PRINCIPLES OF PROGRAMMING LANGUAGES

R. D. TENNENT

73.8722/
T297

PRINCIPLES OF PROGRAMMING LANGUAGES

R. D. TENNENT
Department of Computing and Information Science
Queen's University, Kingston, Canada

Prentice/Hall International

5506325

ENGLEWOOD CLIFFS, NEW JERSEY LONDON NEW DELHI
SINGAPORE SYDNEY TOKYO TORONTO WELLINGTON

Library of Congress Cataloging in Publication Data

TENNENT, R D 1944–
Principles of programming languages.

Bibliography: p.
Includes index
1. Programming language (Electronic computers)
I. Title.
QA76.7.T47 001.64'24 80-24271
ISBN 0-13-709873-1

DS71/01

British Library Cataloguing in Publication Data

TENNENT, R D
Principles of programming languages.

1. Programming languages (Electronic computers)
2. Electronic digital computers—Programming
I. Title
001.64'24 QA76.7
ISBN 0-13-709873-1

ISBN 0-13-709873-1

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY., LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE., LTD., *Singapore*
PRENTICE-HALL, INC., *Englewood Cliffs, New Jersey*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

PREFACE

This book is a systematic exposition of the fundamental concepts and general principles underlying programming languages in current use. It may be used as a text for courses in computing science and software engineering programs, and as a reference by advanced programmers, programming theorists, and programming language implementers, describers and designers. Linguists and logicians may also be interested to see how the methods of mathematical logic may be applied to formal languages that are much more complex than the traditional logical calculi.

The material and the presentation have been strongly influenced by the approach to programming language theory founded by Dana Scott and the late Christopher Strachey at Oxford University, particularly the first chapter of *A Theory of Programming Language Semantics* by Robert Milne and Strachey (Chapman and Hall, London, and Wiley, New York). But I have emphasized intuitive concepts, rather than formalism and mathematical theory. I hope that this will help to make their work accessible to a wider audience.

Readers are expected to have enough programming experience to appreciate the basic ideas of programming methodology (importance of program correctness, readability and modularity, as well as efficiency; separation of levels of abstraction; stepwise refinement), and to have a reading knowledge of PASCAL, which is used as a standard example throughout. There are also "case studies" of interesting aspects of several other languages used in practice, but no attempt is made to give complete descriptions of languages, or to discuss experimental languages. The emphasis is on significant differences and similarities between linguistic concepts. The only mathematical prerequisite is a basic knowledge of sets and functions.

Undergraduates with adequate programming experience and mathematical maturity can cover all the material in the order presented in two terms. For students with weaker backgrounds, the "starred" sections (on the principles underlying Scott's theory of computation) may be omitted. It is also possible to use the final chapter as the outline of an introductory graduate course in formal description of programming languages, referring to material in earlier chapters as needed.

There are exercises, project suggestions and an annotated bibliography at the end of almost every chapter. An additional bibliography of suggested readings on each of the programming languages mentioned in the text is given in an appendix.

I am very grateful to everyone who gave me suggestions and comments on various drafts, particularly Michael Gordon, Robert Milne, Tony Hoare, David Barnard, Mike Jenkins, Molly Higginson, David Leeson, Bill O'Farrell, John Gauch and Bruce Stratton. The remaining errors, obscurities and prejudices are my responsibility. I would also like to thank Michael Levison for his help in preparing the manuscript with his IVI text-editing system and the Natural Sciences and Engineering Research Council of Canada for financial assistance.

R. D. T.

CONTENTS

152-41

B1

11 CONCURRENT PROCESSES 165

12 TYPES 179

1 INTRODUCTION

1.1 PROGRAMMING LANGUAGES

A programming language is a system of notation for describing computations. A useful programming language must therefore be suited both for *describing* (i.e., for human writers and readers of programs), and for *computation* (i.e., for efficient implementation on computers). But human beings and computers are so different that it is difficult to find notational devices that are well suited to the capabilities of both. Languages that favor humans are termed *high-level*, and those oriented to machines *low-level.*

Let us consider some extreme examples of programming languages. In principle, the most "powerful" language for any computer is its machine language, which provides direct access to all of the resources of that computer. However, programs in such a language cannot conveniently be implemented on *other* computers. Furthermore, it is very difficult to write or read machine-language programs. Human beings cannot cope with the complete lack of structure in both programs (sequences of machine instructions) and data representations (sequences of machine words).

It might be thought that "natural" languages (such as English and French) would be at the other extreme. But, in most fields of science and technology, the formalized symbolic notations of mathematics and logic have proved to be indispensable for precise formulation of concepts and principles and for effective reasoning. However, in their full generality the notational devices of mathematics are not even implementable on computers, for deep reasons that will be discussed later.

