In spite of major advances in the theoretical foundations of programming language semantics over the past thirty years, most working programmers no more than a passing acquaintance with the tools that have been developed for language specification. This is in part due to a belief that these tools rely on abstract and unrealistic views of program behavior, have forbidding syntax, and generate descriptions that are immensely complex and difficult to implement. A compiler writer would rather rely on an intuitive understanding of a language than translate complex and abstruse descriptions of its semantics into a practical implementation.

This reluctance to put formal semantics to practical use is mirrored in the current offerings of texts and university courses. On the one hand, programming language surveys and programming texts depend almost exclusively on informal descriptions of language constructs. On the other hand, texts in the area of programming language semantics tend to focus on the theory of the descriptive tools themselves, rather than using them as tools to elucidate the workings of language constructs.

Some may argue that semantic tools have not achieved a sufficient level of completeness, simplicity, and power to be universally accepted as practical descriptive tools. Without entering into a debate on this subject, I would argue that for many the perception of formal semantics as obscure and difficult to understand is just as great a barrier to its acceptance. In fact, I believe just the opposite is true. While a certain investment of time is required to become comfortable with the notation, this should present no greater challenge than any other technicalities that a student of computer science is required to master. Once the semantics tools have been mastered, they provide a clear and unified map of the programming language landscape. The connections and contrasts between different language constructs are made clearer, and are more elegantly expressed by the formalism, than is possible with an English language description.

The goal of this text is to introduce some of the tools of semantic description, and show how these can clarify issues that underly the design of many languages. The book bridges the gap between the introductory surveys of programming languages and the more advanced presentations of formal semantics referred to above. It brings the tools of formal semantics,
specifically denotational semantics, to bear on a discussion of the structures common to most programming languages. While we do provide some proofs in a later chapter, the text emphasizes the use of denotational semantics as a tool and passes lightly over those issues that require formal proof, e.g., properties of domains, existence of fixed points, etc. In this, the approach is similar to that of a calculus text which skips proofs of the existence of limits and derivatives in favor of applications of those concepts.

One effect of basing the presentation so completely on formal semantic specification is that the syntactic style of a language tends to fall away and become relatively insignificant. At the same time, semantic concepts that transcend the particular style of individual languages begin to take on the role of guiding principles. This kind of unification is helpful in trying to comprehend the Tower of Babel of modern programming languages.

Denotational semantics has a reputation for complexity and abstruseness. This arises in part from its historical foundation in lambda calculus. Lambda calculus, itself, is simple and elegant, but it is on the level of machine language in the minuteness of its focus. Descriptions written in lambda calculus are no more palatable than programs written in machine code.

The text uses high-level functional descriptions, with the implicit assumption that these can be reduced to lambda calculus descriptions if required. Furthermore, we borrow an idea from Peter Lee [Realistic Compiler Generation, MIT Press, 1989] which he calls high-level semantics. Language definitions are given in terms of language based concepts such as state and environment. These concepts are then separately specified as semantic domains with certain allowable actions. In a rather loose analogy with object technology, we call these semantic objects. For example, State is the domain of all memory states, with actions State.Put and State.Get that modify and interrogate the current state. It should be emphasized that these semantic objects are simplifications of low-level lambda calculus descriptions and are not realizations of abstract data types; the axiomatic definition of data types is a subject beyond the focus of this text. The operations that are applied to these semantic objects to produce language descriptions are simple functional operations such as composition and argument substitution.

This style gives rise to rigorous semantic descriptions that are readable and correspond to the reader's intuitive understanding of the behaviors being described. A measure of the directness of the descriptions is the ease with which a compiler for a simple language can be developed from its formal description, once the semantic objects involved have been implemented in the target language. Such compilers are not necessarily efficient, but they do illustrate that the semantic concepts, however abstract they may seem at first, are founded in reality and are not just figments of a theorist's overactive imagination. Furthermore, the notation helps to identify
similarities between different language constructs as well as emphasize the simplicity and universality of certain important concepts, e.g., state, bindings (environment), parameter passing mechanisms, and closure.

The thirteen chapters of the book are divided into three parts. Part I, consisting of Chapters 1 through 6, provides an overview of the field of programming language design (Chapter 1) and a discussion of the formal tools that are required.

Chapter 2 is a review of set theory and functions. Most of this material should be familiar to anyone who has completed a course in discrete mathematics. However, there are some topics, notably partial functions, partial evaluation and currying, higher-order functions, and fixed points, that will be new to most students. These concepts play an important role in the discussion of semantics and will be used frequently, so particular attention should be paid to them.

