside 500 sequential locations in memory under the name MAD, which will have the individual names MAD(1), MAD(2), MAD(3), . . . , MAD(500). These locations can then be used to store and manipulate values just like any ordinary simple variable. The great advantage is that, if the same operation is to be done to every member, or to some consecutive subgroup, of the array, it can be done simply and elegantly in a loop.

Before FORTRAN 77, indicating the array dimension as a single integer (n) was the only form available, indicating that the number of elements in the array was n and that they were to be subscripted from 1 to n . In FORTRAN 77, the programmer can specify a lower limit and an upper limit to the subscript range for the array, in the following manner:

DIMENSION {or type} name(low:high)

where low and high are both integer constants or symbolic integer constants (as defined in a PARAMETER statement). Thus arrays may be dimensioned as in these examples:

```
REAL BAD(5:50), CAT(-4:4)
DIMENSION FAT(200:300)
INTEGER REF(0:9)
```

The array BAD will have 46 locations, named BAD(5), BAD(6), ..., BAD(50), and CAT will have nine locations, named CAT(-4), CAT(-3), CAT(-2), CAT(-1), CAT(0), CAT(1), and so on. The array FAT will have 101 locations, from FAT(200) through FAT(300), and the integer array REF will have ten locations, from REF(0) to REF(9).

Note: In Fortran 90, the DIMENSION statement will be largely obsolete (though it will still be included in the compiler to maintain the upward compatibility of earlier programs), since most array size declarations will be made in type statements, as we have indicated here. The additional feature in Fortran 90 is that several arrays may be declared at once to have the same configuration, by using the :: notation in the type statement:

```
REAL, DIMENSION (-5 : 15) :: A, B, C
INTEGER (0 : 100) :: MAX, REF
```

Values may also be initialized in arrays using the type array declaration form:

```
INTEGER (10) :: X = (/1,2,3,4,5,5,4,3,2,1/)
```

Rules for Subscripts

We have already indicated that a subscript may be any legal integer expression. [Note: This was not true in earlier versions of FORTRAN, which restricted subscripts to the following forms, where k and c represent integer constants and v represents an integer variable; in these compilers, a subscript could only be: c, v, k*v, k*v + c, or k*v - c. If you must deal with an earlier compiler, or with a program written for one, you should be familiar with these limitations. It even happens occasionally that a “glitch” in a FORTRAN 77 compiler in dealing with subscripts is tied to these older rules, which makes it useful to be familiar with them.]

You *should* be able to use any legal integer expression as a subscript in FORTRAN 77, *as long as its value does not go outside of the dimensions of the array*. If your subscript exceeds the array dimensions, you will probably get an execution-time error. This is because storing beyond the array dimensions (that is, the space that the compiler has set aside for the array) would overwrite some other location in your program—another variable, or even an instruction. If your system does *not* give you such an error, you may accidentally override some other, important part of your program, causing damage that is very difficult to find and debug.

Note: In Fortran 90, a subscript notation may be more complex, allowing it to refer to a segment of an array. This notation is a *subscript triplet*, of the form:

$$(\text{[lower bound]} : \text{[upper bound]} : \text{[stride]})$$

where each expression must be a scalar (non-array) integer value, and each

is optional. This allows the programmer to specify a range of subscripts referred to (from *lower bound* to *upper bound*) and, optionally, to specify a step size (*stride*) between them. If either the lower bound or the upper bound is omitted, the actual lower or upper bound of the declared dimensions of the array is used. If no stride is specified, the step size is assumed to be 1. Thus for an integer array A dimensioned to 10:

INTEGER A(10)

- the statement A = 6 would set every element of the array to the value 6. A reference to A(2:5) implies locations A(2), A(3), A(4), and A(5); however, A(:3) refers to A(1), A(2), and A(3); A(8:) refers to A(8), A(9), and A(10). Finally, a reference indicating a stride, such as A(3:9:2) indicates A(3), A(5), A(7), and A(9). Such references can be made in assignment, DATA, or I/O statements.

Loops With Arrays

Since a subscript can be any integer expression within the range of legal subscripts for the array, we can have a loop with an integer loop control variable access all the locations (or a given subset) of the array. For example, if we assume that there are 200 weights (reals) stored out on tape (we will have declared it to be on unit 4), and we wish to read them in and average them we might proceed as follows. The values stored on the tape begin with 66.0, 75.0, 88.5, . . . , and end with 94.0, 99.0. We show in the following diagram how the program stores them in consecutive locations in memory identified under a single array name (in this case, WEIGHT). The numbers at the left of the diagrammed list refer to the *positions* in the array, or the numeric *subscripts* for those positions; the values on the right (in boxes) are the actual numbers that are stored in the array. Thus, the value 75.0 (the second grade read in) is stored in WEIGHT(2), and so on.

	WEIGHT
REAL WEIGHT(200)	1 66.0
SUM = 0.0	2 75.0
DO 10 N = 1, 200	3 88.5
***** ONE WEIGHT STORED PER RECORD *****	4 96.5
READ (4,*) WEIGHT(N)	. .
SUM = SUM + WEIGHT(N)	. .
10 CONTINUE	199 94.0
AVER = SUM/200.0	200 99.0

The diagram at the right indicates how the array is stored in memory as 200 consecutive locations (the size to which the array was dimensioned). When

the program must access a location in the array, say WEIGHT(N), it simply looks at the Nth position in this group; that is, it accesses an address which is N-1 below (that is, greater than) the address of the beginning of the array (WEIGHT(1) in this case).

After we complete this part of the program segment, we will have calculated the average of the 200 weights, and all 200 will be stored in the array WEIGHT, so we can process them further, if desired. Let us say that we wanted to go on and calculate the standard deviation of the weights, a sort of average distance of the weights away from the average. Our mathematics tells us that, for an array X with average A, the standard deviation is:

$$\text{S.D.} = \sqrt{\sum_{i=1}^N (X_i - A)^2 / N}$$

Thus, we can add onto our program the instructions to calculate the sum of the squares of the distances of each value from the average, and then calculate the standard deviation, as follows:

```
SUMSQ = 0.0
DO 20 I = 1, 200
    DIST = WEIGHT(I) - AVER
    SUMSQ = SUMSQ + DIST**2
20 CONTINUE
SD = SQRT(SUMSQ/200.0)
```

Compare this with the situation in exercise 6 at the end of the previous chapter. In that case, without arrays, we had to go back and *reread* the data after we had calculated the average, in order to find the standard deviation. Thus, given the restriction that these are the operations we must perform, and in that order, our current solution using arrays is much better. Actually, this problem as formulated is a little deceptive. We have assumed that we need to calculate the average *first*, and then go back and compare it with all of the weight entries to calculate the standard deviation. This is because of the form of the equation we used:

$$\text{S.D.} = \sqrt{\sum_{i=1}^N (X_i - A)^2 / N}$$

However, there is another formulation of the equation for standard deviation, as:

$$\text{S.D.} = \sqrt{\frac{\sum X_i^2}{N} - \frac{(\sum X_i)^2}{N^2}}$$

Notice that, in this formulation, we can calculate the average (it appears in the second term, squared), but we do not have to do it *first*. As we are going along in the READ loop, we could be accumulating a sum of the squares of the values (the first term) as well as the sum of terms. We could then calculate the standard deviation without ever having to go back and look at the list of values again. Brief modifications to our WEIGHTs program to accomplish this would be to insert the accumulator:

```
SUMSQR = 0.0
```

after the `SUM = 0.0` line, insert the command:

```
SUMSQR = SUMSQR + (WEIGHT(N))**2
```

after the line `SUM = SUM + WEIGHT(N)`, and then, after we have calculated the average, `AVER`, insert the line:

```
SD = SQRT(SUMSQR/200.0 - AVER*AVER)
```

We could thus have calculated the standard deviation more efficiently, and we would not have had to resort to the use of arrays (or rereading the data) to do the job, if we had only been aware of this more elegant approach. However, since we have used this device to introduce arrays, let us not throw the idea away, since we will find that they have many other valuable applications. For example, we could still expand this program to go on and sort the collection of weights into descending order (largest to smallest), which we will learn how to do in a later section. If they were sorted, we could then find the *median* (or middle), representative weight.

In a similar example, after calculating the average of a set of test grades, we might determine that it is too low, and decide to "curve" the grades upward (or too high, and so scale them downward). Assume that the teacher processing these grades takes 70 to be the ideal average for student grades. Thus, if this set of grades has an average different from 70 by any significant amount, the grades should be "massaged" so that they *do* average out to 70. To do this, we would find the difference between the average and 70, and "adjust" the grades in the list accordingly. Let us say that if the average is within one point of 70, the teacher will leave the original grades alone, but otherwise they must be curved. To accomplish this, we would add the following section, after calculating the standard deviation:

```
***** CURVE THE GRADES IF NECESSARY *****
IF ( ABS(AVER - 70.0) .GT. 1.0) THEN
    DIFF = 70.0 - AVER
    DO 30 I = 1, 200
        GRADE(I) = GRADE(I) + DIFF
30 CONTINUE
ENDIF
```

Notice that arrays are also very useful for containing *counter* values, since a set of N counters can be handled in an array with N locations. Reexamine problem 17 at the end of the previous chapter; we can write it much more simply by using an array of counters KOUNT, dimensioned to 10, instead of ten unique variables. The problem is to read in 800 measurements in the range from 101 to 200 grams, and count how many fall into each 10 gram range, printing out a histogram of the results.

```

PROGRAM HIST02
CHARACTER A(120)
INTEGER KOUNT(10)
REAL GRAMS
DATA A/ 120*'*/, KOUNT/10*0/
DO 20 I = 1, 800
    READ*, GRAMS
    KSPOT = (GRAMS - 100.01)/10. + 1
    KOUNT(KSPOT) = KOUNT(KSPOT) + 1
20 CONTINUE
DO 30 J = 1, 10
    PRINT 33, (A(L), L = 1, KOUNT(J) )
33   FORMAT ('0',3X, 120A1)
30 CONTINUE
STOP
END

```

Note: Regarding the DATA statement used to initialize arrays, refer to the section on "DATA Statement Initialization of Arrays" near the end of this chapter.

We were able to use a formula to calculate which of the counter slots a particular weight fell into; do you understand the way the formula works?

Lists and Vectors

We have already seen an example of a one-dimensional array used to store a list of related values (grades). Arrays can be used to store many other lists as well: mailing lists (alphabetic), lists containing the months of the year or the days of the week, lists of experimental data values, lists of illegal credit card numbers, temperature readings by the hour or over the last 365 days, lists of movies available on video tape (character entries), or any other groups of related numeric or character data you like.

