Roland C. Backhouse

Syntax of Programming Languages

Theory and Practice

PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE

C.A.R. HOARE SERIES EDITOR

SYNTAX OF PROGRAMMING LANGUAGES
Theory and Practice

ROLAND C. BACKHOUSE
Heriot-Watt University
Edinburgh, Scotland

Prentice/Hall PHI International

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

CONTENTS

PREFACE

The analysis of the syntax of programming languages is a topic which, deservedly, occupies a prominent position in computer-science curricula because of its relevance both to the practice of compiler construction and to the theory of computing. It is this bridge between theory and practice which I have chosen to stress.

The book is primarily directed towards computer-science students in the third or final year of an undergraduate degree course. It is assumed that the reader is familiar with the standard mathematical notation for sets and with the mathematical concept of proof, in particular proof by induction. The reader should have attended a course on the design of algorithms and data structures, preferably one in which the use of loop invariants to provide correctness proofs is an integral part. It is also preferable if the reader is familiar with PASCAL. However, I have always made a clear distinction between algorithms and programs so that the former can be understood without reference to any specific programming language.

Chapter 1 (Fundamentals) begins by introducing the principal concepts and terminology associated with context-free grammars, illustrating them with examples taken from the "Revised Report on ALGOL 60" as well as abstract examples. This is followed by a discussion of graph searching and its application to eliminating useless productions from a given context-free grammar. The material on graph searching is included in the "Fundamentals" because it is used extensively throughout the remainder of the book.

Chapter 2 discusses the variety of ways in which regular languages may be defined. The motivation for this chapter is the relationship between extended BNF and regular expressions (discussed in Chapter 3) and the use of a deterministic finite-state machine as the "heart" of an LR parser (Chapter 4). The material is standard but some of the presentation is novel—in particular, when designing an algorithm to convert a non-deterministic machine to a deterministic machine, I have taken care to stress the relationship to graph searching.

The emphasis in Chapter 3 is on the correspondence between parsing by recursive descent and the theory of LL parsing. The chapter begins by presenting a basic scheme for constructing a recursive-descent parser for a strong LL(k) grammar. This is followed by an algorithm for testing the strong LL(k) property, which is then specialized to the case $k=1$. The chapter is concluded by a discussion of the relationship between the LL(k) and strong LL(k) properties, and of practical problems arising from the use of recursive descent.

The treatment of LR parsing in Chapter 4 is entirely non-standard and is notably shorter than Chapter 3. This is because, in my view, a study of recursive-descent parsers is much more fruitful for the non-specialist. I have, therefore, stressed the principles underlying an LR parser rather than the practical details of its construction.

Chapters 5 and 6 are an in-depth study of the theory of error repair and its application to the design of the error recovery in a recursive-descent syntax analyzer. Error recovery is, of course, an extremely important part of syntax analysis and a theory which stops short of this topic does not deserve to be called practical. The theory developed in Chapter 5 is based on the work of M. J. Fischer and R. A. Wagner (Sec. 5.1, string-to-string repair), R. A. Wagner (Sec. 5.2, regular language repair) and A. V. Aho and T. G. Peterson (Sec. 5.3, repair of context-free languages); my own contribution has been to unify the treatment of the three topics. I have preferred the term "repair" to the more prevalent "correction" because the former is less emotive and, in my view, more accurate. In Chapter 6 error recovery is seen as a compromise between efficiency constraints and the ideal of "least-cost" repair. Apart from a discussion of Wirth's recovery technique, which is included to motivate the ensuing development, the material in this chapter is entirely original.

Chapter 7 concludes the book by exposing the limitations of context-free grammars and encouraging the reader to a further study of the semantics of programming languages.

A feature of the text is the inclusion of an almost complete LL(1) test, implemented in PASCAL. This program, which is begun in Chapter 1 and extended in Chapter 3, has been included for two reasons. Firstly, I suspect that many students remain sceptical of the practicality of algorithms if they do not see them implemented. Secondly, and more importantly, I wanted to provide a basis for intellectually demanding but realistic projects which could be used to reinforce the material in the text. Project work is, quite rightly, a significant element in the training of a computer scientist, but too often its educational value is nullified by the sheer quantity of coding required, most of which is rather trivial in nature. The LL(1) test is, therefore, intentionally incomplete and the exercises request the reader to supply the missing procedures—between 50 and 100 additional lines of coding being required. As an aid to institutions recommending this text to their students I am willing to supply a copy of all programs and associated test data which were written during its preparation. Anyone interested should write to me at:

Department of Computer Science, Heriot-Watt University,
79 Grassmarket, Edinburgh, EH1 2HJ, Scotland.