Chapter 3 introduces lambda calculus as an abstract model of function evaluation. While it is introduced as a system that provides a rigorous framework for the investigation of algorithms, no theorems are proved. It is presented as a simple formalism for developing algorithms from scratch, whose behavior is completely understandable because it depends on just one transformation, $\beta$-conversion. Special emphasis is placed on the fact that lambda calculus provides no mechanism for looping or recursion. This introduces a discussion of fixed points and the fixed-point combinator for finding the fixed points of expressions.

The description of programming language syntax is presented in Chapter 4. Context free grammars and Backus-Naur form are discussed. However, the presentation is very selective and no attempt is made to provide a complete introduction to formal languages. Parse trees and the concept of ambiguity are introduced, and abstract syntax is discussed in some detail because of the important role that it plays in semantic descriptions.

The central focus of Part I is Chapter 5. This chapter provides a practical introduction to denotational semantics. The concepts are illustrated by means of simple examples of language specifications. Denotational semantics depends heavily on the use of higher-order functions and the technique of currying, but beyond this the material is straightforward. More subtle issues, which only arise as the languages involved become more sophisticated, are introduced as needed in later chapters.

It is important that this chapter be completely mastered before continuing, because the rest of the text depends so heavily on it.

Chapter 6 completes Part I with a short discussion of the issues involved in language translation. The concepts involved are illustrated with examples of compilers generated from the semantic specifications given in Chapter 5.

© 1994, Andrew Kitchen
Part II, Chapters 7 through 12, is the core of the text. Here the constructions that form the basis of most programming languages are investigated, and certain unifying concepts are identified: bindings, scope, environments, state, closures, looping and recursive constructs as solutions of specific fixed point problems, continuations.

In particular, it is instructive to see the concept of closure (the partial evaluation of a semantic function through the establishment of an environment) recur repeatedly in the definition of scope rules and the specification of different parameter passing protocols.

Chapter 7 introduces the concept of naming, and discusses assignment and binding. The discussion includes a detailed analysis of the issues arising from side effects in expressions.

In Chapter 8, control structures are introduced. The fixed point theory discussed in Chapters 2 and 3 is applied to the problem of calculating a closed-form expression for the semantics of looping.

Understanding of the scope of declarations and the use of binding environments is the focus of Chapter 9. This chapter introduces block structured languages and procedures, and the concepts of static and dynamic scoping.

Chapter 10 investigates parameter passing and shows how most parameter passing protocols are descended from the two most elementary forms (from a theoretical point of view), pass-by-constant-value and pass-by-name.

The state-based semantics used up to this point, has serious shortcomings when it comes to the description of sophisticated flow control. We have seen that this style of semantics can handle if-then-else statements and while loops. However, more general constructs, like the "infamous" goto statement, present serious problems. In Chapter 11, we discuss these problems and introduce a solution, in the form of a more powerful descriptive mechanism for control flow, continuation semantics. Continuation semantics is used to analyze constructs like the C language statement break. A major portion of this chapter is devoted to a discussion of exception handling.

Earlier chapters have carefully skirted issues of data representation and type checking, in order to keep the presentation simple. Chapter 12 discusses data types and shows how the semantic tools introduced in earlier chapters can be used to enforce type consistency. Input and output processing is introduced as one example of data manipulation.

The final section of the book, Part III, consists of a single chapter. Chapter 13 is an introduction to the theory of domains. It provides theoretical justification for some of the techniques used in Parts I and II and proves the existence of the recursively defined domains and functions that appear in the specifications of many of the languages discussed in earlier chapters.

© 1994, Andrew Kitchen
Some readers may be disappointed to find no discussion of logic languages and other languages based on non-standard programming paradigms. Others may be unsatisfied by the limited coverage of type structure and object-oriented languages. While both of these are reasonable objections, it was decided not to attempt to make the coverage encyclopedic, but rather to focus on a solid presentation of a core of basic material.

This text assumes that the reader has a basic knowledge of discrete mathematics (equivalent to that provided by an introductory discrete mathematics course for mathematics or computer science majors), and has some significant programming experience. A survey course on programming language concepts is highly recommended, as is some familiarity with Lisp or another functional programming language.

The material is designed to be covered in a single, fifteen week semester or a ten week quarter. However, some topics will have to be covered lightly or skipped in order to fit the material into ten weeks. The essential core of the course is embodied in Part II. Part I could be covered in less detail, if the students involved have a uniformly strong background. For example, it might be sufficient to include just the advanced material on functions and the chapter on lambda calculus. Part III could be included between Parts I and II, if the emphasis of the course is more theoretical, and it could be skipped altogether for a course that emphasizes programming language structure over theoretical issues.

The presentation depends heavily on the specification and implementation of a set of simple languages. The specifications and implementations of these languages are available on the Web. For details, see Appendix A.