A *vector* (of length n) is the set of coordinates of a point in an n-dimensional space. Thus, our most common examples would be the coordinates of a point (A) in two-dimensional space (x, y), or (B) in a three-dimensional space (x,

y, z). These could be represented, respectively, by arrays with two and three locations, as indicated in the following expression:

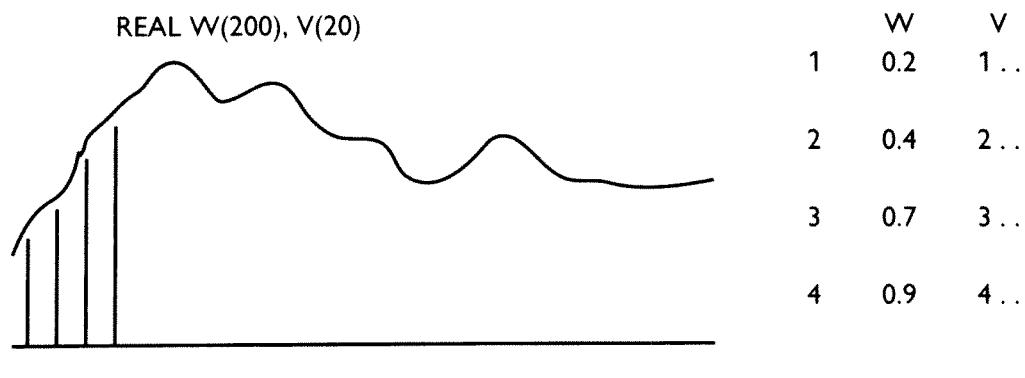
```
REAL A(2), B(3)
```

The vector can be thought of as a line from the origin $(0, 0)$ or $(0, 0, 0)$ to the point described. Further, our point A in two-dimensional space has a relationship with the complex values we have been using; its two values are the real and imaginary parts, respectively, of the point in the complex plane:

```
COMPLEX ZA
ZA = CMPLX ( A(1), A(2) )
```

We can even extend this beyond our geometric intuition of two- and three-dimensional spaces to consider a vector as a point in "n-space," representing an entity with n defining conditions or measurements. For example, if we were working on a sophisticated pattern-recognition project, to discriminate faulty jet engines from good engines by their sound patterns, we might determine 10 or 20 significant measurements that could be made on the waveforms representing their audio output (such as average height of peaks, average distance between peaks, greatest distance from a high to a low, and so on). These 10 or 20 measurements could then be stored in a one-dimensional array, or vector, V, defining the characteristics of each waveform we examined; these vectors could then be compared. Further, since the actual waveforms involved would be continuous, we would have had to sample them at periodic points to obtain a discrete representation that could be handled on our discrete machine, and these sampling points could also be stored in an array.

As a matter of fact, Shannon's Sampling Theorem says that for such a continuous signal that lasts t seconds, and which has a frequency from 0 to w cycles per second, that $2tw$ discrete measurements of the signal are sufficient to characterize it (because of the harmonic composition of such a sound wave). Thus, we could have one array (W) containing the sampled waveform points (giving the y-values at $2tw$ uniformly spaced x-coordinate points), and a second array (V) containing the 10 or 20 significant measurements we will compute for this waveform.



Of course, the values in V would be *computed* in a program by performing the appropriate measurements on the waveform. V would then represent a vector in 20-space, giving 20 characteristic values of the waveform, which could be then compared to the vectors of other jet engine waveforms.

There is an optional section on further vector manipulations included at the end of this chapter, for readers whose interests and/or background carry them along such lines.

I/O of One-Dimensional Arrays (and the Implied List)

We indicated in the previous section a very simple way to input values into an array, one value per record (line). However, such values are not usually stored or entered in this manner, since it is very wasteful of space. More often, many values are entered on one line or record; several lines or records (but not as many as for one item per record) represent all of the data. This means that we need the capability of indicating that we want to read in a longer list all at once—in fact, a list containing all the items in the array. Since it would be very time-consuming if we had to write out the name of every entry in the array on our READ list, there has to be a better way!

In fact there *is* a better way, and we will refer to it as the *implied list*. It looks and operates somewhat like a DO loop structure, but its form should never appear in a DO loop, which has its own unique syntax. We can create a list of variables (or other expressions) to be repeated under certain conditions specified by a structure that looks like the latter part of a DO statement, that is, of the form: var = init, final [,step]. The item[s] to be repeated under this control, and the control itself, are then enclosed in parentheses. For example, to create a list of all of the items in the real array A, dimensioned to 150 locations, we write:

(A(K), K = 1, 150)

Such an implied list may appear in an I/O statement (a READ or a WRITE or PRINT), or in a DATA statement, *but nowhere else*. It actually creates a list, under the implied list control, of all of the items (in our case, the A(K)'s) for K = 1 to 150; the list actually created by the compiler for this is effectively A(1), A(2), A(3), . . . , A(150), with *all* of the entries filled in. This list of variables is then the list to be filled or output by the I/O or DATA statement. It is a neat, convenient shorthand, making things much easier for the programmer. If only part of an array is to be filled or output, that can be done as:

```
REAL BIG(300)
READ 7, (BIG(N), N = 101, 200)
7 FORMAT (10F7.2)
```

These instructions fill only the middle third of the array BIG. locations BIG(101) through BIG(200). Notice that the FORMAT statement indicates records containing ten (10) real values in each. Since the I/O list is longer than the FORMAT, the format description will be repeated as many times as necessary to fill the list of variables (that is, 100/10, or ten, times). That is, ten successive records will be processed by the READ.

An example such as that which fills the middle third of array BIG must always contain a full-blown implied list control. However, if you want to execute I/O or fill an array with a DATA statement and the entire one-dimensional array is involved, there is an even more convenient shorthand in FORTRAN. Just refer to the one-dimensional array by name in an I/O or DATA statement, and the compiler will assume that you intend the entire array, in the order in which it is stored in memory. Thus, the following statements are equivalent:

<pre>DIMENSION NATURE(300) READ 8, (NATURE(I),I=1,300)</pre>	<pre>DIMENSION NATURE(300) READ 8, NATURE</pre>
--	---

Both the statements on the left and the right are interpreted by the compiler to create a list of *all* of the elements in the integer array NATURE, and fill them using FORMAT 8.

Implied lists can be used to set up repetitions of patterns other than array elements, as the following examples indicate:

```
PRINT*, (K, K = 4, 10)
```

This PRINT* statement will output the values of K while K goes from 4 to 10, that is:

```
4 5 6 7 8 9 10
```

The statement

```
PRINT*, (M**2, M = 3, 9, 2)
```

will output the values of M**2 for M from 3 to 9 in steps of 2, and the output will be:

```
9 25 49 81
```

The instruction

```
PRINT*, (5, L = 1, 3)
```

will output the value 5 three times:

```
5 5 5
```

You may find such implied lists useful in creating interesting outputs, particularly for labels and the like.

As we indicated in the formatted input/output chapter, you must be careful of interior parentheses in format descriptions that must be repeated in a case where the list is longer than one use of the format will accommodate. If there are interior parentheses in the format, the entire format will be used the first time, but thereafter, it will only be repeated from the left parenthesis that pairs with the rightmost inner right parenthesis. Thus, if the integer array NUMB is filled as indicated and printed out with the PRINT/FORMAT combination indicated, the results will look strange:

```
INTEGER NUMB(20)
DO 6 N = 1, 20
6      NUMB(N) = N
      PRINT 7, NUMB
7      FORMAT('0', 5X, 5(2X,I3))
```

The output from this program will skip a line, and then will be:

	1	2	3	4	5
6	7	8	9	10	
11	12	13	14	15	
16	17	18	19	20	

Thus, you must be very careful with the use of interior parentheses in FORMAT statements, and you must be aware of their potential effects if the format must be repeated to fill a list. The format in the previous example could have been rewritten to avoid the inner parentheses, and have had the desired effect, as:

```
7      FORMAT('0',5X, 5I5)
```

As long as integers to be output are no more than 3 digits long, an I5 format descriptor will have the same effect as a (2X, I3).

The Visual Display of Data Points Program Revisited

Now that we have acquired the array as a tool, and also learned how to output a list of values from an array, we can improve the program written at the end

of the previous chapter to display a visual histogram of the magnitudes of data points read in. Instead of using a single-character variable A containing 100 asterisks, we can use a character array A of 100 locations, each containing a star. This will be easier to fill, and its use will allow us to print the displays *left-adjusted*, as we wanted to do originally. The program should be rewritten as follows:

```
***** VISUAL DISPLAY OF DATA POINTS, VERSION II *****
CHARACTER A(100)
DATA A/100*('*')/
INTEGER N
PRINT 44
44 FORMAT('1', 30X, 'DISPLAY OF DATA POINTS FROM EXPERIMENT')
OPEN (2, FILE = 'DATAPT', STATUS = 'OLD')
100 CONTINUE
      READ (2, 155, END = 200) N
155  FORMAT(I3)
      PRINT 199, N, (A(I), I = 1, N)
199  FORMAT(3X, I6, 3X, 100A1)
      GO TO 100
200 STOP
END
```

The output from the program might look like, for example:

```
5      *****
10     *****
19     *****
3      ***
```

etc.

Reference Arrays

An array can be used to contain data that will be read in, or be filled with data that will be output, such as generating a list of all the possible combinations of the letters 'C', 'A', and 'T', each taken only once. An array may also be used as a *reference array*, into which values are stored by one part of the program, and then are referenced—or used—by another part of the program. One example of this might be a program which calculates and stores a list of factorials, where the factorial of n ($n!$) is the product of the integers from 1 through n. These factorials might then be used in calculating the number of combinations of m things taken n at a time, $C_{m,n}$:

$$C_{m,n} = m!/(n! (m-n)!)$$

Since factorials are products of integers, it would make sense to make the factorials themselves integer, and we will do this. However, since factorials get very big very fast, we will limit the values we store to no greater than $12!$, since that integer value will fit into the memory word on most mainframes. Since we will be calculating these factorials in order, we will make use of the fact that factorials may also be defined *recursively*, giving the following definition:

$$n! = n \times (n-1)! \quad \text{where } 0! = 1$$

Thus, if we begin with $0!$ defined as 1, we can fill the rest of the array using the previously defined element. We will then allow the user to input a pair of values for m and n , and we will output the number of combinations of m things taken n at a time.

```
***** FACTORIALS AND COMBINATIONS *****
***** MAKING USE OF A REFERENCE ARRAY OF FACTORIALS *****
      INTEGER M, N, NFACT(0:12), COMB
      CHARACTER ANS*3
***** FILL THE REFERENCE ARRAY *****
      NFACT(0) = 1
      DO 15 N = 1, 12
         NFACT(N) = N*NFACT(N-1)
15   CONTINUE
***** NOW HAVE THE USER INPUT PAIRS OF COMBINATION VALUES *****
20   CONTINUE
      PRINT 25
25   FORMAT(//-'0', 5X, 'PLEASE INPUT TWO INTEGER VALUES M AND N'
$ , ' TO DETERMINE THE NUMBER OF COMBINATIONS OF M THINGS',
$ ' TAKEN N AT A TIME. PLEASE CHOOSE M AND N UP TO 12.')
      READ*, M, N
      IF (N .GT. M) THEN
         COMB = 0
      ELSE
         COMB = NFACT(M)/(NFACT(N)*NFACT(M-N))
      ENDIF
      PRINT 30, M, N, COMB
30   FORMAT('0',4X,'THE NUMBER OF COMBINATIONS OF',I3,' THINGS',
& ' TAKEN',I3,' AT A TIME IS',I12)
      PRINT*, 'DO YOU WANT TO ENTER ANOTHER PAIR OF VALUES?'
      PRINT*, 'ENTER YES OR NO'
      READ 33, ANS
33   FORMAT(A3)
      IF (ANS .EQ. 'YES') GO TO 20
      STOP
      END
```

However, this program uses an algorithm which is very inefficient, and has serious magnitude limitations as well, since we cannot deal with any combinations which involve factorials greater than 12!. Imagine that we wanted to calculate the number of combinations of 52 things taken 5 at a time (that is, the number of possible poker hands—a reasonable request). If we look at the formula for this, it gives us:

$$C_{52,5} = \frac{52!}{5! 47!} = \frac{1 \times 2 \times 3 \dots \times 47 \times 48 \times 49 \times 50 \dots \times 52}{5! 1 \times 2 \times 3 \dots \times 47}$$

We thus see that the lower factorial can “cancel out” part of the upper term, and many fewer multiplications need to be performed. Thus, a better way to approach the problem would be to see which factorial in the denominator is larger, cancel it into the upper term, and proceed from there. Our program might be written:

```
***** PROGRAM TO CALCULATE COMBINATIONS *****
INTEGER M, N, NF, MMNF, COMB, NUM, DEN
CHARACTER ANS
20  CONTINUE
      PRINT 25
25  FORMAT(//,'0',' PLEASE ENTER M AND N FOR C(M,N)')
      READ*, M, N
      IF (N .GT. M) THEN
          COMB = 0
      ELSE
          NF = N
          MMNF = M - N
          IF (NF .GT. MMNF) THEN
              MTEMP = NF
              NF = MMNF
              MMNF = MTEMP
          ENDIF
          NUM = 1
          DO 30 I = MMNF + 1, M
              NUM = NUM*I
30  CONTINUE
          DEN = 1
          DO 40 I = 1, NF
              DEN = DEN*I
40  CONTINUE
          COMB = NUM/DEN
      ENDIF
      PRINT 45, M, N, COMB
```

```

45      FORMAT('0 THE NUMBER OF COMBINATIONS OF ', I3,
&          ' THINGS TAKEN ', I3, ' AT A TIME IS ', I8, '.')
      PRINT*, 'DO YOU WANT TO ENTER ANOTHER PAIR OF VALUES?'
      PRINT*, 'ENTER A ''YES'' OR A ''NO'' IN SINGLE QUOTES'
      READ*, ANS
      IF (ANS .EQ. 'YES') GO TO 20
      STOP
      END

```

Refer also to problem 24 at the end of this chapter for another approach to the combinations problem, using arrays.