There is a language called LAMBDA (invented by D. Scott) that has many of the properties of conventional mathematical notations and is as expressive as possible: *all* and *only* the operations that apparently are possible to compute are definable in LAMBDA. These properties make it useful as a specification language and in theoretical studies of computability.

But LAMBDA is so far removed from conventional computers that, though implementable in principle, it would not be practical as a *programming* language.

In short, an ideal programming language would combine the advantages of machine languages and mathematical notations, but achieving this aim has proved to be a very difficult problem. Many existing languages have only managed to combine countless "features" into a jumble that is neither easy to implement nor a pleasure to use.

There are so many programming languages and most are so complex and irregular that it would be nearly impossible and certainly pointless to learn *every* feature of *every* existing programming language (or even of the dozen or so more important ones). Fortunately, there is a great deal of *conceptual* overlap between programming languages, even those that on the surface appear to be quite dissimilar. Almost every practical programming language has mechanisms for dynamically updating storage, introducing symbolic names, transferring control, structuring data, defining procedures, and so on. In every language, these mechanisms are governed by the same general principles.

It is on these fundamental concepts and general principles that this book concentrates. Understanding them will make it easier to use, describe, compare, implement, and design programming languages.

It will be convenient to use a single programming language as a standard example in this book. PASCAL has been chosen because it is widely known and has been one of the most successful at reconciling conflicting design criteria (though it is certainly not the final step in the evolution of programming languages!) The reader is assumed to have a reading knowledge of PASCAL as well as experience in programming with some high-level language. Jensen and Wirth (1974) or a comparable description should be available for reference. Minor variants or extensions of PASCAL will be described and discussed when convenient or necessary to illustrate a point. Several case studies of other well-known languages will provide a broader perspective. Appendix A is a bibliography of suggested readings for each of the programming languages discussed. It should be noted that many of the program fragments used as examples are intended only to illustrate language concepts and do not necessarily exemplify good programming style.

1.2 SYNTAX, SEMANTICS AND PRAGMATICS

It is traditional when dealing with languages of all sorts to try to separate concerns with form, the subject of ***syntax****, from concerns with meaning, the

*Important technical terms are introduced in bold italic face.

field of ***semantics***. Consider the simple "language" of ***binary numerals***. Some examples of binary numerals are

```
       0
       1
     101
    0101
10011010
```

A communication in this language evidently consists of a finite sequence of characters '0' and '1'. This is just syntax however, and says nothing about what such a communication is intended to *mean*.

The usual interpretation for such numerals is that each numeral denotes a *natural number* (i.e., zero or one of its successors). For example, '101' and '0101' both denote the number five, the fifth successor of zero. Numbers are "abstract" mathematical concepts, whereas the digit strings that appear on paper are numerals, that is to say, symbolic representations or descriptions of numbers. Many other languages have this same set of numbers as their meanings: decimal numerals, Roman numerals, and so on.

In general, then, *syntax* is concerned with only the format, well-formedness, and compositional structure of communications in a language, and *semantics* with their meaning.

The ***pragmatics*** of languages have to do with their origins, uses, and effects. So, the pragmatic aspects of programming languages include language implementation techniques, programming methodology, and the history of programming-language development. In this book, important pragmatic considerations will be pointed out wherever appropriate, but systematic expositions of programming methodology, language implementation, and history are outside its scope.

The criterion for correctness of a language processor is that it implement the syntax and semantics of the language. However, because of pragmatic factors, processors often do not meet their specifications for all possible programs and data. For example, suppose that the language of binary numerals were to be "implemented" by representing numbers in a storage register of fixed size. It is evidently impossible for every numeral in the language to be correctly implemented as specified.

If a processor is unable to meet its specifications for some input, it should signal this with an appropriate warning message. Otherwise, it is termed ***insecure***. Output from an insecure processor must be treated with suspicion unless it can be verified that the program has not breached any of the insecurities.

An important goal of programming language design is to make it easier for implementers to eliminate insecurities without incurring severe penalties in execution time or storage space. Unfortunately, with most current computer designs, some kinds of programming error cannot be detected economically, so that the goal of eliminating insecurities should also be taken up by computer designers.

1.3 SYNTAX-DIRECTED SEMANTICS

Programmers are encouraged to program in a "structured" way, that is to say, to use the *syntactic* structures of their programming language to help them systematically develop and more clearly express the *semantic* structure of their algorithms. Similarly, languages are best ***described*** by basing specifications of their semantics on an appropriate syntactic description. Programming languages are so complex that a structured approach is almost essential for conceptual understanding.

