

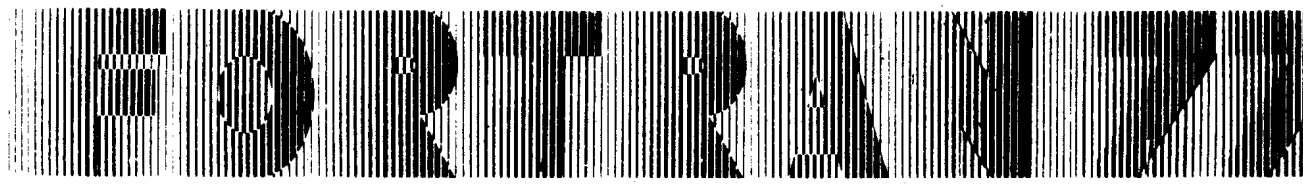
F O R T R A N

LANGUAGE AND STYLE

A Structured Guide to Fortran 77

MICHAEL J. MERCHANT

73.87221
M55



LANGUAGE AND STYLE

A Structured Guide to Fortran 77

MICHAEL J. MERCHANT



Wadsworth Publishing Company
Belmont, California

A Division of Wadsworth, Inc.

5506347

5506347

Computer Science Editor: Jon Thompson
Editorial production services: Cobb/Dunlop Publisher
Services, Inc.

EC81/06
©1981 by Wadsworth, Inc. All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transcribed, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher, Wadsworth Publishing Company, Belmont, California 94002, a division of Wadsworth, Inc.

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 85 84 83 82 81

Library of Congress Cataloging in Publication Data

Merchant, Michael J.
FORTRAN 77: language and style.

Includes index.

1. FORTRAN (Computer program language) 2. Structured programming. I. Title.

QA76.73.F25M463 001.64'24 80-39551
ISBN 0-534-00920-4



ABOUT THE ART

The publisher acknowledges with great appreciation the generosity of the persons and institutions named below for providing the computer-generated images that appear at the beginning of each chapter and on the cover of this book. In addition to their visual appeal, these images illustrate an important application of FORTRAN, the original high-level, scientific language.


The images that precede Chapters 1 and 7 are the result of computer graphics research into the problem of depicting three-dimensional objects. The "ball of yarn" image was produced in part by "random walk" commands. Reprinted by permission of Ken Knowlton, Bell Laboratories.

Chapters 2, 8, and 9, as well as the text's cover, are illustrated by simulation experiments involving stress, contour, and other factors. These studies were conducted at the University of Utah, and the graphics are reprinted by permission of Hank Christiansen, Brigham Young University.

Three images were produced with the aid of a massive graphics program running on one of the largest computer installations in the world. The dual magnets (Chapter 3) and the nose cone impact simulation (Chapter 10) are reprinted by permission of Bruce Brown, and the portion of the DNA molecule (Chapter 4) is reprinted by permission of Nelson Max, Lawrence Livermore National Laboratory.

The illustration of pawns on a chessboard (Chapter 6) is one example of many synthetically generated images of complex scenes produced at the University of Utah under a research contract with DARPA, and it is reprinted by permission of Martin Newell, Xerox Palo Alto Research Center.

"L Space," the work opening Chapter 5, is an artistic exercise in balancing random parameters and control parameters created at Syracuse University. It is reprinted by permission of Judson Rosebush, Digital Effects Incorporated.



PREFACE

This book was written with two questions in mind: What is the best way to teach the new FORTRAN language, and what is the best way to teach structured programming using FORTRAN?

In answering the first question, choices about the selection of topics and the order of presentation often were dictated by features of the 1968 FORTRAN language standard. Character data, for example, has been placed on an equal footing with numerical data and has been covered early in the text. The addition of a CHARACTER data type not only extends the capability of the language, but also makes it easier to teach, since it opens a broad range of applications to the student which illustrate important programming concepts without requiring advanced mathematics. The addition of list-directed input and output allows students to write programs without having to master the rather complicated technical details of FORMAT statements.

Teaching structured programming with FORTRAN was difficult before the new standard because the language lacked the necessary control statements. The addition of the IF-THEN-ELSE structure has made it possible to teach students to write clearly structured programs.

