Chapter 1

Introduction

Order and simplification are the first steps toward the mastery of a subject — the actual enemy is the unknown.

— The Magic Mountain, Thomas Mann

1.1 Programming Languages

Programming is a great load of fun. As you have no doubt experienced, clarity and simplicity are the keys to good programming. When you have a tangle of code that is difficult to understand, your confidence in its behavior wavers, and the code is no longer any fun to read or update.

Designing a new programming language is a kind of meta-level programming activity that is just as much fun as programming in a regular language (if not more so). You will discover that clarity and simplicity are even more important in language design than they are in ordinary programming. Today hundreds of programming languages are in use — whether they be scripting languages for Internet commerce, user interface programming tools, spreadsheet macros, or page format specification languages that when executed can produce formatted documents. Inspired application design often requires a programmer to provide a new programming language or to extend an existing one. This is because flexible and extensible applications need to provide some sort of programming capability to their end users.

Elements of programming language design are even found in "ordinary" programming. For instance, consider designing the interface to a collection data structure. What is a good way to encapsulate an iteration idiom over the elements of such a collection? The issues faced in this problem are similar to those in adding a looping construct to a programming language.

The goal of this book is to teach you the great ideas in programming languages in a simple framework that strips them of complexity. You will learn several ways to specify the meaning of programming language constructs and will see that small changes in these specifications can have dramatic consequences for program behavior. You will explore many dimensions of the programming language design space, study decisions to be made along each dimension, and consider how decisions from different dimensions can interact. We will teach you about a wide variety of neat tricks for extending programing languages with interesting features like undoable state changes, exitable loops, pattern matching, and multitasking. Our approach for teaching you this material is based on the premise that when language behaviors become incredibly complex, the descriptions of the behaviors must be incredibly simple. It is the only hope.

1.2 Syntax, Semantics, and Pragmatics

Programming languages are traditionally viewed in terms of three facets:

1. **Syntax** — the *form* of programming languages.
2. **Semantics** — the *meaning* of programming languages.
3. **Pragmatics** — the *implementation* of programming languages.

Here we briefly describe these facets.

Syntax

Syntax focuses on the concrete notations used to encode programming language phrases. Consider a phrase that indicates the sum of the product of w and x and the quotient of y and z. Such a phrase can be written in many different notations: as a traditional mathematical expression

$$wx + y/z$$

or as a LISP parenthesized prefix expression

```
(+ (* w x) (/ y z))
```

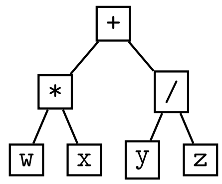
or as a sequence of keystrokes on a postfix calculator

$$\boxed{\text{W}}\ \boxed{\text{ENTER}}\ \boxed{\text{X}}\ \boxed{\text{ENTER}}\ \boxed{\times}\ \boxed{\text{Y}}\ \boxed{\text{ENTER}}\ \boxed{\text{Z}}\ \boxed{\text{ENTER}}\ \boxed{\div}\ \boxed{+}$$

or as a layout of cells and formulae in a spreadsheet

	1	**2**	**3**	**4**
A	`w=`		`w*x =`	*A2 * B2*
B	`x=`		`y/z =`	*C2 / D2*
C	`y=`		`ans =`	*A4 + B4*
D	`z=`			

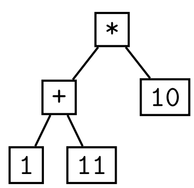
or as a graphical tree



Although these concrete notations are superficially different, they all designate the same abstract phrase structure (the sum of a product and a quotient). The syntax of a programming language specifies which concrete notations (strings of characters, lines on a page) in the language are legal and which tree-shaped abstract phrase structure is denoted by each legal notation.