You will find any number of uses for reference arrays as your programming career progresses. We will suggest several exercises in which you can make use of reference arrays.

Prime Numbers and Factorization

A *prime number* is a positive integer which has only 1 and itself as divisors. Thus, 2 is a prime number, but no other even numbers are prime. Odd numbers such as 3, 5, and 7 are prime, but 9 is not a prime, and so on. There is an elegant proof that there is no greatest prime number; that is, there is an infinite number of primes. Some computer enthusiasts amuse themselves by writing programs to find the next larger prime, one that is greater than the last value published by a colleague. We will not deal in such large values, which are difficult, though not impossible, to store on the computer. We will instead attend to smaller prime numbers, and to determining whether a number is prime or not.

One way to find prime numbers in a certain range is the “sieve of Eratosthenes,” which can be an interesting computer exercise using arrays, and which we will reserve for one of your exercises at the end of the chapter. Another approach is simply testing a number for “primality,” and saving it in a reference array if it passes the test. Let us create a program, not overly complex, which will test all the integers from 2 to 100 to see if they are primes, and save the primes we find in a reference array, for possible later use.

We know that 2 is a prime, and that no other even numbers are prime. Thus, we could store 2 as the first entry in our reference array, without having to bother to test it, and then we could skip testing any other even numbers, since we know *a priori* that no other evens will make it. Then we can construct a loop to examine all of the odd numbers from 3 to 99 (or in some other range if you prefer), and see which ones are prime. Now, a number is prime if it has no divisors other than 1 and itself. Since we know that 1 divides into every integer, we will not test 1. We also know that we will not be testing any even numbers in our loop, so we do not have to try 2 (or any other even number) as a divisor, since no odd number is divisible by 2. We need to test possible

divisors of each odd number (except 3, which is clearly prime), beginning with 3, and testing odd divisors in sequence.

How many odd numbers must we test before we have determined whether a given integer is prime or not? Of course, if we do find one exact divisor, we do not have to test any more divisors—the number has failed the test. It turns out that, to determine if a number N is a prime, the largest divisor we have to test is the square root of N . Since divisors come in coordinate pairs (e.g., if 3 is a divisor of 15, its coordinate divisor is $15/3$, or 5), if N has a divisor greater than the square root of N , it had a coordinate divisor less than the square root, and we would have already found it out. Thus, we need, for each odd candidate prime number N , a loop to test all the possible odd divisors from 3 to the square root (truncated to integer) of N . This clearly means we do not have to test 3 or 5 or 7 for primeness, since their square roots are less than 3; we will just store them in our PRIME array and begin with 9.

We will thus begin by storing 2, 3, 5, and 7 in our PRIME array. The next prime we find should go into location 5 of the array, the next into location 6, and so on. Thus, we must introduce a counter into our program that will keep track of where to store the next prime we find. The counter will be incremented by 1 each time we find a new prime, and it will tell us where in the array to store it. The counter will start off at 4, since we have already stored 4 primes into the array. This seems to be sufficient initial analysis of the problem, and we should be ready now to write the program.

```
***** CREATE REFERENCE ARRAY OF PRIMES *****
***** WE BEGIN WITH 2, 3, 5, AND 7 AS KNOWN PRIMES *****
      INTEGER PRIME(60), N, NDIV, NP, NSQRT
      PRIME(1) = 2
      PRIME(2) = 3
      PRIME(3) = 5
      PRIME(4) = 7
** COUNTER NP WILL KEEP TRACK OF WHERE TO STORE ANY NEW PRIMES **
      NP = 4
      DO 50 N = 9, 99, 2
      NSQRT = N**0.5
***** NSQRT IS THE LARGEST DIVISOR WE HAVE TO TEST *****
      DO 40 NDIV = 3, NSQRT, 2
***** IF YOU FIND A DIVISOR, N IS NOT A PRIME *****
      IF( MOD( N, NDIV ) .EQ. 0) GO TO 50
      40    CONTINUE
***** IF NO DIVISORS WORKED, N IS A PRIME - STORE IT *****
      NP = NP + 1
      PRIME(NP) = N
      50    CONTINUE
```

The array of primes created in this program may have uses in later operations. One such operation is that of determining the prime factors of any number (included as an exercise at the end of this chapter).

If you wanted to print out the array of primes, presumably you would want to see more than one per line, so you might use:

```
PRINT 60
60 FORMAT('1', 20X, 'THE PRIMES BETWEEN 1 AND 100 ARE: /')
      PRINT 65, (PRIME(K), K = 1, NP)
65 FORMAT(5X, 15I4)
```

We must use the full implied list in the PRINT statement, because we will not have found 60 primes, and so we do not want to print out the whole array, just the entries we stored.

The program employs a loop, the one testing divisors, which has one entry but two possible exits (one if the number is a prime, one if it is not). This is a sensible way of looking at the problem; in programming it, a fairly structured approach has been used, even though it uses a GO TO. However, the approach might be criticized on the grounds that all program structures should have only one entry and only one exit. To accomplish this here, we would have to construct a WHILE or a REPEAT/UNTIL type loop with a complex termination condition.

This alternate formulation requires a bit of careful thought. We wish to test all of the possible odd divisors, in the range from 3 to the square root of the number being tested, unless one of them *does* divide N exactly. If it does, then we know the number is not a prime, and simply want to go on and test the next candidate prime. However, if none of the divisors work, then we know the number *is* prime, and we want to store it in the reference array PRIME. Thus, there are two possible terminal conditions for the divisor loop—that a divisor divides N exactly or that we have exhausted all of the divisors. But the action to be taken on leaving the loop depends on which way it was left. Our earlier formulation takes advantage of the two exits provided to take care of the different actions to be taken.

If we enforce the “one entry/one exit” rule, then we must somehow set a value in the loop that we can test when we leave the loop, in order to take the appropriate action. We will introduce a logical variable FLAG which will be set to .TRUE. before the loop begins, but set to .FALSE. in the loop if a number fails to be a prime. We can then test this variable upon leaving the loop to determine the appropriate action. We will implement parallel WHILE-type and REPEAT/UNTIL-type structures, for the sake of comparison with our original formulation.

```

INTEGER REM, NDIV, PRIME      INTEGER REM, NDIV, PRIME
NDIV = 3                      LOGICAL FLAG
REM = MOD( N, NDIV )          FLAG = .TRUE.
NSQRT = N**0.5                 NSQRT = N**0.5
***** WHILE *****
30 IF (REM.NE.0.AND.          ***** REPEAT/UNTIL *****
  & NDIV.LT.NSQRT) THEN        30 REM = MOD( N, NDIV )
    NDIV = NDIV + 2           NDIV = NDIV + 2
    REM = MOD ( N, NDIV )     IF(REM.EQ.0) FLAG = .FALSE.
    GO TO 30                  IF (FLAG.AND.NDIV.LE.NSQRT)
ENDIF                           & GO TO 30
IF(REM.EQ.0) THEN              IF (FLAG) THEN
  NP = NP + 1                 NP = NP + 1
  PRIME(NP) = N                PRIME(NP) = N
ENDIF                           ENDIF

```

Fortran 90 Additions to Array Handling

We have seen that a whole array, or a segment of an array, can be handled in one statement, without the need of a loop. Thus, if arrays A and B have been dimensioned the same, the instruction $A = B$ will store the entire contents of B, location by location, into A. Similarly,

$$A(3:10) = B(12:19)$$

will store the value of B(12) into A(3), the value of B(13) into A(4), and so on, through the value of B(19) into A(10).

The new control word WHERE will allow selective treatment of array elements that satisfy a certain condition (that is, *where* that condition occurs); for example:

```

REAL :: A(100)
READ*, A
WHERE (A.LE.50.0)
  A = A + 20.0
ELSEWHERE
  A = A - 5.0
ENDWHERE

```

The preceding code would examine each element of the filled array A, increasing any value in it which is less than 50.0 by 20.0, and decreasing all other values in the array by 5.0.

Other new system functions in Fortran 90 that can be used to manipulate arrays (such as DOTPRODUCT, SUM, MAXVAL, MINVAL, MAXLOC, MINLOC, and PRODUCT) are discussed in Appendix E.

USER-DEFINED TYPES

We mentioned earlier that FORTRAN only allows six basic data types, the various numeric types, logicals, and characters. Yet the programmer may occasionally want to make use of other kinds of ordered data types, such as the months of the year or the days of the week. There is no automatic means of defining your own data types, but with a little ingenuity you can manage well.

Let us say that you want to define two new “data types,” the months of the year and the days of the week. You would then like to be able to manipulate these data values in ways similar to those available for the other data types. However, it would not make sense to add them, or multiply them, or even to concatenate them. So what *would* you want to do with them? You might want to be able to output them, as labels or the like (incidentally, this capability is not available for most languages, such as Pascal, which *do* include the capability for user-defined types); we will see how to do this. The other thing you might want to do is to use them as limits to a loop, or as subscripts. In languages that allow user-defined types, this can be done directly, as:

DO MON = FEBRUARY, JUNE
or A(MONDAY) = ...

We will not be able to do this directly, but we can do the next best thing.

First of all, to set up our user-defined types, which would have to be defined in some way in any language, we will store their names in arrays:

```
CHARACTER*10 MONTH(12), DAY(7)
DATA MONTH/'JANUARY', 'FEBRUARY', 'MARCH', . . . /
DATA DAY/'SUNDAY', 'MONDAY', 'TUESDAY', . . . /
```

These assignments could also be made using a PARAMETER declaration, since they will not change. Because these are short arrays where we know the position of the various entries, we can establish the loop suggested earlier as:

```
CHARACTER MON*10
DO 5 N = 2, 6
    MON = MONTH(N)
    ...
5   ...
```

If we want to use a “user-defined type value” as an appropriate subscript, we can merely take advantage of the position we know these data entries occupy in the reference array, and refer to:

$A(2) = \dots$

since we know that ‘MONDAY’ is in position 2 of the reference array. If we had a much longer user-defined type, such as all of the elements of the Periodic Table, we could search for the appropriate entry in the reference array:

```
CHARACTER*15 TABLE(103)
DATA TABLE/'HYDROGEN','HELIUM','LITHIUM', . . . /
```

If we then wanted to use the appropriate position of, say, Silicon, as a subscript, we could search the reference TABLE to determine its position:

```
CHARACTER*15 FIND
FIND = 'SILICON'
K = 1
5 IF (FIND .NE. TABLE(K) ) THEN
    K = K + 1
    GO TO 5
ENDIF
NN = K
```

We could then use NN as the subscript to stand for Silicon. We have simplified the loop on the understanding that we are sure to find what we are looking for in the table. How might you modify the logic to take care of a case where the element you are looking for is somehow not found in the table?