A number of people have helped me to write this book, but first and foremost my thanks go to my wife, Hilary. The reviewers, Professor A. V. Aho, Dr. D. J. Cooke and Mrs. J. Hughes, have done an admirable job helping me to avoid many errors and offering a number of incisive comments. Thanks also go to Stuart Anderson, Simon Kestner, Nick Kotarski and Janet Meynell for their help in debugging the text. The PASCAL compiler used to debug the programs was written by Dag Langhmyr; without his efforts this book could not have been written. The assistance of Dave Cooper, John Fisher and Mike Staker has also been invaluable in ensuring that the PASCAL system was truly reliable. Dr. J. Welsh and Professor A. H. J. Sale have given advice on the technicalities of PASCAL for which I am very grateful, and a number of students have helped me by undertaking projects related to the material. I should like to mention particularly Jim Farquhar, Bryan MacGregor and Callum Mills. I would also like to thank Isabel Warrack for the excellent job she did of typing the manuscript, and my home department for the facilities it has made available to me. Finally, thanks go to Henry Hirschberg and Ron Decent of Prentice-Hall International and to Tony Hoare for providing encouragement at the opportune time.

ROLAND C. BACKHOUSE

February, 1979

GLOSSARY OF SYMBOLS

$\in$	belongs to, is an element of
$\notin$	does not belong to
$\varnothing$	empty set
$\cup$	set union
$\cap$	set intersection
$\supseteq$	contains, is a superset of
$\subseteq$	is a subset of
$\sim$	not
$\wedge$	and
$\vee$	or
$\supset$	implies
iff	if and only if
$\forall$	for all
$\exists$	there exists
$\square$	end of a proof
$\rightarrow$	replacement symbol (productions)
$::=$	replacement symbol in ALGOL 60 Report
$\Rightarrow$	replacement symbol (derivation sequences)
$\not\Rightarrow$	does not generate
$\Rightarrow_l$	generates leftmost
$\Rightarrow_r$	generates rightmost
$\rightarrow$	replacement symbol (repairs)
$\longrightarrow$	replacement symbol (function definitions)
Λ	empty word
$\mid$	alternate sign in productions
*	reflexive and transitive closure, iteration zero or more times
$^+$	transitive closure, iteration one or more times
O	order

SYNTAX OF PROGRAMMING LANGUAGES
Theory and Practice

1 FUNDAMENTALS

ALGOL 60 has lost a great deal of its popularity as a programming tool. It never succeeded in superseding FORTRAN, and has itself been superseded by programming languages like PASCAL and ALGOL 68. Nevertheless, I would venture to say that the contribution of ALGOL 60 to computing science will far outlive the contributions of PASCAL, ALGOL 68, FORTRAN or any other programming language in existence today. The main contribution of ALGOL 60 to computing is the standard it set for the *definition* of programming languages. The ALGOL 60 report introduced a method of defining the syntax of programming languages which has, subsequently, been used to define almost all programming languages of any merit. The method goes under a number of different names: *Backus Normal Form*, after its inventor J. Backus; *Backus–Naur Form* (BNF), after Backus and the editor of the report, P. Naur; and *context-free grammars*, the name suggested by N. Chomsky who independently (and prior to Backus) applied the method to defining the syntax of natural languages.

The definition of a programming language is normally split into two parts—the *syntax* and the *semantics* of the language. The syntax of the language specifies those combinations of symbols which are in the language. For example

$$\textbf{begin real } x;\ x := 0 \textbf{ end}$$

is an ALGOL 60 program, but

$$\textbf{begin end } x \textbf{ real};\ x := 0$$

is not. The semantics of a language specifies the "meaning" of syntactically correct constructs in the language. In programming-language terms the semantics specifies, for each program in the language, an *input–output* *relation*; that is, the output of the program given a particular input. The ALGOL 60 report introduced a *formal* method of defining the syntax, but defined the semantics *informally* in English. Subsequently the theory of the syntax of programming languages, or *language theory* as it is commonly called, developed rapidly and has now, more or less, stabilized. In contrast, the theory of the semantics of programming languages has been slow to develop and has not yet stablized. The practical significance of this disparity is that the construction of the syntax-analysis phase of a compiler is now a highly reliable process tackled with ease by compiler writers; on the other hand the incorporation of code generation is much more difficult and often subject to debate.

This book introduces the reader to context-free grammars and presents a sample of the many theoretical results which relate to their use in defining programming languages. The main objectives of the book are to examine critically the value of context-free grammars as a definitional tool and, as a by-product, to convince the reader of the need for formal definitions and a sound *theory* of programming-language definition. We shall use the "Revised Report on the Algorithmic Language—ALGOL 60" [1.12] as the primary source of illustrative examples. This report, whilst now quite old in comparison to most computing-science literature (it was published in 1963), is one of the few classics and should be obligatory reading for all computing-science students. We would advise the reader to have a copy on hand when reading this text. (A list of the journals in which it has been published appears in the bibliography at the end of this chapter.) In subsequent sections we shall refer to the "Revised Report on ALGOL 60" or, simply, the "Revised Report" rather than give it its full title.