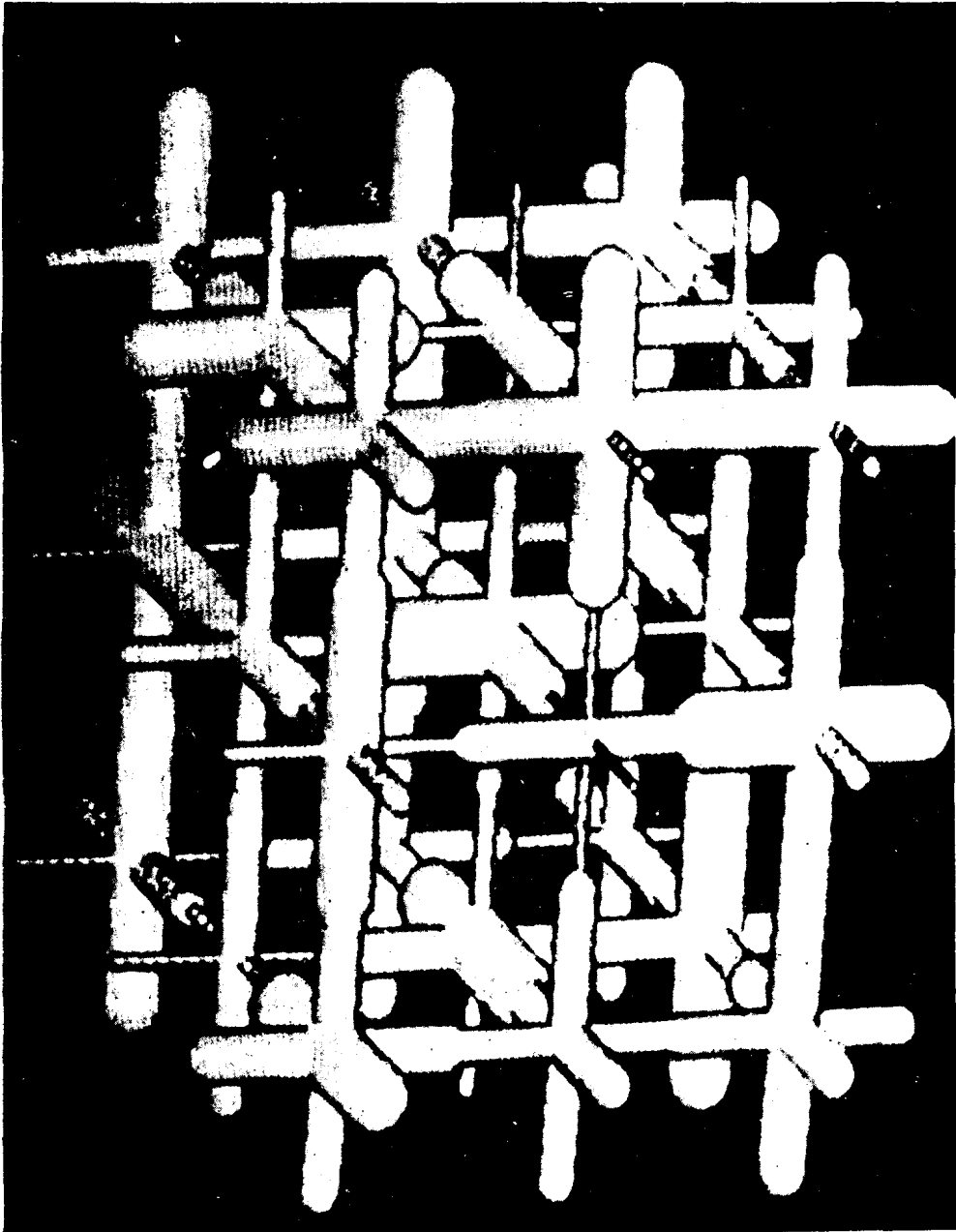
Throughout the book, programming language concepts are integrated with concepts of programming style. Style modules in each chapter give practical advice on how to write a program and how to apply the FORTRAN language. Top-down design and structured programming are continually emphasized.