Notice that all of the elements of your “user-defined type” are available for output (unlike those in languages that have built-in capabilities for user-defined types). If we wanted to print out the months defined in the loop, we could simply print out MON (or MONTH(N)). If we wanted to know the identity of the element represented by the subscript NN, we could output TABLE(NN). This gives us some flexibility in handling different data types.

Note: Fortran 90 provides the capability to construct “derived types” out of combinations of the existing types in the language. This still does not allow the creation of entirely new kinds of data objects, as we have in our MONTH example. These Fortran 90 derived types will be discussed in Chapter 13.

TWO-DIMENSIONAL ARRAYS

We have covered one-dimensional arrays, or lists or vectors. It is now time to look at another way of organizing data—the two-dimensional array, or matrix.

Uses

Often the information you are dealing with comes in *tabular* form, a two-dimensional arrangement in which both the rows and the columns have a special significance. For example, a page in a grade book is a two-dimensional array, where each row contains the set of grades belonging to a particular student, and each column represents the grades on a particular test or assignment.

	HW-1	HW-2	HW-3	CP-1	...	QUIZ #3	QUIZ #4
ADAMS, G.	90	98	65	88	...	80	87
BROWN, C.	88	97	76	75	...	75	78
...							
YATES, E.	96	88	80	92	...	95	93

You could store the numerical part of this table, without the labels for the rows and columns, in a two-dimensional array in FORTRAN. Just as you had to declare the size of a one-dimensional array in a DIMENSION or type statement at the beginning of the program, you must do the same for the two-dimensional array. If our grade book has 35 students (rows) and 20 assignments, the array declaration statement should be:

```
INTEGER GRADES(35, 20)
```

The number of rows comes first, followed by the number of columns. This statement will instruct the compiler to set aside 700 consecutive locations in memory under the name GRADES, and each element in the array will be represented by its *position*, with the row subscript first, the column second; for example,

GRADES(2,4) is the grade (75) in the second row, fourth column,

the grade for BROWN on CP-1. Each element of a two-dimensional array is identified by the array name, followed by the subscripts in parentheses, row first, then column:

array(row, column)

Subscripts can be any integer expression, but they should fall within the subscript bounds set in the dimension statement.

Notice that our idea of "user-defined types" could also be brought in here, if we wanted to use the appropriate actual labels for our rows and columns instead of just their integer representations. We could have two parallel arrays, one containing the labels for the rows (the NAMES of the students) and another containing the identification of the columns as course assignments (WORK). These could be established:

```
CHARACTER*12 NAMES(35), WORK(20)
DATA NAMES/ 'ADAMS, G.', . . . /
DATA WORK/ 'HW-1', . . . /
```

Then if a particular piece of information (such as the highest grade in the table) is identified as being in row 10, column 4, we can determine, and print out if we like, that this grade belonged to NAMES(10) [EINSTEIN, A.], and that it was on WORK(4) [CP-1]. This greatly enhances our abilities to manipulate arrays and maintain interesting labels for them.

Just as with one-dimensional arrays, the array size in one dimension can be a single number, indicating subscripts from 1 to that number, or it can be expressed in terms of a lower limit and an upper limit for the subscript in that dimension. Thus, you may have arrays dimensioned in any of the following ways:

```
INTEGER JAWS(40, 20), CUJO(-3:3,5), ALIEN(6,10:20)
REAL KILLER(-10:10, 5:20)
```

The array JAWS has 40 rows (numbered from 1 to 40) and 20 columns (numbered from 1 to 20); the array CUJO has 7 rows (numbered from -3 to 3) and 5 columns (numbered from 1 to 5); the array ALIEN has 6 rows (numbered from 1 to 6) and 11 columns (numbered from 10 to 20); and the array KILLER has 21 rows (numbered from -10 to 10) and 16 columns (numbered from 5 to 20). Usually, you will find a single number for the dimension adequate, but there are special cases—as we have already seen—when it is useful to be able to specify a range of subscripts to fit a problem that do not necessarily begin at 1, or are even positive.

Your choice of *data structure* (in this case, one-dimensional or two-dimensional array) is determined by the nature of the problem you are dealing with. The appropriate choice of data structure, just as the appropriate choice of algorithm, will help greatly in solving the problem facing you. Making the correct choice of data structure will be an ongoing concern as we proceed.

Nested Loops With Two-Dimensional Arrays

Since there are *two* dimensions in your structure, it will usually be appropriate to handle each dimension in a separate loop when dealing with the whole array. For example, assume we need to fill a two-dimensional “multiplication table” for the integers from 1 to 10, so that we have a table of the values from 1×1 to 10×10 . To do so, we set up a 10 by 10 array, set up an outer loop to access the rows, and an inner loop to access the columns. In this way, the array is filled in row-by-row order, going to all of the columns in a particular row before going on to the next row. A simple program segment to do this would be:

```
INTEGER MULT(10, 10)
DO 25 I = 1, 10
    DO 25 J = 1, 10
        MULT(I, J) = I*j
25 CONTINUE
```

We have used the integer variables I and J for subscripts, since this corresponds to the mathematical notation for two-dimensional arrays or matrices. In the multiplication table example, the order of filling the array did not matter; if we had made the column (J) loop the outer loop, the end result would have been the same. However, in some cases this is not true.

Imagine a problem where you are to fill a 5×5 array with the integers from 1 through 25, in order, across the rows, so that the first row contains 1, 2, 3, 4, 5, and so on. One way to do this would be the following:

```
INTEGER NUMBER(5, 5)
N = 1
DO 30 I = 1, 5
    DO 30 J = 1, 5
        NUMBER(I, J) = N
        N = N + 1
30 CONTINUE
```

In this case, the order of the loops *does* make a difference, since the counter N changes by 1 each time, and we want it to be stored across rows, not down columns. Of course there is another way to fill this array without using N:

```
NUMBER(I, J) = (I - 1)*5 + J
```

and if this formula were used, again the order of the loops would not matter, since the end result would be the same.

You can use loops to fill arrays, as we did in the previous examples, or to

manipulate data already stored in the arrays. To do this, we must first study the input and output of two-dimensional arrays.

Input/Output of Two-Dimensional Arrays

Again, we will make use of the implied list for I/O of two-dimensional arrays. However, since the programmer almost always thinks of a two-dimensional array a row at a time, usually reading from left to right as you would a book, we strongly recommend handling this I/O in a loop that specifically deals with one row at a time. In this way, you can deal with input that fills one row at a time with values, and you can output one row per line to your output device, which is the way you think of such an array's appearance. If you had to fill the two-dimensional array KAT, with 10 rows and 12 columns, with values stored or entered 12 per record, each representing one row, we would recommend the following:

```
INTEGER KAT(10, 12)
DO 20 I = 1, 10
    READ 15, (KAT(I, J), J = 1, 12)
15      FORMAT(12I5)
20  CONTINUE
```

Now it is true that the same information could be input without a loop, using doubly nested implied lists instead:

```
READ 15, ((KAT(I, J), J = 1, 12), I = 1, 10)
```

Notice that the *outer* parameter in the implied list is I, the *row* parameter, so that the array will be filled in row-by-row order. The compiler sets up a list beginning: KAT(1,1), KAT(1,2), KAT(1,3), and so on. The only problem with this formulation is that, in order to read the data entered properly, the repetition factor in the FORMAT must exactly match the number of *columns* in each row of the array. If the repetition factor is larger, say 16I5, the first example, the loop, will still work properly, but the second example, the doubly nested implied list, will read (or try to read) 16 values from each record, thus not entering the input values correctly. Of course, if the number of format descriptors in the FORMAT is less than the number of columns, both examples are in trouble.

Similarly, on output, you usually want to see one row of the two-dimensional array printed per line. The safest way to do this is in a DO loop on the rows. Thus if, after the array KAT was filled with values, and any manipulations performed, we could output it with the following:

```

DO 30 I = 1, 10
    PRINT 26, (KAT(I, J), J = 1, 12)
26      FORMAT('0', 12I6)
30  CONTINUE

```

In this case, only because the number of columns in the implied list is exactly the same as the repetition factor in the FORMAT statement, we could have also written:

```
PRINT 26, ((KAT(I, J), J = 1, 12), I = 1, 10)
```

and the result would have been the same.

By now, you are probably remembering the shorthand we used when we wanted to do I/O on an entire one-dimensional array, by just naming the array, and wondering whether we can do the same thing here. We can, but the results will be unexpected. You can name the two-dimensional array in an I/O or DATA statement without using subscripts, and the compiler will give you the entire array, *but in the order in which it is stored in memory*. Unfortunately for this purpose, the two-dimensional arrays in FORTRAN are stored in column-by-column order. Memory is like one big linear list, so the two-dimensional array must be mapped onto a one-dimensional storage medium. A decision has to be made by compiler writers whether to do this in row-by-row order (or “row-major” order), as many languages do, or in column-by-column (or “column-major”) order, as FORTRAN designers chose to do.

Thus, if you simply write:

```

DIMENSION KAT(10,12)
PRINT 26, KAT

```

the effect is the same as if you had written:

```
PRINT 26, (( KAT(I, J), I = 1, 10), J = 1, 12)
```

making the *column* parameter the outer control, and thus creating a list of locations one column at a time: KAT(1,1), KAT(2,1), KAT(3,1), and so on. Although occasionally this may be what you want, in most instances you want to deal with the array a row at a time, and so you must use fully-detailed implied lists.

Recall the array NUMBER (5 x 5) that we filled with the integers 1 through 25. We indicate in the following diagrams the way *we* think of this array being stored (and we can generally manipulate it under that assumption) and, on the right, the way it is *actually* stored by the computer.

NUMBER					NUMBER	
	1	2	3	4	5	(1,1)
1	1	2	3	4	5	(2,1)
2	6	7	8	9	10	(3,1)
3	11	12	13	14	15	(4,1)
4	16	17	18	19	20	(5,1)
5	21	22	23	24	25	(1,2)
						(2,2)
						(3,2)
					
						(4,5)
						(5,5)

Matrix Manipulations

Since you probably have studied some scientific discipline, you may be familiar with the use of vectors and matrices to represent various problems you have to deal with. Some of these may involve multiplying a matrix by a scalar, or adding or subtracting two matrices of the same dimensions (we will leave these to exercises for you to try). A more interesting problem is that of *matrix multiplication*. In such a case, a matrix of size $m \times n$ is multiplied by one of size $n \times nn$, according to the following rule, if C is the product matrix:

$$C_{i,j} = \sum_{k=1}^N A_{i,k} \times B_{k,j}$$

Of course, these matrices have to be *conformable*; that is, the number of columns of the first matrix must be the same as the number of rows of the second matrix. Thus, if we had an array A of size 4×5 , and an array B of size 5×6 , both already filled with values, we could create a matrix product of A \times B in array C (4×6) in the following way:

```

PARAMETER (M = 4, N = 5, NN = 6)
REAL A(M,N), B(N,NN), C(M,NN), SUM
..... {input values to A and B}
DO 50 I = 1, M
    DO 50 J = 1, NN
        SUM = 0.
        DO 40 K = 1, N
            SUM = SUM + A(I,K)*B(K,J)
40          CONTINUE
        C(I, J) = SUM
50          CONTINUE

```

We will discuss matrix manipulations further at a later point in the text.