How are we to evaluate the definition of a programming language? We can obtain a number of criteria by looking at various methods of definition in common use and examining their inadequacies.

The method by which we all learnt our first programming language will undoubtedly have been by example. This is an excellent method but, unfortunately, suffers from the drawback that it can never be complete, since we can only ever see a finite number of examples. Everyone must, at some time, have asked the question "am I allowed to . . . ?". One should not blame oneself for not knowing the answer, nor should one blame one's teacher for not having already given the information—the method of definition is at fault. Thus our first criterion is:

C1. *The definition should provide a complete description of all aspects of the language.*

Examples are normally supplemented by verbal descriptions in order to rectify the last criticism. However English, like all natural languages, is full of ambiguities. In programming it is extremely important that we be quite unambiguous in everything we do. It is very important, therefore, that any language we use is also unambiguous or, if it is ambiguous, that such ambiguities be easily avoided.

There are two issues involved here and so we shall introduce some terminology to clarify them. When we define ALGOL 60 in English, there are two languages involved—ALGOL 60, the language being defined, which is called the *object language*, and English, the language used to make the definition, which is called the *metalanguage*. Now, it is a requirement of the word "definition" that the metalanguage be completely unambiguous, and this is the paramount motivation for a formalized or mathematical definition. However, in computing, we also impose the following requirement.

C2. *Any ambiguities in the object language should be intentional and immediately clear from the definition.*

In the event of a misunderstanding of a programming-language feature the compiler of the language is often referred to. A compiler does in fact define a language precisely and unambiguously, but it is perhaps the worst method of definition for a number of reasons. Firstly, there is no compiler to which the very first implementor of a language can refer. Secondly, there is no guarantee that two compilers of the same language agree. Finally, the compiler introduces a very large amount of machine-dependent detail which is irrelevant and confusing to the user. In summary, therefore:

C3. *The metalanguage should be easily understood and machine-independent.*

There are two parties interested in the definition of a programming language —the user and the compiler writer. Sometimes the requirements of these two parties are conflicting and there is growing evidence that, with regard to the semantics of a language, more than one mode of definition should be used. However there is one respect in which the compiler writer and user have a very strong mutual interest—the reliability of the compiler. Our final criterion is thus:

C4. *The method of definition should facilitate the systematic construction of a compiler.*

This book is based around a critical examination of the "Revised Report on ALGOL 60" with respect to these four criteria. The most important criterion is C4 and the success of the BNF definition of ALGOL 60 is very much due to the ease with which reliable syntax analyzers can now be written. For this reason discussion of techniques to facilitate the systematic construction of syntax analyzers occupies the major part of the text. Chapters 2–6 are all primarily concerned with this topic. Chapters 3 and 4 describe the two most important techniques used in syntax analyzers. The reader should not be put off by their names ("LL parsing" and "LR parsing") which are singularly meaningless and uninspiring. LL parsing is sometimes called parsing by "recursive descent" and is a very natural and easy technique to use. LR parsing is less natural but more powerful and is the technique normally used in so-called "translator writing systems" or "compiler-compilers" i.e. computer programs which input the definition of a language and output a compiler for that language. Chapter 2 includes preparatory material which is necessary to understand parts of all the remaining chapters. Error analysis is an extremely important part of syntax analysis and Chaps. 5 and 6 attempt to give it the thorough treatment it deserves. Chapter 5 develops a mathematical model of error repair which is then applied in Chap. 6 to the design of an error-recovery scheme.

Chapter 7 examines the "Revised Report on ALGOL 60" with respect to criterion C1. Surprisingly the context-free definition of ALGOL 60 does *not* completely define the syntax of ALGOL 60. We have learned to live with the limitations of the Report, but it is important to know exactly what they are! Chapter 7 is concluded by a discussion of some of the problems which have been caused by the lack of a formal method for completely defining the syntax and semantics of programming languages. It is hoped that the reader will then be motivated to go beyond this text and consult some of the references quoted in the bibliography of Chap. 7.

This first chapter includes in Sec. 1.1 the basic definitions and notation which are used throughout the text. Criterion C2 is then examined in Sec. 1.2. The word "ambiguous" is, of course, itself ambiguous. However in Sec. 1.2 we show how we can precisely characterize "ambiguous" context-free grammars. A test for ambiguity is delayed (for reasons which will become evident) until Chap. 3. The semantic ambiguities in the "Revised Report on ALGOL 60" are discussed in the final chapter.

A major emphasis in this book is on the design of efficient algorithms. The author agrees with the view of D. E. Knuth ("Computer Science and its Relation to Mathematics", *Computers and People*, September 1974, 8–11) that "computer science . . . is the study of algorithms". To set the tone of the remainder of the book a discussion of graph searching and its application to finding useless productions form the final two sections of this chapter. Although the connection of graph searching to language theory is not immediately apparent, it will repay the reader to study Sec. 1.3 thoroughly. It is referred to again and again in subsequent chapters and hence is one of the most important in the book.