In the firm belief that programming is a subject best learned by practice, Chapters 1 and 2 are organized to get the student running programs right from the start. In Chapter 1, students see a complete program developed from top-down design to finished code. Chapter 2 guides students through the details of running the program on a computer.

The entire text can be covered in a one-semester introduction to programming with FORTRAN. Students who are already familiar with another programming language could omit Chapters 1 and 2, or cover them rapidly. By omitting the sections labeled as optional, Chapters 1 through 7 can serve as a short course in FORTRAN or as a supplement to a course in introductory data processing.

Although the responsibility for any deficiencies in this text is my own, I would like to express my sincere thanks to the many people who read the manuscript and made valuable suggestions for its improvement: John H. Crenshaw, Western Kentucky University; Henry A. Etlinger, Rochester Institute of Technology; Krzysztof Frankowski, University of Minnesota; Robert Frye, Central Michigan University; Kenneth Geller, Drexel University; Mark Luker, University of Minnesota—Duluth; Franz Oppacher, Concordia University; and Frank G. Walters, University of Missouri—Rolla.

Michael Merchant



CONTENTS

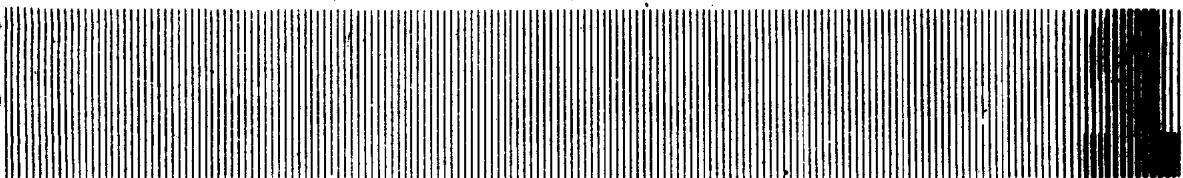
Preface	vii
1 COMPUTERS, ALGORITHMS, AND PROGRAM DESIGN	1
Introduction	1
Algorithms	2
Top-Down Design	3
Structured Flowcharts	8
The FORTRAN Language	10
What Computers Do	11
Summary	24
Vocabulary	25
Exercises	26
2 RUNNING YOUR FIRST PROGRAM	35
Introduction	36
Preparing Your Program for the Computer	36
Style Module—Typing FORTRAN Statements	40
Style Module—Using Comments	41
Preparing Your Data	43
Setting Up Your Job Deck	45
Running Your Program	46
Running on an Interactive System	48
Errors and Diagnostics	50
Operating Systems	53
Summary	54
Vocabulary	55
Exercises	55

3	CONSTANTS, VARIABLES, AND EXPRESSIONS	59
	Introduction	59
	Constants	59
	Exponential Form	61
	Why Integers and Real Numbers Are Different	62
	Variables	63
	Executable and Nonexecutable Statements	64
	Implicit Type Declaration	65
	Default Type Declaration	66
	Style Module—Choosing Variable Names	67
	Style Module—Using Type Statements	67
	The Assignment Statement	68
	Expressions	69
	Real and Integer Arithmetic	74
	Type Conversion with an Assignment Statement	76
	Symbolic Constants	78
	Style Module—Symbolic Programming	80
	Style Module—Writing Expressions	81
	Summary	83
	Vocabulary	84
	Exercises	84
4	CONTROL STATEMENTS AND STRUCTURED PROGRAMMING	91
	Introduction	91
	Control Structures	91
	The GO TO Statement	93
	Style Module—Using Statement Labels	94
	STOP, PAUSE, and END Statements	94
	The End-of-File Specifier in the READ Statement	96
	The IF-THEN-ELSE Statements	98
	Relational Expressions	99
	Logical Expressions	100
	Other Forms of the IF Statement	104
	Style Module—Structured Programming	107
	Style Module—Loops with an Exit in the Middle	112
	The ELSE IF Statement	118
	The DO Statement	124
	The CONTINUE Statement	128
	Rules for DO Loops	128
	More Rules for DO Loops	130
	Style Module—Misuse of the DO Statement	132