ARRAYS OF HIGHER DIMENSION (UP TO 7)

Earlier versions of FORTRAN only handled arrays of up to three dimensions, but FORTRAN 77 will accommodate arrays of up to 7 dimensions. Each of these dimensions is declared as a single number or a range, and then the array is subscripted accordingly. Thus you could have:

```
DIMENSION A(5,6,7), B(-5:5,-2:7,8,6), C(2,3,4,5,5,6:12)
```

We will look into uses of three-dimensional arrays later in the book. Arrays of higher dimension are used rarely, but they might have applications in more complex problems. If you view a two-dimensional array as analogous to a page in a book, then a three-dimensional array could be the book itself. Another dimension could be introduced by having many books, then these books might be in different libraries, further organized by city.



ARRAY MANIPULATIONS

Finding Maxima and Minima

There are a number of common processes that you will perform over and over in dealing with arrays; for this reason, it is good to get an idea of how to carry out these operations early in your experience with arrays. Very often you will need to find the largest, or the smallest, value in an array. We have already done similar problems to find the largest or smallest value in a sequence of values that are read in. To adopt a technique that will work on arrays of any

size, begin with the first element of the array, and consider that to be the largest value you have seen so far; save it in a location for keeping track of the largest value seen. Then test every other value in the array against the one saved. If a value tested is larger than the one stored, put the new value in that location. By the time you have finished looking at the entire array, the value saved will be the largest in the whole array. (An analogous operation can be performed to find the smallest value in an array.)

```
***** FINDING LARGEST VALUE IN A ONE-DIMENSIONAL ARRAY *****
REAL A(200)
READ*, A
***** USE LOCATION BIG TO KEEP TRACK OF LARGEST VALUE *****
BIG = A(1)
DO 20 N = 2, 200
    IF(A(N).GT.BIG) BIG = A(N)
20 CONTINUE
PRINT*, BIG, ' IS THE LARGEST VALUE'
```

You can also do the same job on a two-dimensional array.

```
***** FINDING LARGEST VALUE IN A TWO-DIMENSIONAL ARRAY *****
REAL B(20, 30)
...{input values to B}
BIG = B(1,1)
DO 40 I = 1, 20
    DO 40 J = 1, 30
        IF (B(I,J).GT.BIG) BIG = B(I,J)
40 CONTINUE
PRINT*, BIG, ' IS THE LARGEST VALUE'
```

Note that we were able to begin the loop at 2 in the case of the one-dimensional array, since we had already examined the first element in the list, A(1). However, in the case of the two-dimensional array, although we had already examined the first element in the array (B(1,1)), we cannot begin the I and J loops at 2, since that would ignore the rest of row 1 and column 1. Thus, we begin these loops at 1, and tolerate looking at B(1,1) twice, as a small penalty to pay for correctness.

Searching (Linear)

Finding a maximum or a minimum is a kind of search. There are other searches you can perform on arrays, to find whether a particular value occurs in the array, or how many times values in a certain range occur in the array, and so on.

To determine whether a particular value is in an array (for example, whether a license plate number is on the list of stolen cars), we begin our search at the first entry in the array, and continue it until either the value is

found or the search is exhausted, whichever comes first. Our first thought is to write a program that exits prematurely from the loop if the value is found, and takes a normal exit if it is not found.

```
***** IS INPUT VALUE IN THE ARRAY? *****
DIMENSION KAT(300)
READ*, KAT
PRINT*, INPUT THE VALUE YOU WANT TO FIND'
READ*, LOOK
DO 60 I = 1, 300
    IF (KAT(I) .EQ. LOOK) GO TO 75
60 CONTINUE
PRINT*, LOOK, ' WAS NOT FOUND IN THE ARRAY'
STOP
75 PRINT*, 'THE VALUE', LOOK, ' WAS FOUND IN LOCATION', I
STOP
END
```

We have already discussed the preferability, from a structured point of view, of having loop structures with one entry and one exit. Because our search program violates that principle, we will create an alternate version of the program that has a more structured form, by making use of a logical flag value and using a While structure that repeats the loop while the value has not been found and the list has not been exhausted.

```
***** LINEAR SEARCH FOR VALUE IN ARRAY *****
INTEGER KAT(300), LOOK
LOGICAL FOUND
FOUND = .FALSE.
READ *, KAT
PRINT*, 'INPUT THE VALUE YOU WANT TO FIND'
READ*, LOOK
I = 1
30 IF (.NOT.FOUND .AND. I.LE.300) THEN
    IF (KAT(I) .EQ. LOOK) FOUND = .TRUE.
    I = I + 1
    GO TO 30
ENDIF
IF (FOUND) THEN
    PRINT*, 'THE VALUE', LOOK, ' WAS FOUND IN POSITION ', I-1
ELSE
    PRINT*, 'THE VALUE ', LOOK, ' WAS NOT FOUND'
ENDIF
STOP
END
```

A similar problem is that of counting how many times a value occurs in a list. In this case, we must look through the entire array, since we want to find all occurrences of the value. Let us adapt this search technique to a two-dimensional array.

```

INTEGER NICK(30, 50), COUNT
... {input values to NICK}
PRINT*, 'ENTER THE VALUE YOU WANT TO COUNT'
READ*, MINE
COUNT = 0
DO 80 I = 1, 30
    DO 80 J = 1, 50
        IF (NICK(I,J).EQ.MINE) COUNT = COUNT + 1
80 CONTINUE
PRINT*, 'THERE ARE',COUNT,' OCCURRENCES OF THE VALUE',MINE
STOP
END

```

Arrays Used for Counters

A variation on the previous problem is to count how many values fall in different *ranges* in a set of data. Let us imagine that we have a list of experimental values that we know lie in the range from 1 to 100, and we wish to determine how many values fell between 1 and 5, how many between 6 and 10, and so on. To do this, we need 20 different counters, one for each range. Clearly, the only sensible way to do this is to use an array of counters, where the first one is for the range from 1 to 5 (that is, $1.0 \leq V \leq 5.0$), the second for the range from 5 to 10 (that is, $5.0 < V \leq 10.0$), and so on. This choice of the array data structure will allow us to initialize the counters easily, and we can even find a clever way to know which counter to increment without having to perform twenty IF tests.

Examine the relationships we want to establish between values and counters:

<u>Value</u>	<u>Counter</u>
1.0 – 5.0	1
5.0 – 10.0	2
10.0 – 15.0	3
.....	..
95.0 – 100.0	20

It is clear that there is a mathematical manipulation we can perform on the data value V to transform it into the appropriate position of the counter we want to increment, K :

$$K = \text{INT}(V - 1.0)/5 + 1$$

where INT takes the integer part of the value given to it.

Thus, a counter program could be written to examine 1000 points of experimental data, which have already been read into an array called VALUE, and count how many of them fall into each of our 20 classifications.

```
***** COUNT RANGES OF EXPERIMENTAL DATA *****
***** WHERE VALUES ARE KNOWN TO LIE FROM 1.0 - 100. *****
      REAL VALUE(1000), LOW, HIGH
      INTEGER COUNT(20), K
      DATA COUNT /20*0/
      OPEN (3, FILE = 'ASTRO', STATUS = 'OLD')
      READ(3,*) VALUE
      DO 50 I = 1, 1000
         K = INT ( VALUE(I) - 1.0 )/5 + 1
         COUNT(K) = COUNT(K) + 1
50   CONTINUE
      PRINT 55
55   FORMAT('1', 20X, 'COUNTS OF DATA DISTRIBUTION'//5X,
& 'VALUE RANGE', 3X, 'COUNT'//)
      LOW = 1.0
      DO 70 J = 1, 20
         HIGH = LOW + 4.0
         PRINT 65, LOW, HIGH, COUNT(J)
65   FORMAT(4X, F4.1, ' - ', F5.1, 3X, I5)
         LOW = LOW + 5.0
70   CONTINUE
      STOP
      END
```

A Look at FORTRAN 90 Features for Two-Dimensional Arrays

Many-dimensional arrays may also be declared, several at one time, to have the same dimensions, by using the :: notation, and they may be given initial values in such a statement. The subscript triplets we discussed under one-dimensional arrays can also be used for higher-dimensional arrays; along any dimension, a lower bound, an upper bound, and a stride may optionally be specified to select out a range of array locations. Entire arrays of the same dimensions may be combined or compared just by using the array names; for example,

```
INTEGER, DIMENSION (50:100) :: KIT, KAT, KAKE
READ*, KIT, KAT
KAKE = KIT - KAT
```

The WHERE structure may also be used on higher-dimensional arrays; for example, the previous example could be revised to:

```

WHERE (KIT .GT. 100)
    KAKE = KIT - KAT
ELSEWHERE
    KAKE = KIT + KAT
ENDWHERE

```

As with the one-dimensional arrays, there are special new system functions that will take the product or sum of terms of an array, or find the maximum or minimum value or its location (along one dimension, if requested). There also is a new function to multiply together two conformable two-dimensional arrays, and one to take the TRANSPOSE of a matrix. See Appendix E for discussion of these new features under Fortran 90.



SORTING

Probably more computer time is spent doing sorting than any other single operation. Numbers must be sorted into ascending or descending order, lists of names must be sorted into alphabetical order, and so on. There are many techniques, some very complicated and tricky, for sorting arrays of data. There are tradeoffs in time or space utilization efficiency in choosing different sorting procedures. We will present two sorts here that are conceptually easy to understand and implement, even though they are not very efficient (that is, they take a greater time to sort long lists than alternate procedures take). At the end of the chapter, we will suggest references that will introduce you to more complex and efficient sorting methods.

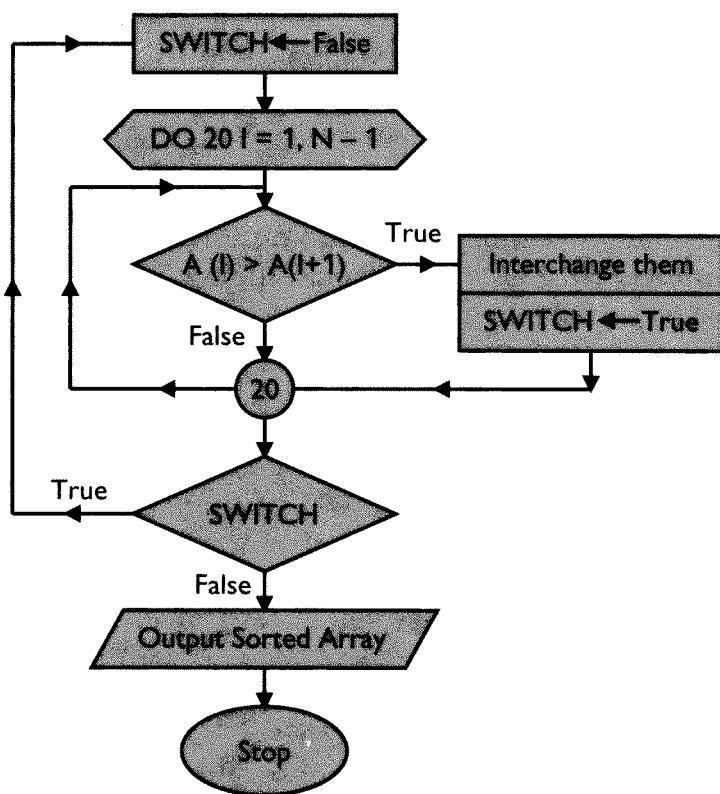
A simple sort to understand is the “bubble sort,” which rearranges an array by comparing adjacent values in the array. If the pair of values is in the proper order, nothing is done; if they are out of order, they are interchanged, and a note is made that an interchange has taken place. After a pass down the list is completed, if any changes have been made, the process is repeated. This procedure continues until a pass can be made down the list when no changes are made. At this point, the array is in sorted order. Examine the following example, in which we want to sort the values into *ascending* order.

