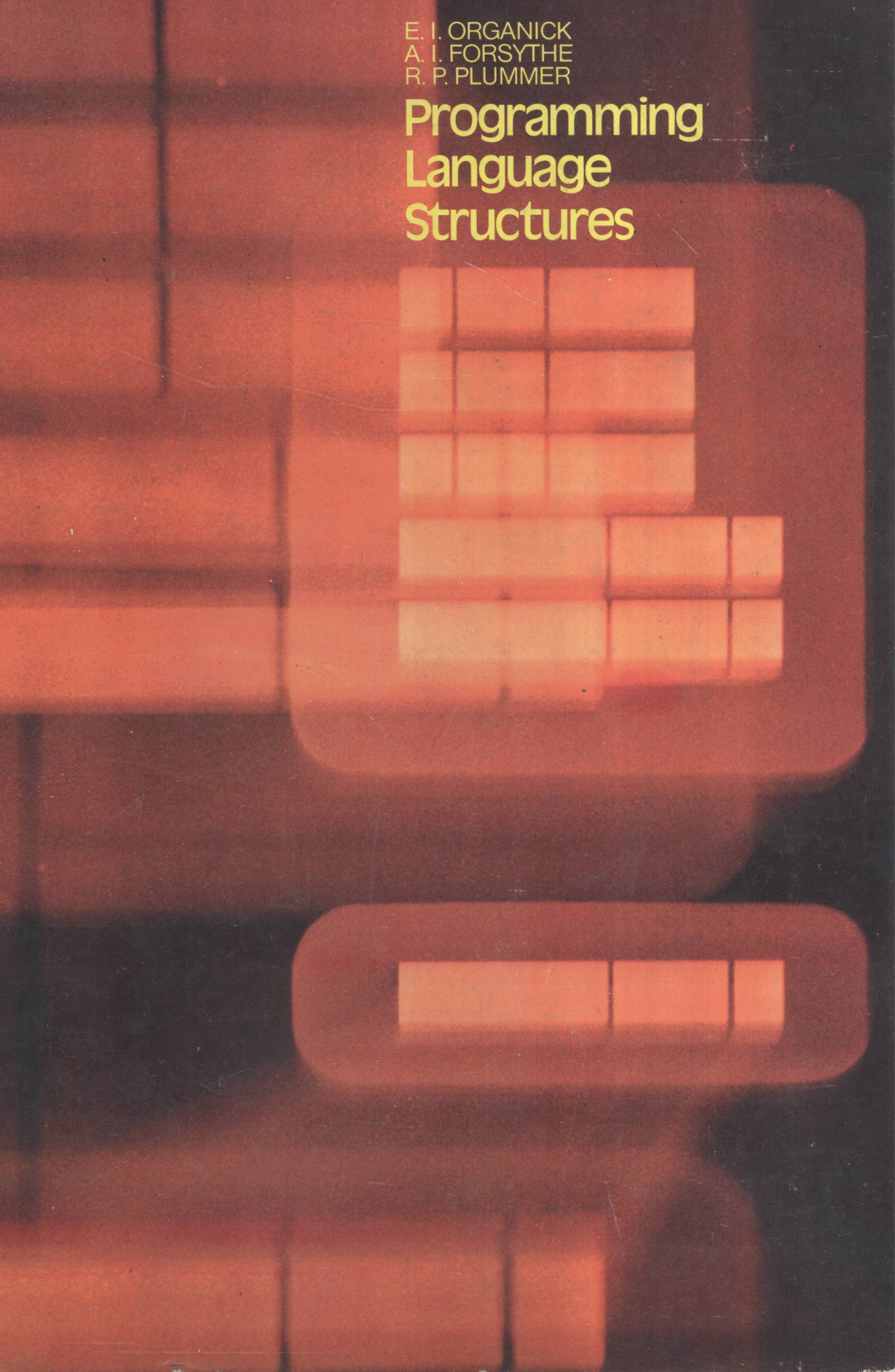
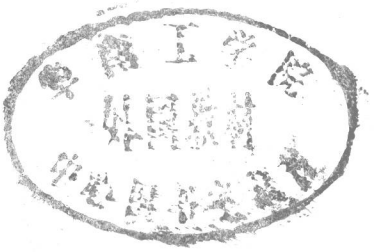
E. I. ORGANICK
A. I. FORSYTHE
R. P. PLUMMER

Programming Language Structures



PROGRAMMING LANGUAGE STRUCTURES

ELLIOTT I. ORGANICK
ALEXANDRA I. FORSYTHE
ROBERT P. PLUMMER



ACADEMIC PRESS
New York
San Francisco
London
A Subsidiary of Harcourt Brace Jovanovich, Publishers

Cover sculpture by Miriam Brofsky

Cover photo by Terry Lennon

Several exercises, a few pages of text, and some figures were reprinted or adapted, by permission, from COMPUTER SCIENCE: A FIRST COURSE (2nd edition), A. I. Forsythe, T. A. Keenan, E. I. Organick, and W. Stenberg. Copyright © 1975 by John Wiley and Sons, Inc.

ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24/28 Oval Road, London NW1

ISBN: 0-12-528260-5
Library of Congress Catalog Card Number: 78-51655

PRINTED IN THE UNITED STATES OF AMERICA

PROGRAMMING LANGUAGE STRUCTURES

To
BETTY BLANCHARD ORGANICK

for typing (and retyping) so much of our manuscript with accuracy and enthusiasm, and for her patience and cordiality during the six-year-long pursuit of our goals.

PREFACE

In their initial contact with computer programming, many students have been exposed to only one programming language. This book is designed to take such students further into the subject of programming by emphasizing the structures of programming languages. The book introduces the reader to five important programming languages, Algol, Fortran, Lisp, Snobol, and Pascal, and develops an appreciation of fundamental similarities and differences among these languages. A unifying framework is constructed that can be used to study the structure of other languages, such as Cobol, PL/I, and APL.

The book also has other objectives. For instance, it illustrates several of the tools and methodologies needed to construct large programs. Because similar interpreter structures and methods of data structuring and accessing are used to model not only programming languages but also computers, the book indirectly prepares the student to study computer organization. Working programmers whose everyday jobs confine them to a single language can use this book for self-study and gain a better perspective on the tools they now use.

By emphasizing semantics over syntax, this book differs significantly from conventional programming language texts. The *semantics* or meaning of a program can be studied by means of a snapshot sequence produced by an abstract machine-interpreter that executes the program. The semantics of a programming language can be understood (informally) by studying the semantics of a representative set of case study programs. A *snapshot* is a data and control structure that displays the state of a computation. The snapshots, or *contour diagrams*, of this book are drawn in a uniform style, clearly delineating the distinctive control structures and data structures of each language. Other diagramming conventions such as *call trees* and flowcharts are used in the exposition of semantics.

The first three chapters develop the state transition semantic framework and diagramming principles that define an abstract model of a computer. In later chapters the model is applied to describe five specific programming languages. All of these, Algol, Fortran, Lisp, Snobol, and Pascal, were developed between 1955 and 1975.

The same 20-year period saw the development of other important languages such as Cobol, APL, Basic, and PL/I, although space considerations prevent their inclusion in this book. The principles and techniques required to describe those languages, however, are similar to the ones used here for Algol, Fortran, and Lisp. Any serious reader should be able to apply the methods to study other languages and teach them to students. Thus, if an instructor wants to teach the elements of Cobol semantics using the approach of this book, he can use Fortran (Chapter 6) as a guide, since the control structures of Cobol closely match those of Fortran. In the same way, APL semantics can be explained using Lisp (Chapter 7) as a guide, since APL control structures and scoping rules are similar to those of Lisp.

The particular languages in this book were selected for comparative linguistic study because each uses a different set of rules to define the scopes of variables and procedures, and different means to treat procedure parameters. Such rules define the control structure of a language. Other language concepts and features such as pattern matching, concurrent tasks and coroutine structures, data typing, and data structure management are discussed in terms of particular languages or language extensions. These features imply variants of the abstract model of language semantics. The view is taken that one can best comprehend the semantics of a programming language, at least informally, by understanding the abstract model that enforces the control structure of the language.

More recently, other programming languages have come on the scene, such as SIMULA 67, CONCURRENT PASCAL, and Modula, and experimental versions of Alphard and CLU. Not only do these languages allow the programmer to define data types (as Snobol and Pascal do), but they also allow *abstract* data types. This facility makes possible greater clarity and modularity in programs. The snapshot diagramming techniques described in this book will, in all probability, prove as useful in explaining the semantics of the newer languages as they have been in explaining the languages in this book. Another edition or a companion volume may be required later to do justice to this claim.

A Simplifying Theme

"Computations are characterized in terms of the data structures to which they give rise during execution. . . . Programming languages may be syntactically described in terms of the data structures required for their representation and may be semantically described in terms of the data structures which they generate during execution" [Peter Wegner (1971)].

Ten years ago the proliferation of programming languages caused many people to foresee the development of a computer-age Babel where, in total ignorance of every other language, each programmer would learn only his own chosen language. That unhappy situation has not occurred for several reasons. First, effective efforts have been made to standardize particular languages such as Fortran and Cobol. It should be pointed out that pragmatic rather than scientific considerations motivated this standardization movement. However,

the second reason that Babel has been averted is that computer scientists have begun to apply the scientific method to organize the classification, comparison, and appreciation of various programming languages.