Style Module—Comparison of Real Expressions for Equality	134
Style Module—Real DO Variables	135
Summary	136
Vocabulary	136
Exercises	137
5 CHARACTER DATA	145
Introduction	145
Character Constants	145
Character Variables	146
Symbolic Character Constants	148
Character Expressions	150
Substrings	151
Character Assignment	154
Another Look at Input and Output	156
Comparing Character Expressions	163
Summary	171
Vocabulary	172
Exercises	172
6 ARRAYS	181
Introduction	181
Arrays and Subscripts in FORTRAN	182
Dimension Declarations	183
Style Module—Subscript Errors and How to Prevent Them	196
Style Module—A Subtle Subscript Error	202
Two-Dimensional Arrays	204
Higher-Dimensional Arrays	206
Implied DO Loops	207
Style Module—Data Structures and Top-Down Design	211
Summary	218
Vocabulary	219
Exercises	219
7 SUBPROGRAMS	229
Introduction	229
Intrinsic Functions	230

Function Subprograms	239
The FUNCTION Statement	241
Program Units	243
Arguments	243
The RETURN Statement	245
Character Data as a Function Argument	247
Character Data as a Function Value	248
Changing the Value of an Argument	251
Subroutines	254
Arrays in Subprograms	258
Style Module—Subscript Errors in Subprograms	263
Style Module—Subprogram Style	265
Style Module—Subprograms and Data Structures	267
Style Module—Subprograms and Top-Down Design	270
Top-Down Testing	276
Subprogram Structure Diagrams	277
Summary	278
Vocabulary	279
Exercises	280
 8 FORMATTED INPUT AND OUTPUT	 293
Introduction	293
The Formatted PRINT and READ Statements	293
The FORMAT Statement	295
Input and Output Fields	296
The I Edit Descriptor	299
The F Edit Descriptor	302
The Apostrophe Edit Descriptor	308
The X Edit Descriptor	308
The E Edit Descriptor	309
The A Edit Descriptor	311
The L Edit Descriptor	312
Carriage Control	312
Some Common Mistakes	315
The BN and BZ Edit Descriptors	317
The Slash in the FORMAT Statement	318
The T, TL, and TR Edit Descriptors	320
Repeat Factors	321
Interaction of the Input-Output List with the Format	322
An Alternative Way of Writing Format Specifications	324
Summary	326
Vocabulary	327
Exercises	327

9	FILES	337
	Introduction	337
	Records, Files, and Units	338
	READ and WRITE Statements	339
	The OPEN and CLOSE Statements	341
	The Input-Output Status Specifier	342
	REWIND, BACKSPACE, and ENDFILE Statements	346
	Internal Files	350
	The Unformatted READ and WRITE Statements	356
	Direct Access Files	357
	Summary	363
	Vocabulary	364
	Exercises	364
10	ADDITIONAL TOPICS	371
	Introduction	371
	Complex Numbers	371
	Double-Precision Numbers	378
	The DATA Statement	381
	The Computed GO TO	383
	The Assigned GO TO	384
	The Arithmetic IF Statement	385
	Common Storage	386
	BLOCK DATA Subprograms	389
	Statement Functions	389
	The SAVE Statement	391
	Alternate Returns from a SUBROUTINE	392
	The EXTERNAL Statement	393
	The EQUIVALENCE Statement	395
	Summary	396
	Vocabulary	397
	Exercises	397
A	INTRINSIC FUNCTIONS	403
B	ANSWERS TO SELECTED EXERCISES	415
	INDEX	445



COMPUTERS, ALGORITHMS, AND PROGRAM DESIGN

INTRODUCTION In science fiction novels, in movies, and in cartoons, computers are often represented as being much like superintelligent human beings. In such works of fiction, computers converse fluently in the English language, exercise independent judgment in solving problems, and are able to retrieve from their memory banks all pertinent data for any problem at a moment's notice. It is an exciting dream. And one of the exciting aspects of studying computing is that the dream may someday come true—perhaps within your lifetime.

But the day of the genuinely intelligent computer is still in the future. The computers of today are incredibly fast, and they can follow extremely complex sets of instructions, but they are just machines—in some ways, rather simple machines at that. All a computer can really do is to follow very simple orders which have been carefully thought out by a programmer and written in a programming language like FORTRAN.