As a simple example of syntax-directed semantic description, consider again the language of binary numerals. The syntax of this language may be precisely specified as follows:

(a) Characters '0' and '1' are binary numerals.
(b) If N is a binary numeral, then N with a '0' or a '1' appended to the right of it is also a binary numeral.
(c) These are the only binary numerals.

Rule (a) describes the two *elementary* (i.e., nondecomposable) syntactic forms. Rule (b) describes the two *composite* forms; in this rule, the binary numeral N referred to is an example of what is termed an ***immediate constituent*** (of a composite syntactic form). Rule (c) specifies that the set of binary numerals is to be the *smallest* set meeting requirements (a) and (b).

Note that this syntactic description specifies not only the criteria for well-formedness of a binary numeral, but also its ***phrase structure***, that is to say, how it is analyzed into immediate constituents, and these into their

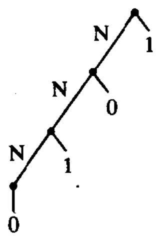

Fig. 1.1

immediate constituents, and so on, until elementary forms are reached. For example, the phrase structure of binary numeral '0101' may be depicted by the tree shown in Fig. 1.1. The process of determining the phrase structure of text is known as ***parsing***.

A specification of the meaning of (i.e., the number denoted by) every binary numeral may now be based on the above syntactic description as follows:

(a) Binary numerals '0' and '1' denote numbers zero and one, respectively.
(b) If N is a binary numeral that denotes number n, then (i) N with '0' appended to the right of it denotes number $2\times n$, and (ii) N with '1' appended to the right of it denotes number $2\times n+1$.

For example, consider numeral '0101'. Working from the leftmost character,

'0'	denotes zero, using rule (a);
hence, '01'	denotes $2\times0+1=1$, using rule (b), part (ii);
hence, '010'	denotes $2\times1=2$, using rule (b), part (i);
hence, '0101'	denotes $2\times2+1=5$, using rule (b), part (ii).

Each non-terminal node of the phrase structure tree for '0101' may be "labelled" with the semantic object denoted by the corresponding phrase (Fig. 1.2). Thus semantics chases denotation up the syntax tree (with apologies to W. V. Quine).

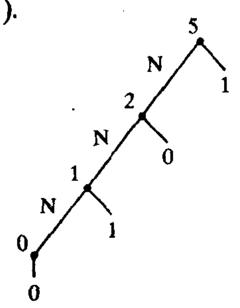

Fig. 1.2

The above description of the syntax and semantics of binary numerals is an example of what is known as the ***denotational*** approach to language description. The general idea is simply to specify the meanings of (i.e., the semantic objects denoted by) elementary forms directly, and the meanings of composites in terms of the meanings of their immediate constituents. This "structured" approach has a long history in logic and linguistics. Subsequent chapters will explain how *programming* languages may be described denotationally.

EXERCISES

1.1 Suggest two "unusual" semantic interpretations for binary numerals.

1.2 Suppose that rule (b) of the definition of the syntax of binary numerals were changed to
(b) If N is a binary numeral, then N *prefixed* by a '0' or a '1' is also a binary numeral.
Define the usual semantics of binary numerals using this syntactic description.

1.3 Describe the syntax and usual semantics of binary numerals with fractions, such as '101.0101'.

*1.4 Prove that, according to the syntax and semantics given, *every* finite binary numeral has a *unique* meaning, using mathematical induction on the length of the numerals.

*Solutions to starred exercises require a higher level of mathematical maturity.

PROJECT

Write an essay on the history of one of the major programming languages.

BIBLIOGRAPHIC NOTES

There is a large literature on programming language design. Three papers by Hoare [1.6, 1.7, 1.9] are especially recommended. The language LAMBDA was described by Scott [1.15].

The trichotomy between syntax, semantics, and pragmatics was proposed by Morris [1.12, 1.13] and Carnap [1.2]. The history of programming languages is discussed in papers by Knuth and Pardo [1.10] and Hoare [1.8], in a book by Sammet [1.14], and in a conference proceedings [1.18]. There are large literatures on programming methodology and language implementation; see, for example, collections edited by Gries [1.5], and Bauer and Eickel [1.1], respectively.

The denotational approach to language description may be traced back to Frege [1.4], Carnap [1.3], and Tarski [1.17]. Its use for formal description of programming languages was developed by Scott and Strachey [1.16]. Montague [1.11] gave a denotational description of a fragment of a natural language.

1.1 Bauer, F. L. and J. Eickel (eds.). *Compiler Construction, An Advanced Course*, Springer, Berlin (2nd edition, 1976).

1.2 Carnap, R. *Introduction to Semantics*, Harvard University Press, Cambridge (1942).