Due to the efforts of McCarthy (1962), Landin (1964), Strachey (1966), Wegner (1968), and others who provided insight into operational models of computation, we can now evaluate programming languages in terms of a unifying view of *computation structures.* Semantics and the expressive power resulting from modularity can now be studied in terms of the data structures and the accessing paths to them established during the execution of the control statements of the language.

Statements that invoke procedure entry, function entry, coroutine resumption, and block entry are intended to produce a shift of context, that is, to redefine the data structures accessible to the processor. On the other hand, procedure return, function return, and block exit are steps specifying the restoration of an earlier context. The declaration of a variable, a procedure, or a new data type is essentially a rule to allocate or restructure the workspace. These rules are clearly language-dependent, and so is the time at which such allocation and restructuring takes place. Allocation steps cause establishment of access paths and these paths may result in the sharing of data. Another kind of declaration is the specification of a parameter. This, too, can be interpreted as a space allocation rule and may result in the establishment of a new access path for the processor.

In this book we have concentrated on explaining the structures of programming languages in terms of data structures and accessibility concepts. What was previously regarded as a difficult and complex subject is now, we think, becoming a set of simple unifying ideas, concepts, and principles. The pedagogy and display methods used here derive from the pioneering work of John B. Johnston, designer of the Contour Model (1971), and others like Daniel Berry (1971) and Peter Wegner (1971), who were among the first to suggest how to apply the Contour Model to explain the semantics of a variety of programming languages. This book builds on the contributions of these individuals.

The Book in More Detail

The book may be thought of as divided into three parts. Part one, the first four chapters, introduces the basic concepts and models for understanding syntax and semantics. Part one would be a good review if the text were used in an upper division or first-year graduate course for students who come to computer science from other disciplines. Part two (next four chapters) provides the heart of the book, the intimate understanding of the comparative semantics, and secondarily the comparative syntax, of the four "mutually orthogonal" language types: Algol, Fortran, Lisp, and Snobol. Part three, the last two chapters, is intended to illustrate some of the more recent directions in language design—not a complete overview but more of a teaser to convince the student there is much more to learn. Multisequence control structures typified by asynchronous tasks

and coroutines are dealt with in Chapter 9. Data structure and data management issues are introduced in Chapter 10.

The first chapter of this text summarizes the relevant concepts and principles about algorithms, flowcharts, and computation that a student is expected to know from the first course. But that is by no means all that the first chapter accomplishes: it carefully introduces flowchart notation used consistently throughout the text (but not regarded as critical to the study) and, more importantly, introduces the abstract machine interpreter and the basic snapshot diagramming conventions. Thus although Chapter 1 might appear to be unnecessary for students whose first course has enabled them to become capable programmers in a particular language, such as Basic or Fortran, we regard the first chapter as important, especially for students not accustomed to top-down decomposition, the design of algorithms with well-documented flowcharts, or the use of an abstract interpreter for understanding the meaning of an algorithm.

Chapter 2 is an in-depth introduction to the semantics of procedure and function call, and to argument–parameter matching with various kinds of parameters. The use of contour diagrams clarifies the subtle distinctions between procedures and functions, and emphasizes the differences between globals and parameters as a means of information sharing among procedure modules.

Recursion was once regarded as a difficult concept in programming semantics, but through the medium of the contour and the call-tree diagram, recursion becomes easy to understand and to treat in some depth. In Chapter 3 recursion is looked at as a process closely related to tree traversal. In later chapters it appears again in terms of the distinctive semantics of Algol, Lisp, and Snobol. In each case *snapshot sequences* and *call trees* are used to trace what happens at key places in the execution of a recursive procedure.

Chapter 4 introduces the essential ideas of syntax formalism (for context-free languages). Backus–Naur Form and Syntax Chart representations are used.

Chapter 5 covers ALGOL 60 but the first part of the chapter deals with the general idea of block structuring. Six case study programs are presented to reinforce the reader's understanding of ALGOL 60 semantics of procedure declaration, block and procedure entry and exit, and parameter treatment.

Fortran, as covered in Chapter 6, includes not only the primary syntactic and semantic structures based on the 1966 ANSI Standard FORTRAN, but also highlights several of the innovations introduced in the new 1977 Standard. Three case study programs reinforce understanding of Fortran semantics of subprograms, parameters, and COMMON blocks. There is also a brief introduction to input–output format. The chapter ends with a general discussion comparing Fortran with ALGOL 60.

Knowing how to describe recursion effectively turns out to be a key to understanding the semantics of an expression-based (functional) language like Lisp. Lisp recursion presents no high hurdle for a student who has already studied recursion in Chapter 3 and ALGOL 60 recursion in Chapter 5. To introduce Lisp in Chapter 7 it is sufficient to emphasize its data structures rather than its control

structures. Thus Lisp is presented as a language to manipulate binary tree data rather than a language to process lists.