In this book you will learn two things. The first thing is how to solve problems with a computer—that is, how to make up instructions to the computer to get your job done. Specifically, you will learn to design and write an **algorithm**, which is a procedure that a computer can carry out.

In this chapter we explain what an algorithm is, how you can represent an algorithm by a kind of diagram called a **flowchart**, and how you can use the method of **top-down design** to invent an algorithm.

After you design an algorithm to solve your problem, you must express the algorithm in a language that the computer can understand. A **computer program** is an algorithm written in a programming language like FORTRAN. So the second thing you will learn, which goes hand in hand with the first, is how to write a FORTRAN program.

5506347

Chapter 1 will give you an introduction to the FORTRAN language. In Chapter 2, you will see how to run a FORTRAN program on the computer.

ALGORITHMS

People are often imprecise when giving instructions to each other. A simple instruction such as "Go to the store for a loaf of bread." requires hundreds of decisions to carry out: Should you go out the front door or the back? Should you turn left or right? Which way is the store? Where is the bread? Do you want whole-wheat, sourdough, or caraway rye? People, having intelligence and reasoning ability, can figure out the real meaning of general, ambiguous instructions. Computers, on the other hand, have no common sense. When you write a procedure for a computer to carry out, every instruction must be explicit.

Suppose you want to solve a math problem using a calculator. You do not have a calculator, but you have a friend who does, so you call her on the phone to ask for help. She is not at home, but her young brother, who knows very little math, offers to work the calculator if you will tell him exactly what to do. Now you must specify how to solve your problem using instructions that are so precise and unambiguous that he cannot possibly misinterpret them. You might say, "Enter the number 56.2, press the plus key, enter the number 475.3, press the plus key, enter the number 11.63, press the equals key, and read me the number in lights at the top." This is an *algorithm*, expressed in English.

An **algorithm*** is a step-by-step procedure for solving a problem. A correct algorithm must meet three conditions:

1. Each step in the algorithm must be an instruction that can be carried out.
2. The order of the steps must be precisely determined
3. The algorithm must eventually terminate.

An algorithm does not necessarily have to be written for a computer. You could carry out the instructions with a paper and pencil, for example. A cookbook might give an "algorithm" for baking chocolate chip cookies. The instruction "Bake until done." might be meaningful for a human cook, so it satisfies the first condition for an algorithm.

It would be convenient if you could just tell a computer "Solve the following problem . . ." and make the machine obey. A computer, however, has an **instruction set** consisting of only a hundred or so basic commands that it can carry out electronically. These commands are similar to the commands you can give to a calculator by pressing the keys. For example, you can tell a computer to add two numbers. The first condition for a correct algorithm means that when writing an algorithm for a computer, each step must be something that a computer can carry out by executing these basic instructions.

In carrying out the instructions in an algorithm, the machine performs one

*The word *algorithm* comes from the name of the Persian mathematician al-Khowarizmi (c. 825).

instruction at a time. The second condition for a correct algorithm means that the algorithm must precisely specify the order in which the instructions are performed.

The third condition means that an algorithm must not go on forever. The following procedure, then, is *not* an algorithm.

Procedure to Count

- Step 1 Let N equal zero.
- Step 2 Add 1 to N .
- Step 3 Go to Step 2.

If you tried to have a computer carry out this procedure, the machine would, in theory, run forever. The following procedure, on the other hand, *is* an algorithm, because it will eventually terminate.

Algorithm to Count to One Million

- Step 1 Let N equal zero.
- Step 2 Add 1 to N .
- Step 3 If N is less than 1,000,000, then go to Step 2; otherwise, halt.

TOP-DOWN DESIGN

When working on an algorithm for a simple problem, a solution may suddenly occur to you after just thinking about the problem for a while. Practical computer applications, however, are seldom so simple. Professional programmers commonly write programs consisting of thousands of computer instructions. Designing such a program can be as complicated as designing a machine with thousands of parts. In order for it to work, all the pieces must fit together in an organized framework.

Top-down design is an approach to the problem of designing an algorithm. It is a method you can use to organize your work and also to organize your algorithm.