(7)	5	5	(5)	4
5	7	7	4	5
(8)	(6)	(7)	(4)	6
6	4	4	6	6
4	8	8	8	7
				7
				8

Notice how the smallest value, 4, “bubbled up” to the top of the list. If a list is already close to sorted order, this procedure will not take very long, but it can be very time-consuming to use on long lists that are badly out of order. However, it is easy to write a FORTRAN program to implement this sort procedure. That is a great advantage, and for most short lists you will deal with, it is adequate.

Bubble Sort

A flowchart will illustrate the algorithm we discussed for accomplishing this sorting procedure for a list of length N:



We will now present a simple FORTRAN program that performs a bubble sort, as it is described in the previous section and in the flowchart. We will use a logical variable SWITCH to indicate whether any changes are made in a pass down the list, and then test it at the end of the pass to see if we need to repeat the procedure. If there are no changes made, the list is sorted.

```
***** BUBBLE SORT *****
***** SORTING IN ASCENDING ORDER *****
***** A REAL ARRAY OF SOME SPECIFIED LENGTH N *****
```

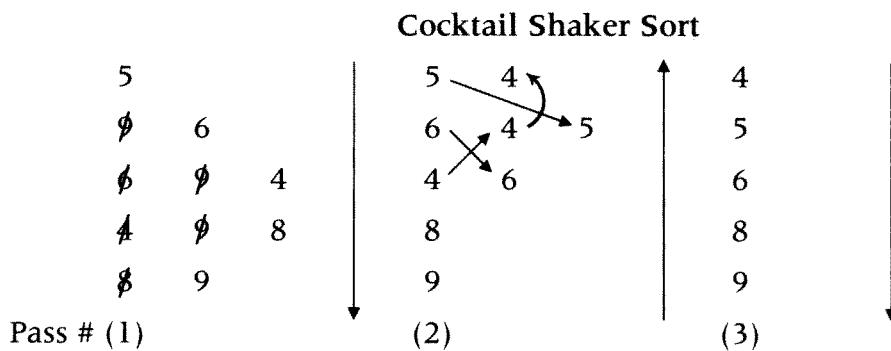
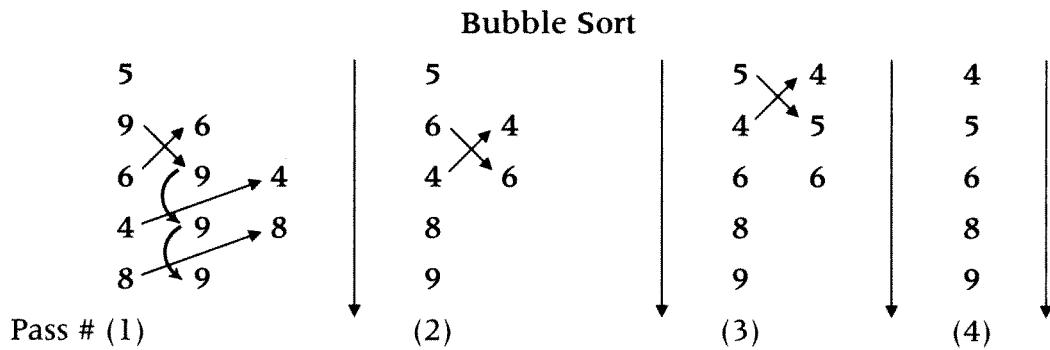
```

REAL A(1000)
LOGICAL SWITCH
PRINT*, 'ENTER THE NUMBER OF ELEMENTS IN YOUR LIST'
READ*, N
PRINT*, 'NOW ENTER THE VALUES, SEPARATED BY COMMAS'
READ*, (A(I), I = 1, N)
***** SET UP A PASS DOWN THE LIST *****
***** REPEAT UNTIL NO CHANGES ARE MADE IN LIST *****
10 CONTINUE
    SWITCH = .FALSE.
    DO 20 I = 1, N - 1
***** IF THE PAIR OF VALUES IS NOT IN PROPER ORDER *****
        IF (A(I) .GT. A(I+1)) THEN
***** SWITCH THE VALUES AND NOTE A SWITCH WAS MADE *****
            TEMP = A(I)
            A(I) = A(I+1)
            A(I+1) = TEMP
            SWITCH = .TRUE.
        ENDIF
20     CONTINUE
***** DO WE NEED TO PERFORM ANOTHER PASS? *****
    IF (SWITCH) GO TO 10
    PRINT 66
66    FORMAT('1', 40X, THE SORTED LIST IS:'//)
    PRINT 77, (A(I), I = 1, N)
77    FORMAT('0',12F6.1)
    STOP
END

```

The bubble sort is easy to conceptualize and turn into appropriate FORTRAN code. However, it is a very inefficient sorting procedure. There are various ways in which you can try to improve the sort itself. For example, one could notice that, on each pass, the largest value *sinks* to the bottom of the list; as a result, we should not have to examine it again. Thus, you might shorten the length of the sublist to which you are applying the bubble sort on each pass, so that on the first pass you examine through location N, but on the next one only through N-1, and so on. This would shorten execution time somewhat; however, on the accepted definition of *efficiency* of an algorithm, it still has the same poor order of efficiency, proportional to N^2 . You might also notice that, if no interchanges take place below a certain point in the array on a pass, they will not take place below that point in future passes. Thus, if you used a variable to keep track of the position of the last interchange made on a pass, on the next pass you could use that as the lower limit of pairs you would compare to test for proper order.

One other improvement to the bubble sort has been suggested. If you reverse the order in which the list is scanned on alternate passes, it may cut down on the number of passes needed, because on one pass the largest value sinks to the bottom and on the next pass the smallest value rises to the top. This is sometimes called the “cocktail shaker” sort, for obvious reasons. The code for this is a bit trickier, but you might want to try writing it for practice in developing your skills. Using a short list to sort into ascending order, we can demonstrate that the cocktail shaker sort cuts down the number of passes needed to sort that list.



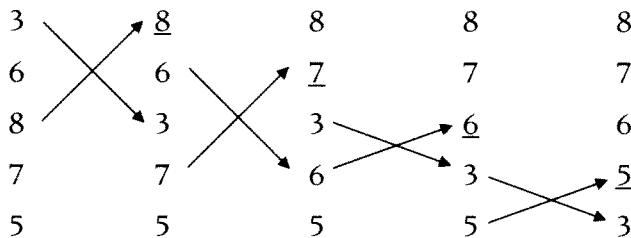
The bubble sort may not be very efficient in sorting long lists, but, if the data is in close-to-correct order, it can proceed very fast. Many sorts, such as the next one, always take the same amount of time to execute, even if the data is in perfect order to start with. At least, the bubble sort procedure takes advantage of any inherent order in the data.

Shuttle/Interchange, or Selection, Sort

The *selection sort* is also simple to conceptualize and write, since it is based on an obvious approach to sorting that you might use if doing it by hand. If you want to sort a list into order—let us make it descending order this time for a change—then you would want the largest value on top, the second largest value in position 2, and so on. One way to do this is to search the entire list for

the largest value, and then put it on the top of the list, taking the value that was in that position and placing it where the largest value used to be. Once this is done, the list is sorted one element deep, so you do not need to pay any more attention to the value in the first position. Next, you begin at position 2, looking for the largest value in the sublist from there to the end of the list; when you find it, you interchange it with the value in position 2. And so on. This is a very logical approach, and it works, though it is not fast.

The following is an example of a short list on which this procedure is applied:



We have already learned a technique for finding the largest (or the smallest) value in an array, and we can apply it in developing our sorting procedure. Notice that each time we search a sublist for its largest value, we begin one location below the initial position for the last pass. Thus, the sublists get shorter and shorter, until finally we have exhausted all of them, and the list is in proper descending order.

```

***** SELECTION SORT *****
***** INTO DESCENDING ORDER *****
***** FOR ANY REAL ARRAY OF SPECIFIED LENGTH *****

REAL B(1000), BIG
INTEGER NN, WHERE
PRINT*, 'ENTER HOW MANY VALUES YOU WISH TO SORT'
READ*, N
PRINT*, 'ENTER YOUR VALUES, SEPARATED BY BLANKS OR COMMAS'
READ*, (B(J), J = 1, N)
NN = N - 1
***** LOOP TO EXAMINE ALL SUBLISTS FOR LARGEST VALUE *****
DO 50 I = 1, NN
***** FIND LARGEST VALUE IN SUBLIST *****
    BIG = B(I)
    WHERE = I
    DO 40 J = I+1, N
        IF (B(J) .GT. BIG) THEN
            BIG = B(J)
            WHERE = J
        ENDIF
    ENDIF
END
  
```

```

40    CONTINUE
***** INTERCHANGE LARGEST VALUE WITH TOP OF SUBLIST *****
    B(WHERE) = B(I)
    B(I) = BIG
50    CONTINUE
***** LIST IS COMPLETELY SORTED *****
    PRINT 77
77    FORMAT('1', 40X, 'THE SORTED LIST IS:'//)
    PRINT 88, (B(J), J = 1, N)
88    FORMAT('0',12F6.1)
    STOP
    END

```

Efficiency Considerations

Neither the bubble sort nor the selection sort is the most efficient sort available, but they are the easiest to understand and to program. If—for most of your applications—you are not sorting very long lists, they will serve you well. The length of time a sorting procedure takes is always some function of the length of the list to be sorted, N . Unfortunately, the two procedures you have learned generally operate in a time proportional to N^2 , which means that as you go to very long lists, the times required to sort them can become astronomical.

Very efficient sorting procedures such as the Shell Sort, the Quicksort, the Heapsort, and so on, are more difficult to understand and program, and the latter two are best implemented using recursion (which is not standardly available in FORTRAN). We will not take the time to cover them here, since we have many other topics to cover that are more important to most scientific applications. However, if you are interested in pursuing the matter of efficient sorts further, say at a time when you are faced with 100,000 or more values to sort, you are referred to the references at the end of this chapter. For most applications, you probably will not need these techniques, but the literature is available, and your programming skills should be adequate to the task. If you are very fortunate, your computer system may have a built-in efficient sort subroutine that you can use. If there is one, check it out, and read the literature on it to see which of the algorithms it uses.



BINARY SEARCH

If an array has already been put into sorted order (which we have just learned how to do), a much more efficient method of finding a value in the array is that of the *binary search* procedure (the binary search has an efficiency on the order

of $\log_2 N$, whereas the linear search has an efficiency on the order of N). In this procedure, we simply keep cutting the part of the array we look in by half, until either we have found the value or the list is exhausted. Let us assume our list has been sorted into ascending order (the more common case). We look at a value in the middle (roughly) of the list; if it is the value we are looking for, great, and we can stop. If not, we test whether it is greater than or less than the value we are looking for, and this determines which half (upper or lower) to look in next. We repeat this procedure until the value is found or the list is exhausted, whichever comes first.

