ALLEN B. TUCKER, JR.

Programming Languages

PROGRAMMING LANGUAGES

Allen B. Tucker, Jr.
Georgetown University

McGRAW-HILL BOOK COMPANY

New York St. Louis San Francisco Auckland Bogotá Düsseldorf
Johannesburg London Madrid Mexico Montreal New Delhi Panama Paris
São Paulo Singapore Sydney Tokyo Toronto

PROGRAMMING LANGUAGES

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

This book was set in Helvetica by Progressive Typographers. The editors were Peter D. Nalle, Claudia A. Hepburn, and Matthew Cahill; the production supervisor was Leroy A. Young. The drawings were done by Long Island Technical Illustrators.
R. R. Donnelley & Sons Company was printer and binder.

Library of Congress Cataloging in Publication Data

Tucker, Allen B
Programming languages.

(McGraw-Hill computer science series)
Includes index.
1. Programming languages. (Electronic computers)
I. Title.
QA76.7.T8 001.6'424 76-26031
ISBN 0-07-065415-8

Preface

Language is not an abstract construction of the learned, or of dictionary makers, but is something arising out of the work, needs, ties, joys, affections, tastes, of long generations of humanity, and has its bases broad and low, close to the ground.

Walt Whitman
Slang in America

Whitman is of course referring to natural languages, rather than programming languages. However, in light of the present distance between the view of the designer and that of the professional programmer toward programming languages, this statement might just as well represent the programmer's view.

Most texts on programming languages tend to represent the designer's view rather than the programmer's. They discuss principles of language design rather than applications. Although principles are important to any study of the subject, they should not be presented *to the exclusion* of the language needs of the various programming applications. After all, a programming language is—first and foremost—a language for *programming*.

This text attempts to unify the study of programming language principles with the study of programming language applications. We take the programmer's point of view, as well as the designer's. Thus, we present language features that have value in day-to-day programming applications; we

illustrate language strengths and weaknesses by showing their use in solving various representative "benchmark" problems; we run programs using a variety of different computers and compilers; and we evaluate languages and their compilers using uniform and meaningful evaluation criteria.

Six languages have been chosen for presentation, evaluation, and comparison in this text: ALGOL, FORTRAN, COBOL, PL/I, RPG, and SNOBOL. These particular languages were selected because of their wide usage, different functional and stylistic characteristics, and range of applications. One chapter is dedicated to each of these six languages. A brief glimpse at the table of contents shows that the six language chapters have mutually identical organizations. Moreover, the same examples and exercises are repeated in the different language chapters. These two characteristics should encourage frequent cross-referencing between chapters as different languages are studied.

Each language chapter also shows the application of its language to one or more representative benchmark programming problems, showing a complete and operational implementation of the problem in that language. Furthermore, the resulting programs were run on a variety of different computers and compilers, in order to demonstrate the language's transferability. Finally, these benchmark programs are used as a basis for evaluating the language at the conclusion of its chapter.

Our criteria for language evaluation are fully developed and illustrated in Chapter 1. In this chapter we support the view that the *value* of a language is determined not only by its compiler's performance (in terms of storage and speed), but also by such characteristics as compile-time and execution-time diagnostics, debugging features, built-in functions, and uniformity of expression. These characteristics are more difficult to evaluate than a compiler's speed since they are not quantifiable. However, they are of such central importance to the overall quality of a language and its implementations that they must be assessed as carefully as possible.

Chapter 8 concludes the text with a comparative evaluation of the six languages in the different application areas that they were designed to serve. These comparisons use the same criteria developed in Chapter 1 and used throughout the language chapters (2 to 7). In Chapter 8 we also discuss factors other than language quality (e.g., the programmer market) that tend to influence programming language selection and usage.

This book was written primarily for use as the text in a junior-senior-graduate level course in programming languages (course I2 in the ACM 1968 curriculum guidelines, or course SE 4 in the IEEE model curriculum). This is not an introductory text. We assume that the reader has a sound knowledge of programming and at least one programming language.

Chapters 1 and 8 should be included in any one-semester course using this text. Beyond that, any selection can be made from among the six language chapters since they are mutually independent. The particular languages

selected for study will depend on the students' particular language experience and the instructor's preferences. For example, if a class has significant PL/I and FORTRAN experience, the instructor may elect to cover the ALGOL, COBOL, and SNOBOL chapters, in addition to Chapters 1 and 8.

Naturally, the study of three different languages in a one-semester course will not yield the depth of knowledge in any one language that would have been gained in a "one-language" course. However, the student should learn enough about each language to use it, to constructively evaluate it, and to compare it with other languages in an application area. This is our basic educational objective.