We deemphasize the unfriendly syntax of McCarthy's LISP 1.5 by first presenting a functional subset of RLISP, A. C. Hearn's (1968) version of Lisp having an Algol-like syntax. (RLISP is available on a variety of computers.) In our threefold attack on the problem of teaching Lisp, control structures are displayed by contour diagrams and call trees; evolving data structures are drawn as binary trees rather than as S-expressions, and the easy-to-learn syntax of RLISP is used. Conventional Lisp syntax is also presented.

Chapter 8 is a survey of SNOBOL 4 emphasizing the pattern matching features, the data management and procedure semantics, and the capability of defining and using new data types. Juxtaposing the chapters on Snobol and Lisp offers the student a good opportunity to compare and contrast the simplicity of Lisp with the expressive power of Snobol. Even without full understanding, one can become aware of some of the trade-offs between these factors.

Chapter 9 is an introduction to the concepts of multitasking and coroutine structures using Burroughs EXTENDED ALGOL as the language vehicle for the case studies. Contour diagrams are again very useful to clarify the semantics of these new features.

Both Chapters 9 and 10 assume the reader has already studied ALGOL 60 (Chapter 5). Chapter 10, therefore, introduces Pascal as a departure from Algol, enabling the reader to quickly encounter Pascal's important (new) data structure definition facilities which are the focus of attention here. A major worked example coded in the dialect SEQUENTIAL PASCAL is discussed and displayed at the end of the chapter.

Place of This Text in the Computer Science and Engineering Curriculum

This text has grown out of our experience in teaching two types of courses at the University of Utah: (1) the second of a four-quarter undergraduate sequence for computer science majors; and (2) a senior/graduate course for nonmajors whose only previous exposure to computer science is a freshman-level course in programming.

The first quarter of the four-quarter sequence emphasizes algorithms, flowcharts, and top-down problem solving, but also includes elementary concepts of procedures such as might be covered in a beginning Fortran, Basic, or Cobol programming course. An elementary but extendable model of computation is used. This model is comparable to the one given in Chapter 1 of this book.

The second quarter, represented by this book, provides the informal introduction to the linguistics of programming. Deferred until later quarters are the formal methods for defining the syntax and semantics of programming languages, syntax analysis, and semantics of abstract machine interpreters.

At Utah most of this text is covered in ten weeks. In addition, students carry out computer laboratory exercises and experiments in two or three languages. We want students to practice with important, unfamiliar languages; at Utah this

means Algol and Lisp. At other schools Chapters 6 through 10 could be used as guides to help instructors prepare supplemental class notes in other languages or dialects locally available.

Experience has convinced us that the first set of laboratory exercises should be designed to provide practice with parameter passing and simple recursion. Later exercises can confirm an understanding of specific language features or build more comprehensive programs possibly involving serious use of recursion. Since students need not be encouraged to become expert programmers in any one language, they may be able to complete up to six such exercises in ten weeks.

An instructor who uses this text in a full semester course of fifteen weeks has the pleasant option of treating the subject either in greater depth or greater breadth. For instance, a comprehensive programming project may be assigned to give students in-depth acquaintance with one of the languages covered in this text. Or, one or more other languages may be surveyed. The student may find it a stimulating challenge to apply the structuring concepts of this book to the understanding of a rich but unfamiliar language, such as Cobol, APL, Simula, or Modula.

Acknowledgments

We were unusually fortunate to receive a wide range of valuable and candid criticism from some 15 specialists who read parts or all of our manuscript. Many were anonymous referees, and we thank them sincerely. Others included experts whose help we sought directly. In particular, we gratefully acknowledge the suggestions offered by Allen Ambler, Daniel Berry, Victor Basili, Daniel Friedman, Narain Gehani, Robert Graham, and David Hanson. Daniel Berry's help was invaluable. He offered useful comments on practically every page of two consecutive "final drafts."

We did our best to respond to as many major and minor suggestions as possible. Inevitably, some of the suggestions seemed to us mutually exclusive or implied extending the content of the book or raising its level beyond what we felt was practical. We sincerely hope all the reviewers are pleased with the final product.

We also received considerable help from student assistants at the University of Utah during the years in which the manuscript evolved from crude "handouts" to almost sensible course notes. These helpful and patient people included D. Bourek, M. S. Dye, R. A. Frank, and J. W. Thomas. The support of our other colleagues and students at Utah and the typists, especially Karen Evans and Carol Brown, is gratefully acknowledged as well. We also thank the editorial and production staffs of Academic Press who appreciated our teaching goals and cooperated with us fully to achieve the graphic styling and clarity we believed essential.

E. I. Organick
A. I. Forsythe
R. P. Plummer

CONTENTS

PART ONE

PART ONE