We will establish two pointers, LOW and TOP, which indicate the first and last locations of the section of the list we are currently examining, and the average of these two will determine the "middle" location we look at. We will employ a logical flag value to indicate if the value has been found, and we will use a While loop of the type we have been employing for similar problems. Since these techniques and the While structure are now familiar, a flowchart is probably not needed, and instead we will comment generously in the program itself.

```
***** BINARY SEARCH PROCEDURE *****
***** FOR LIST OF LENGTH N, SORTED INTO ASCENDING ORDER *****
      INTEGER KAT(1000), N, LOOK, LOW, TOP
      LOGICAL FOUND
      DATA FOUND /.FALSE./
      OPEN (4, FILE = 'KITTEH', STATUS = 'OLD')
      READ (4, *) N
      READ (4, *) (KAT(I), I = 1, N)
      PRINT*, 'ENTER THE VALUE YOU WANT TO FIND'
      READ*, LOOK
***** INITIALIZE POINTER LOW TO FIRST POSITION OF LIST - 1 *****
***** AND POINTER TOP TO LAST POSITION IN LIST - N *****
      LOW = 1
      TOP = N
***** WHILE *****
      20 IF (.NOT.FOUND .AND. LOW.LE.TOP) THEN
***** CALCULATE MIDDLE POSITION OF LIST *****
          MID = (TOP + LOW + 1)/2
          IF (LOOK .EQ. KAT(MID) ) THEN
              FOUND = .TRUE.
          ELSEIF (LOOK .LT. KAT(MID) ) THEN
***** SEARCH IN UPPER HALF *****
              TOP = MID - 1
          ELSE
***** SEARCH IN LOWER HALF *****
              LOW = MID + 1
          END IF
      END IF
  END DO
END
```

RATS					BATS				
	1	2	3	4		1	2	3	4
1	5	5	5	5	1	7	7	2	2
2	5	5	3	3	2	7	7	2	2
3	3	3	3	3	3	7	7	2	2
					4	7	7	2	2



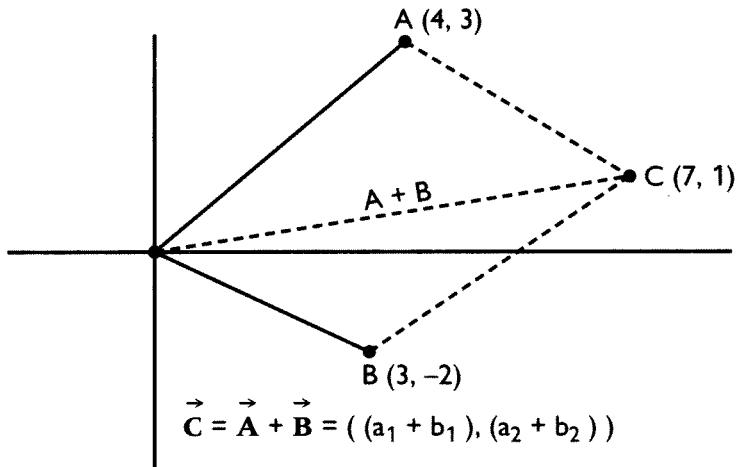
VECTOR MANIPULATION (AN OPTIONAL SECTION)

Vectors can be compared coordinate by coordinate, or added and subtracted, their lengths can be calculated, they can be rotated and/or displaced, or one can take the “dot product” or the “vector product” of two vectors. Our jet engine waveform example would probably proceed by comparing the coordinates of the vectors describing different waveforms, perhaps viewing them as defining different points in an n -dimensional space, and then trying to find a best-separating “surface” (of $n-1$ dimensions) that separates the good jet engine patterns from the bad. This is all somewhat complex, but has a simplified two-dimensional analog where you simply find the equation of the line that best separates the two sets of points in the plane. The various other operations that can be performed on vectors are worthwhile to discuss here, since they have many applications in scientific problems.

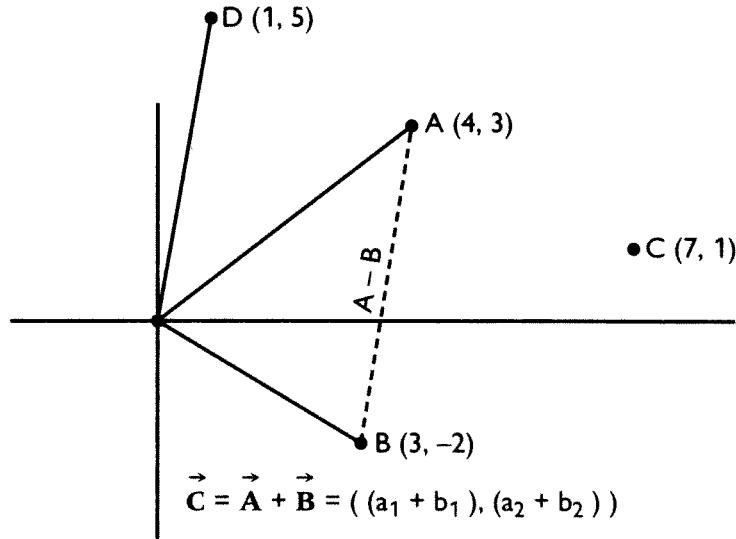
The *addition* of two vectors gives the *resultant* vector, describing the effect if they are both applied (for example, the resultant force of two directed forces). As a simple case, imagine two vectors representing points in a two-dimensional space, \vec{A} and \vec{B} . Each of these can be viewed as a *displacement* of position in the space, and the sum of the two as the resulting displacement if one change (\vec{A}) is followed by the second (\vec{B}). We can see this graphically as shown on the next page.

In our example, this is $\vec{C} = ((4 + 3), (3 + (-2))) = (7, 1)$. Notice in the parallelogram representing the displacements that the upper right edge (from \vec{A} to \vec{C}) is a translated version of the vector \vec{B} , and so we are actually determining the result of displacement \vec{A} followed by displacement \vec{B} . This operation generalizes easily to the addition of two n -dimensional vectors:

$$\vec{A} + \vec{B} = ((a_1 + b_1), (a_2 + b_2), \dots, (a_n + b_n))$$



representing successive translations in n -space. One can, of course, add more than two vectors, representing a succession of translations in the space. Further, if two vectors represent two *forces* applied to a body, the sum of the two is the resultant force on that body, and the analysis of the problem can proceed as if there were one single force (the resultant, $\mathbf{A} + \mathbf{B}$) acting on the body in question.



The *subtraction* of two vectors is shown on the preceding diagram. It is the other diagonal of the parallelogram, and its physical importance is that it represents a translation represented by $\vec{\mathbf{A}}$ followed by a translation of $-\vec{\mathbf{B}}$. In our example, this is shown by point \mathbf{D} , a translation of the “other diagonal” $\vec{\mathbf{A}} - \vec{\mathbf{B}}$ to the origin.

$$\vec{\mathbf{D}} = \vec{\mathbf{A}} - \vec{\mathbf{B}} = ((a_1 - b_1), (a_2 - b_2))$$

In our case,

$$\vec{\mathbf{D}} = ((4 - 3), (3 - (-2))) = (1, 5).$$

Subtraction can readily be generalized to n-dimensional vectors:

$$\vec{A} - \vec{B} = ((a_1 - b_1), (a_2 - b_2), \dots, (a_n - b_n))$$

The *length* (or *magnitude*) of a vector can be calculated, by an extension of the Pythagorean theorem, as:

$$|A| = \text{length of } \vec{A} = \sqrt{A_1^2 + A_2^2 + \dots + A_n^2}$$

For example, our vector \vec{A} in the diagrammed example has a length:

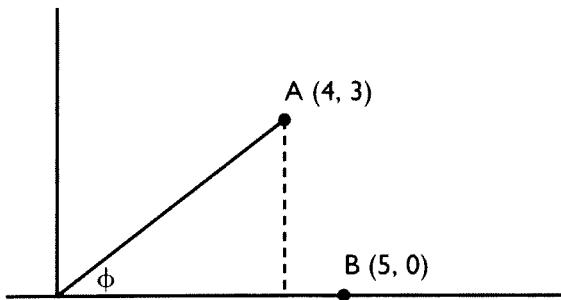
$$|A| = \sqrt{4^2 + 3^2} = \sqrt{25} = 5$$

The *dot product* (or scalar product, or inner product) of two vectors is the product of their magnitudes times the cosine of the angle between them. It is positive if the angle is less than 90° , negative if the angle is greater than 90° , and zero if the angle is equal to 90° (that is, if the two vectors are perpendicular to one another). Thus, the dot product

$$\vec{A} \cdot \vec{B} = |A||B| \cos\phi \quad (\text{where } \phi \text{ is the angle between } \vec{A} \text{ and } \vec{B})$$

If we look at the following simple two-dimensional example of two vectors \vec{A} and \vec{B} , we see that the dot product is the product of the magnitude of \vec{B} times the *orthogonal projection* of \vec{A} onto \vec{B} . In our example, \vec{A} is $(4, 3)$, \vec{B} is $(5, 0)$, and $\vec{A} \cos \phi$ is just the x-projection, or x-coordinate, of \vec{A} . Thus we get

$$\vec{A} \cdot \vec{B} = B (A \cos \phi) = 5 (4) = 20$$



We can also see that the dot product can be represented as:

$$\vec{A} \cdot \vec{B} = A_1 B_1 + A_2 B_2$$

In our example, this calculates to:

$$\vec{A} \cdot \vec{B} = 4(5) + 3(0) = 20$$

The dot product can be generalized to two n -dimensional vectors:

$$\vec{A} \cdot \vec{B} = A_1 B_1 + A_2 B_2 + \dots + A_n B_n$$

The dot product is used in mechanics, to calculate the *work* accomplished by a constant force F (which has a magnitude and a direction, and so can be

represented as a vector) which undergoes a vector displacement \vec{D} :

$$\text{Work} = \vec{F} \cdot \vec{D}$$

Similarly, the work done (in watt-seconds or joules) when electricity moves from one point at a certain potential to another point at a different potential, is the dot product of the force (or the field \vec{E} times the charge q on a positively charged particle exploring the field) and the distance through which it moves.

In FORTRAN, the *dot product* of two vectors in n -space would be calculated by accumulating the (scalar) sum of the products of their respective coordinates. For example, the dot product of two vectors, each with 10 "coordinates," would be calculated:

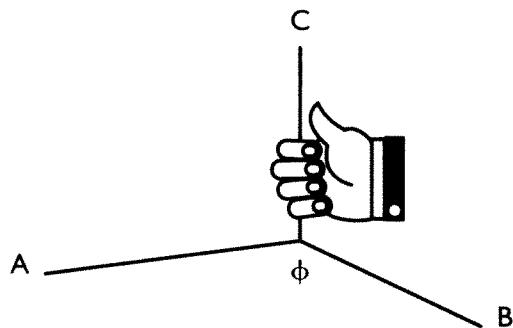
```
REAL A(10), B(10), DOT
      {A and B are filled with coordinate values}
DOT = 0
DO 30 I = 1, 10
      DOT = DOT + A(I)*B(I)
30  CONTINUE
PRINT*, DOT
```

The *vector product*, or *cross product*, of two vectors \vec{A} and \vec{B} is defined as:

$$\vec{C} = \vec{A} \times \vec{B}; \text{ magnitude of } C = AB \sin \phi$$

direction of C is perpendicular to the plane of \vec{A} and \vec{B}

That is, the vector product of two vectors \vec{A} and \vec{B} has a magnitude equal to the product of the magnitudes of A and B times the sine of the angle (ϕ) between them, and it has a direction perpendicular to the plane in which \vec{A} and \vec{B} lie. Since there are two possible perpendiculars to such a plane, the correct one is determined by the "right-hand rule": if you take your right hand and curl it in the direction from \vec{A} to \vec{B} , your thumb will point in the direction of \vec{C} .