The student should also gain a comprehensive basis for evaluating programming languages from this experience. That basis can, in fact, be applied to languages other than those presented in this book. Such an ability is essential for those who intend to contribute to the design and implementation of future programming languages.

This text may also be used by professional programmers and programming supervisors who would like to widen their perspective on languages, compilers, and programming applications. For instance, a COBOL or FORTRAN programmer can use this book to get a quick and nontrivial introduction to PL/I, thus gaining a basis for constructively evaluating PL/I for use in his or her own applications. Since the FORTRAN and PL/I chapters are identically organized and contain programs that solve the same benchmark problems, the programmer has a direct basis for comparatively evaluating the two languages in the scientific application area. The same can be done for COBOL and PL/I, and thus the two languages can be meaningfully compared in the data processing application area.

The exercises at the end of each chapter are designed to put the reader into an active, rather than a passive, role with each language. Some exercises simply review the reader's understanding of essential language features, such as writing subprograms. Other exercises, however, require the reader to write and debug a program using the language, and then to evaluate the language on the basis of this experience. Since the text teaches each language, including implementation dependent differences, the student should require no supplementary materials other than a knowledge of the "job control" cards required at the student's own computer installation.

When studying programming languages, one must always be careful to distinguish between the language itself and the particular machines, operating systems, and compilers which support it. To emphasize this distinction, we have taken two basic steps throughout this text. First, we use the standard (or commonly accepted) version as the authority for a language's definition. Features that are available only for certain compilers are presented in a separate part of the chapter. We have also run the benchmark programs for each language on at least two different machines and/or compilers. Thus, our language evaluations and comparisons are highly machine independent.

The different machines used are the following:

Burroughs	B6700
CDC	Cyber 70
DEC	PDP-10
Honeywell	6000
IBM	370/145
UNIVAC	1108

It would be neither practical nor instructive to run all the benchmark programs on all of these machines. However, in a comparison of two different languages' compile and execute speeds for the same benchmark programming problem, it is essential that the two programs be run on the same machine and operating system environment. We thus ran every benchmark program at least once on the same machine and operating system (the IBM 370-145 OS-MVT system since it was the one most accessible to the author). The results of those runs are used in Chapter 8 as the primary basis for comparing compile and execute speeds. Concurrently, the benchmark program listings shown in Chapters 2 to 7 must unavoidably reflect some IBM implementation dependent characteristics. We feel that these have been minimized. Furthermore, the text of each chapter identifies implementation dependent language features. Thus readers may readily adapt any of our benchmark programs for running at their own installations.

Although it is not the main purpose of this book to teach programming style, our exhibition of certain stylistic preferences is unavoidable. In the ALGOL and PL/I chapters, our programs reflect a structured programming style. This was also possible to a lesser extent in the FORTRAN and COBOL chapters. In all but the RPG chapter, our programs make extensive and mutually consistent use of subprogramming facilities. A reasonable level of commentary documentation is provided throughout all the program listings.

Permit us to add one final thought concerning objectivity. The relative merits of different programming languages have continually been a subject of passionate controversy. The most avid proponents of a language often have difficulty being objective when debating its merits. Each of us, as programmers, has our own personal "best friend" language. In your work with this book, try to put aside your own biases about particular languages, especially the ones you know the best. By doing this, we can all gain new and constructive insights into the present and future states of programming languages.

ACKNOWLEDGMENTS

This book results from the evolution of an attitude toward programming languages. Two particular individuals have had special influence on this attitude, and I would like to acknowledge them. Mr. Carl Nelson—a colleague pro-

grammer and systems analyst at Norton Company in Worcester, Massachusetts, from 1963 to 1967—taught me many practical aspects of good programming style and design. Professor Albert Grau—a teacher and advisor at Northwestern University from 1967 to 1970—gave me a new and important perspective on the influence of ALGOL on the design and implementation of programming languages.

Another important characteristic of this book is its demonstration that programs in the various languages are reasonably transferable among different machines and compilers. To do the actual runs required many hours of preparing control cards, converting decks between ASCII, EBCDIC, and other character sets, scheduling the runs, diagnosing disparate error messages, and so forth. The following individuals and institutions helped me tremendously with this effort, and I thank them sincerely.

Richard Bonano	Catholic University of America
Ira Gold	University of Maryland
Gregory Hislop	Queen's University
Jan Larsen and Lloyd Wall	Georgetown University
Ralph Popp	Honeywell Information Systems, Inc.
William Ulman	Utility Network of America

Thanks are also due to Giuliano Gnugnoll, who contributed to this book's initial design, and to Herbert Maisel, who gave me the time to write it. A special note of appreciation goes to Virginia Stehman and Maida Tucker for their tireless typing, proofreading, and editing of the manuscript, especially at stages when it was in *very* rough form.