In some ways, designing an algorithm is similar to writing an essay. When writing an essay, you begin with a general idea of what you want to say. You then proceed to organize your thoughts and choose words to convey your meaning to the reader. When designing an algorithm, you begin with a general idea of what you want it to do. You must then organize your ideas and choose the right sequence of instructions to the computer that make it carry out the desired actions. But in programming, as in writing, it is often difficult to know where to begin.

A good way to begin writing an essay is to make an outline. First, you set forth the main topics to be covered, as in the following example:

Gettysburg Speech

- I. Conception of the nation
- II. The current civil war
- III. Our purpose here today
- IV. Our resolve for the future

This bare outline provides a framework for the essay, into which all the paragraphs and sentences will fit. The overall structure is determined, and what remains is to elaborate the topics in more detail. You can do that by adding subheadings under each topic, as in the following example:

Gettysburg Speech

- I. Conception of the nation
 - A. The nation was founded 87 years ago.
 - B. It was conceived in liberty.
 - C. It was dedicated to the proposition that all men (persons?) are created equal.
- II. The current civil war
 - A. We are now engaged in civil war.
 - B. The war tests the ability of this nation, as conceived by its founders, to endure.
- III. Our purpose here today
 - A. We are meeting on a battlefield of the civil war.
 - B. We dedicate a portion of this field as a cemetery.
 - C. The soldiers who fought here have consecrated this ground with their brave struggle.
- IV. Our resolve for the future
 - A. We must dedicate ourselves to the unfinished work before us.
 - B. We must devote ourselves to the cause for which the dead have fought.
 - C. We must preserve the ideals of freedom in which the nation was conceived.

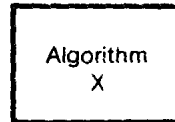
Of course, even with the best of outlines to work from, one cannot hope always to emulate Lincoln's deathless prose; but good writing is always well organized, and making an outline is a very useful way to begin.

In programming, as in writing, good organization is vital. Before beginning to fill in the details, it is best to have a clear overall plan. The method of top-down design is like making an outline of an algorithm. The first step is to formulate a general plan—like writing the main topics in an outline. Next, you fill in more detail, like making subheads in an outline, to specify how to carry out each major step. By successively refining this plan, adding more detail at each stage in the development, you arrive at your ultimate goal: a precise algorithm that can be expressed in terms of the basic types of instructions that a computer can follow.

As an illustration, suppose you want to write an algorithm (we'll call it Algorithm X), to solve some homework problem for your algebra class. As you begin, you have a rough idea of what the algorithm should do, but little idea of the specific instructions. At this point, you conceive of Algorithm X as a single entity (Figure 1.1).

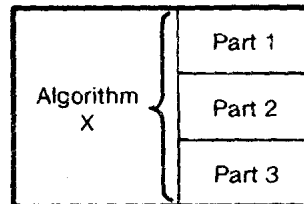
As you think about the problem some more, you realize that there are really three parts to the solution, and you say to yourself, "If I could do Part 1, and then Part 2, and Part 3, that would solve the problem." This is your **top-level design**, as shown in Figure 1.2.

FIGURE 1.1



As you begin to write an algorithm you conceive of it as a single entry.

FIGURE 1.2



In the method of top-down design, you first specify the general organization of the algorithm.

You may not have a detailed algorithm yet, but you have made some progress; instead of one big problem, you have three smaller ones to solve. Continuing your analysis, you might find that Part 1 can be broken down into two subproblems—Part 1A and Part 1B—and that Parts 2 and 3 can be similarly subdivided. This is your second-level design, as shown in Figure 1.3. If the algorithm is complicated, you may need to carry this process to yet another level of refinement. Eventually, you arrive at a detailed plan that you can write in computer language.

In summary, the basic principle of top-down design is this:

Concentrate first on the overall design of the algorithm. Write it as a sequence of general steps to be carried out. Then, using the same method, fill in the details for each step.

An important principle in top-down design is **validation**, which means analyzing your algorithm to make sure that it is correct. Even though your first draft of an algorithm may be an outline, it must be a valid solution to the problem you are trying to solve. The question you should ask yourself is this: If you could carry out each step in your proposed algorithm, would that really solve the problem? Thus, a second principle of top-down design is this:

At each stage of the top-down design process, validate your algorithm by analyzing it for correctness. Do not expand on the details until you are sure that the overall plan is sound.