If \vec{A} and \vec{B} are vectors in three-dimensional space, and we represent a unit vector in the x-axis direction as \vec{i} , one in the y-direction as \vec{j} , and one in the

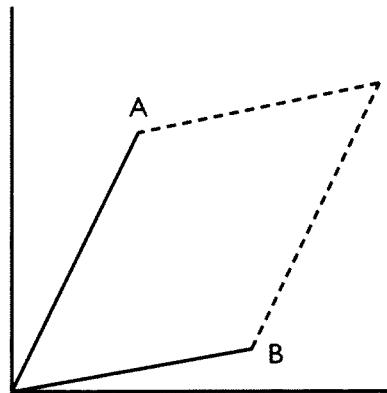
z-direction as \vec{k} , the cross product can be written as:

$$\vec{A} \times \vec{B} = (A_y B_z - A_z B_y) \vec{i} + (A_z B_x - A_x B_z) \vec{j} + (A_x B_y - A_y B_x) \vec{k}$$

where A_x is what we have previously been referring to as A_1 , A_y is A_2 , and A_z is A_3 , and so on. In FORTRAN, it is simply a one-dimensional array with three entries; the first entry is $(A_y B_z - A_z B_y)$, and so on. This can also be represented in a matrix determinant form (to be discussed) as:

$$\vec{A} \times \vec{B} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix}$$

The cross product of two vectors $\vec{A} \times \vec{B}$ is the magnitude of \vec{A} times the component of \vec{B} perpendicular to \vec{A} (or vice versa), and its magnitude is the area of the parallelogram formed by \vec{A} and \vec{B} .



The *angular momentum* (a quantity which is conserved) of a particle moving under a constant force (one in uniform linear motion or in uniform circular motion) is the cross product of the vector defining its position (\vec{r}) and the vector defining its momentum (\vec{p}):

$$\vec{L} = \vec{r} \times \vec{p}$$

This is Kepler's Second Law of planetary orbits. The *torque*, or *moment*, of a force \vec{F} acting on a particle whose position is described by the vector \vec{r} is the product of these two vectors:

$$\overrightarrow{\text{TORQUE}} = \vec{r} \times \vec{F}$$

If the coordinate system (axes) of a vector \vec{A} are *rotated* by an angle θ with respect to the original coordinate system in which it was defined, its new coordinates are given for \vec{A}' :

$$\begin{aligned} \text{If } \vec{A} &= (A_x, A_y), \text{ then} \\ \vec{A}' &= ((A_x \cos \theta + A_y \sin \theta), (-A_x \sin \theta + A_y \cos \theta)) \end{aligned}$$



FORTRAN 90 FEATURES

Fortran 90 provides the capability of declaring several arrays of the same dimensionality in one statement, using ::

```
REAL, DIMENSION (10, -5:5) :: A, B, C
```

and values can be initialized in the arrays using such a type and dimension declaration statement; for example,

```
INTEGER, DIMENSION (5) :: X = (/ 3, 4, 3, 4, 3 /)
```

Segments of an array can be specified along a dimension by a *subscript triplet*—([lower bound] : [upperbound] : [stride])—where if the lower bound designation is omitted, the lowest subscript along that dimension is assumed, and if the upper bound is omitted, the maximum value of the array subscript along that dimension is assumed. The stride indicates a step size between lower bound and upper bound; if it is omitted, it is assumed to be 1. Thus a single reference, such as A(15:25:3), can refer to a whole subset of elements in an array, for purposes of assignment or input/output. Further, the array name with no subscripts is then taken to refer to the entire array, allowing such statements as A = 2*A for an array A.

The WHERE construct allows an action to be taken with regard to parts of an array *where* a specified condition holds (and optionally an alternate action to be taken elsewhere).

Special functions have been developed in Fortran 90 for array manipulation; see the discussion in the chapter and Appendix E. “Derived types” may be set up in Fortran 90 as combinations of variables of the six basic types; this is discussed in more detail in Chapter 13 and Appendix E.



SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

When we encounter a related *group* of data, the elements of which should all be treated the same way, the efficient way to do this is to make the group into an *array* of values. This is done by *dimensioning* an array name (which follows the usual naming conventions) to the appropriate size for the data. For example,

```
REAL ART
DIMENSION ART(100)
```

```
INTEGER MUSIC (-8:8)
```

```
REAL ART(100)
```

```
CHARACTER COMB*4
DIMENSION COMB (0:20, 0:20)
```

each create consecutive locations under a single name, which can then be referred to individually by a *subscript* that indicates the position of the particular element in the array. A subscript may be any integer expression that evaluates to a value within the *range* of values designated by the array dimension. A dimension may be designated to be a specific length (e.g., 100), or it may be between bounds—array (low:high). A *loop* is a particularly effective way of dealing with all of the elements of an array in turn, since the loop index can be the subscript.

An *implied list* can be used for I/O of arrays. Such a list appears within parentheses, names the array with integer variable subscript(s), and then indicates the range of values the subscript(s) are to take on. For example,

REAL BOOK (200)	INTEGER COUNT (1965:2000)
READ*, (BOOK(I), I=1,200)	PRINT*, (COUNT(K), K=1990,2000)

If *all* of the elements of an array are intended for I/O, *in the order they are stored in memory*, then the I/O statement may simply name the array without any subscripts or list, and the whole array is assumed. Thus, in our first example with the array BOOK, we could have written:

```
READ*, BOOK
```

An array may be used to contain data to be operated on, or it may be used as a *reference array* containing calculated values that will be referred to at various times in the program. For example, one could create a reference array of prime numbers that would then be used for coding messages, solving number problems, and the like; such a reference array could even be written out to tape or disk (see Chapter 11) for later use by several programs.

Arrays can also be used to create “user-defined types,” that is, types not standardly available in the language, by creating character arrays with the names of the values in the type listed:

```
CHARACTER*10 COLOR(6)
DATA COLOR/ 'RED','BLUE','YELLOW','GREEN','ORANGE','PURPLE' /
```

Note that a DATA statement can be used to fill the elements of an array; the name of the locations to be filled can be expressed as an implied list or, as in I/O, as just the array name.

Arrays in FORTRAN 77 can have up to 7 different dimensions. The most useful higher-dimensional array for our purposes is the two-dimensional array, which can be used to store tables or to do matrix operations. The first dimension in a 2-D array is the *row* dimension, the second is the *column* dimension.

An array can be *searched* systematically for the value or values that fit a

certain requirement. A one-dimensional array, or a single dimension of a higher-dimensional array, can be *sorted* so that the values are in ascending or descending order. Two simple sorts—a bubble sort and a selection sort—were implemented in programs in the text.



EXERCISES

1. Read in a short array (say, 40 values) of integer equipment inventory counts, print them out with their subscripts alongside, and perform several of the operations discussed in the chapter—calculate the mean and standard deviation, sort the array, calculate the *median* (middle) value and the *mode* (most frequently occurring value), find and print out the largest and the smallest values.
2. Fill a *reference array* with the cubes of the digits 0–9 (dimension the array KUBE(0:9) for ease of reference). Then use this array to determine and print out all of the three-digit integers (from 000–999) that are equal to the sums of the cubes of their digits. Use triply nested DO loops on the digits for this exercise. Now try rewriting the program *without* DO loops, using only IFs and GO TOs. You will find that, in the case of nested loops, DOs provide a definite advantage.
3. Output your inventory array from exercise 1 using an implied list, and then by simply naming the array without subscripts. Make your output such that 10 values are printed per line.
4. The mathematician Fibonacci, in speculating about the growth of rabbit populations over time, came up with the following formula for the relationship between the current population and that of the two previous time periods:

$$F_n = F_{n-1} + F_{n-2} \quad (F_1 = F_2 = 1)$$

A relatively simple way to generate a sequence of Fibonacci numbers is by using an array. Store the value 1 in locations 1 and 2 of the array, and then apply the formula to fill all the other locations of the array. Fill the array to 20 locations, and then print out the first 20 Fibonacci numbers, separated by commas, 10 values per line.

5. Write code to *interchange* two already filled real arrays A and B, both dimensioned to 200 locations.
- 6. Write FORTRAN code to *invert* a one-dimensional real array A containing 200 values.
- 7. To convert an integer number from base 10 to some other base b, you divide through by b, collecting the remainders and then dividing the quotient

again by b, until the quotient goes to zero (refer to Appendix A). The remainders give you, in order from least significant to most significant, the digits of the value in the new base b. Write FORTRAN code which will accept any input integer value N and a base B (integer from 2 to 9) to convert it to, and perform the conversion, storing the remainders in an array. Print out the resulting value in base b by printing the values stored in the array, immediately adjacent to one another, in reverse order.

8. Find the integer less than 1000 whose sum of factors is 888; print out the number and all its factors.

9. Two real two-dimensional arrays B and C, each of 30 rows and 50 columns, have been filled with values. Write FORTRAN instructions to *interchange* the contents of the two arrays.

10. Arrays can be used for manipulating/coding messages. For example, the following one-dimensional array could have a string of characters read into it, and then some characters used to create a new output message. The DATA statement provides the reference to control which characters are printed. What does the output say?

```
CHARACTER LIKE(24), OUT(19)
DATA IT/15,9,7,23,20,19,4,5,10,8,20,6,7,16,19,9,23,23,20/
      READ 5, LIKE
5   FORMAT(24A1)
      DO 30 I = 1, 19
30   OUT(I) = LIKE(IT(I))
      PRINT*,OUT
```

The program reads the following line, beginning with 'A' in column 1.

ALL THE WORLD'S A STAGE

11. Messages can also be manipulated by just rearranging the character data in a two-dimensional array. The following program reads into the array in one order, and prints out in another. What message does the program print out?

```
CHARACTER HAM (6,8)
READ 7, ((HAM (I,J), J = 1,8), I = 1, 6)
7   FORMAT(50A1)
      PRINT 9, HAM
9   FORMAT(3X,50A1)
```

The program reads the following line, beginning with a 'W' in column 1; the \emptyset 's represent blanks.

WHERMOLRHPOKAWEEAIFNHATEI, NISHCWSHONOAEOHNHN

12. Modify the bubble sort procedure in the text to make use of the fact that the largest value *sinks* to the bottom on each pass, and so reduce the number of the maximum position examined on each pass. Then see if you can improve it further by keeping track of the position of the last interchange on each pass, and only going that deep on the next pass. One final refinement would be to implement the “cocktail-shaker” sort, by reversing the direction of the comparison pass on each successive turn. This will drive values to their proper position faster. Try out your modified sort procedures on actual data.
- 13. A two-dimensional array C (30 x 40) contains information on elapsed time of a projectile in flight in the first column, on which the data in the array is to be sorted. Write code to rearrange the array so that the first column values are in *descending* order. This means that every time you interchange two values that are out of order, you must switch the pair of *rows* in which the two values occur.
 - 14. “Sieve of Eratosthenes”. A prime number, as you know, is an integer whose only factors are 1 and the number itself (therefore, automatically all even numbers other than 2 are excluded). A neat and efficient way to find prime numbers was devised by Eratosthenes, a third Century B.C. Greek, and adapts well to an exercise in the use of one-dimensional arrays; it is called the “sieve of Eratosthenes.” We know that 2 is a prime number and that from then on, only odd numbers are candidates. We can thus begin by writing up a list (array) of candidates (all the odd numbers in a given range); begin your program by filling a one-dimensional array (of length 499 elements) with all of the odd numbers from 3 to 999. Now that the list is complete, we can begin at the top, recognizing primes but “crossing out” all of the numbers in the list that are not primes (that is, all multiples of the primes we find).

Begin at the top of the list; you find 3, which is a prime. Now go down the list crossing out all the multiples of 3. You can do this by looking at each entry in the list below the position where you found 3, using the remainder formula to test whether the entry you are looking at is divisible by 3, and crossing it out if it is divisible by 3. You can “cross out” an entry in the list by setting it to zero; later you can come back and squeeze the zeros out of the list. There is a more efficient way to get rid of the multiples of 3, if you recognize it; the multiples of 3 (and of other prime entries in the list) are stored at periodic positions in the list. All you really have to do is go to those positions, if you can identify them, and cross out the multiples there. Once you have crossed out the multiples of 3, go back to the next (nonzero) entry at the top of the list (in this case, 5), and go through crossing out its multiples.

How far do you have to continue this? You only need to cross out multiples in the