To my parents, Allen and Louise Tucker, goes my sincerest appreciation for all they have given me. Finally, to my family—Maida, Jenny, and Brian—I am grateful for their continual love and support, especially during the times that this book's development received more of my attention than it deserved.

Allen B. Tucker, Jr.

Contents

1

INTRODUCTION AND OBJECTIVES

Our purpose in this text is to study the main features of six widely used programming languages, and then to evaluate their respective strengths and weaknesses in the application areas which they were designed to serve.

To attain this end, we will not only study the *principles* of language design, which include the evaluation of expressions, dynamic storage allocation, recursion, and so forth. We will also study the *applications* of programming languages. This will enable us to focus attention on the special features that these languages possess, so that we may become equipped to judge how well they support productive programming.

Although this is an ambitious task, such a level of understanding about languages and their applications ought to be widely attained. This chapter presents our overall perspective on programming languages, their uses, and their evaluation. Here also we present the basis we will use for evaluating languages throughout the remainder of this text.

A "high-level language" has a programming style and functional capa-

bilities that render it closer to a programmer's natural language of discourse than a "machine-level language." For example, a scientific programmer using a high-level language can directly state Z = X + Y * W to denote an evaluation of the algebraic expression $X + YW$ and an identification of the result as Z. Similarly, a data processing programmer using a high-level language can directly state "WRITE A-LINE AFTER ADVANCING 2 LINES" to denote a transfer of data to a specific line of the printed page. To specify either of these actions in a machine-level language, the programmer must use a number of statements, rather than just one, and encode them within more strict syntactic and semantic constraints.

For a language to be high level, it should be substantially independent in its definition from any particular computer's architectural characteristics, instruction set, word length, internal speeds, input-output devices, and so forth. This independence implies that a high-level language must be implemented with a relatively sophisticated kind of software, specifically a compiler or an interpreter. The nature of translation from a high-level language to a machine-level language for a specific computer is therefore necessarily complex. By way of comparison, the task of an assembler to translate from symbolic machine-level language to machine language is much more straightforward.

However, high-level languages are not as close in many respects to the programmer's medium of discourse as "declarative" (or "very high-level") languages that are found in data-base management systems, statistics packages, and so forth. Declarative languages are distinguished by the fact that they principally allow the user to prescribe *what* to do rather than *how* to do it. High-level languages, however, retain as their major component the algorithmic capability. This gives the programmer the tools to express rather precisely how an application is to be performed. However, high-level languages are not altogether void of declarative content. For example, COBOL has a "report-writer" feature that allows the programmer to easily define the format of a printed report. Similarly, FORTRAN has a number of "standard" functions (such as a square-root calculator) which, when referenced, deliver the desired result directly.

1-1 A SELECTION OF SIX LANGUAGES

It would be impossible to present any large number of the well-known high-level languages and leave the reader with a concrete understanding of their different capabilities and relative merits. In addition, it would be irresponsible for us to present a language solely because it has "academically interesting" or "promising" features. The thought of such a presentation is fascinating enough, but it would not serve our main objective.

We have therefore made a very conservative selection of high-level languages for presentation in this text. These languages have been selected on

the basis of their impact on language development, their widespread availability, and their proven usefulness to practicing programmers. Furthermore, one reasonably sized volume could not do justice to more than six languages. Thus we have omitted certain languages that might otherwise have been included. The six high-level languages that we have chosen for presentation and discussion are ALGOL, FORTRAN, COBOL, PL/I, RPG, and SNOBOL.

ALGOL was chosen primarily for its strong influence on subsequent language development. Indeed, there would be no PL/I if ALGOL had not first introduced many fundamental principles of language design. Furthermore, ALGOL has served as the primary vehicle for language standardization and published algorithms in the journals of computing professionals. FORTRAN was chosen because it is the most widely used high-level language for scientific and engineering applications. Similarly, COBOL is included because of its dominance as the most widely used language for data processing applications.

PL/I is included for many reasons. First, it is the first general-purpose language to be implemented and widely used. Second, PL/I is a much younger language than FORTRAN, ALGOL, or COBOL. Thus its programming stylistics and built-in capabilities reflect both the experience gained from these earlier languages and the processing capabilities introduced by third-generation computers and operating systems. Third, although PL/I has not supplanted either FORTRAN or COBOL in their respective application areas, its usage has steadily increased, its implementations have become increasingly efficient, and it has recently become implemented on computers outside the IBM 360/370 family. Fourth, the adoption of a national standard for PL/I is, at this writing, imminent. For these reasons, PL/I's versatility will be examined and evaluated in this text.

RPG is included because it represents the kind of language which is often used in situations where expedient programming (i.e., a single run to produce a special report in a short amount of time) is required. It is also heavily used in small computer installations for their data processing applications.

Finally, SNOBOL is included because of its special capabilities to process text as data. It is especially effective in the area of natural-language analysis and translation. Additionally, SNOBOL is a powerful programming language for data verification and editing. By comparison, COBOL and FORTRAN are inadequate for performing such linguistic tasks; PL/I is adequate (yet less elegant than SNOBOL) in this respect. The data verification problem continues to be so important to the entire programming community that such a language as SNOBOL ought to become widely known. Until recently, SNOBOL was implemented only by an interpreter. This meant, of course, that its programs ran very slowly and could not be translated to machine code for use in a production environment. This is no longer the case. Thus SNOBOL has become an efficient alternative to other languages for text processing applications.

1-2 LEARNING THE LANGUAGES

Before one can reasonably evaluate a language, he or she must attain a working knowledge of it. Thus, our first task is to teach the main features of the six languages.

This might be an impossible task if we had to assume that the reader has no prior computing experience. This text, however, teaches the six languages in a way that exploits that experience. The basic method employed here is "learning by association," which is encouraged by the text's following organizational characteristics:

Common chapter format
Repetition of examples
Uniform metalanguage and notation

Each of the six language chapters (Chapters 2 to 7) is organized in the same way. Each has three main sections. The first section presents an introductory overview of the language, including its history, its main features, and the areas in which it has been effectively used.

The second section presents a "textbook" introduction to the main features of the language. Illustrative examples and exercises are provided so that readers may adequately test their understanding of the material presented. We do not focus our presentation on the trivial features of the language, as one might find in an introductory text. Rather, we present what we consider to be the most widely used features of the language, so that the readers' learning may be useful as well as informative.

The third section offers a deeper, more practical analysis of the language. That analysis is based on our use of the language to implement one or more "case study" problems. These case studies and their purpose are discussed more fully in section 1-3. The third section concludes with an evaluation of the language. Our basis for evaluating languages is discussed in section 1-4.

The reader will notice that a particular example will be repeated in the same place in each of the language chapters where it is appropriate. Thus, for example, when comparing two languages' subprogramming features, the reader will see those features displayed in the *same* example programming problem.

One characteristic that tends to discourage cross-referencing among languages is that each one has its own distinct metalanguage and syntax-description technique. For instance, a FORTRAN "variable name" identifies the same thing as an ALGOL "identifier" and a COBOL "data name." Similarly, ALGOL syntax is described (predominantly) in Backus-Naur form (BNF), while the syntax of COBOL is described using a more informal blend of grammar and English.

Our text tends to eliminate these hindrances. We have adopted a fairly universal collection of metawords, such as "variable," "array," "loop,"

"constant," and so forth, to describe notions with which the reader has a high level of instinctive familiarity. Some readers may not agree entirely with the word selections, but that disagreement should be greatly overshadowed by the degree of cross-referencing that the words permit. Similarly, our syntax description conventions are simple and uniformly applied throughout the six language chapters.[1]

With this level of uniformity in our presentation of the six languages, we hope that readers will have frequent occasion to cross-reference between chapters to expedite learning. As a point of departure, they might begin this study by reviewing a chapter that presents a language they already know. They can then attack an unfamiliar language with the idea of merging their own programming techniques with its facilities for expressing them, using the familiar language's chapter for reinforcement.

1-3 USING THE LANGUAGES

Programmers frequently discover a great difference between learning a language and effectively using it to accomplish actual applications. Most courses do not deal with this latter question. Typically, a short-course introduction to a language is so preoccupied with its trivial elements that the course cannot adequately teach such language elements as implementation dependent diagnostic messages, program tracing and other debugging facilities, interprogram linkage, handling of interrupts, data error recovery, and so forth.

We intend to bridge this gap. We have selected for solution six case study problems. These are not short exercises. Rather, they are chosen as representatives of the ones typically solved in day-to-day programming applications. In the third section of each language chapter, one or more case studies are implemented in that language. The implementations will provide a vehicle for discussing the practical and implementation dependent aspects of the language.

We also intend to provide detailed information on the whole question of language and compiler selection in the context of different kinds of programming tasks. To expedite this, we first characterize the following four application areas:

Scientific applications
Data processing applications
Text processing applications
Systems programming applications

[1] With the exception of RPG, which has no "syntax" in this sense.