Semantics

Semantics specifies the mapping between the structure of a programming language phrase and what the phrase means. Such phrases have no inherent meaning: their meaning is only determined in the context of a system for interpreting their structure. For example, consider the following expression tree:



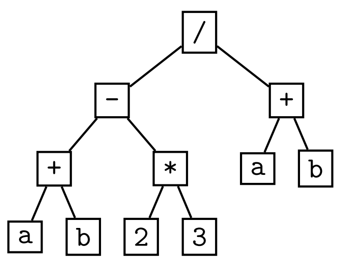
Suppose we interpret the nodes labeled `1`, `10`, and `11` as the usual decimal notation for numbers, and the nodes labeled `+` and `*` as the sum and product of the values of their subnodes. Then the root of the tree stands for $(1+11)\cdot 10 = 120$. But there are many other possible meanings for this tree. If `*` stands for exponentiation rather than multiplication, the meaning of the tree could be 12^{10}. If the numerals are in binary notation rather than decimal notation, the tree could stand for (in decimal notation) $(1+3)\cdot 2 = 8$. Alternatively, `1` and `11` might represent the set of odd integers, `10` might represent the set of even integers, and `+` and `*` might represent addition and multiplication on integer sets; in this case, the meaning of the tree would be the set of even integers. Perhaps the tree does not indicate an evaluation at all, and only stands for a property intrinsic to the tree, such as its height (3), its number of nodes (5), or its shape (perhaps it describes a simple corporate hierarchy). Or maybe the tree is an arbitrary encoding of a particular object of interest, such as a rock or a book.

This example illustrates how a single program phrase can have many possible meanings. Semantics describes the relationship between the abstract structure of a phrase and its meaning.

Pragmatics

Whereas semantics deals with *what* a phrase means, pragmatics focuses on the details of *how* that meaning is computed. Of particular interest is the effective use of various resources, such as time, space, and access to shared physical devices (storage devices, network connections, video monitors, printers, etc.).

As a simple example of pragmatics, consider the evaluation of the following expression tree (under the first semantic interpretation described above):



Suppose that `a` and `b` stand for particular numeric values. Because the phrase `(+ a b)` appears twice, a naïve evaluation strategy will compute the same sum twice. An alternative strategy is to compute the sum once, save the result, and use the saved result the next time the phrase is encountered. The alternative strategy does not change the meaning of the program, but does change its use of resources; it reduces the number of additions performed, but may require extra storage for the saved result. Is the alternative strategy better? The answer depends on the details of the evaluation model and the relative importance of time and space.

Another potential improvement in the example is the phrase `(* 2 3)`, which always stands for the number 6. If the sample expression is to be evaluated many times (for different values of `a` and `b`), it may be worthwhile to replace `(* 2 3)` by `6` to avoid unnecessary multiplications. Again, this is a purely pragmatic concern that does not change the meaning of the expression.

1.3 Goals

The goals of this book are to explore the semantics of a comprehensive set of programming language design idioms, show how they can be combined into complete practical programming languages, and discuss the interplay between semantics and pragmatics. Except for establishing a few syntactic conventions at the outset, we won't say much about syntax at all. We will introduce a number of tools for describing the semantics of programming languages, and will use these tools to build intuitions about programming language features and study many of the dimensions along which languages can vary. Our coverage of pragmatics is mainly at a high level: we will study some simple programming language implementation techniques and program improvement strategies rather than focus on squeezing the last ounce of performance out of a particular computer architecture.

We will discuss programming language features in the context of several **mini-languages**. Each of these is a simple language that captures the essential features of a class of existing programming languages. In many cases, the mini-languages are so pared down that they are hardly suitable for serious programming activities. Nevertheless, these languages embody all of the key ideas in programming languages. Their simplicity saves us from getting bogged down in needless complexity in our explorations of semantics and pragmatics. And like good modular building blocks, the components of the mini-languages are designed to be "snapped together" to create practical languages.

1.4 PostFix: A Simple Stack Language

We will introduce the tools for syntax, semantics, and pragmatics in the context of a mini-language called PostFix. PostFix is a simple stack-based language inspired by the PostScript graphics language, the Forth programming language, and Hewlett Packard calculators. Here we give an informal introduction to PostFix in order to build some intuitions about the language. In subsequent chapters, we will introduce tools that allow us to study PostFix in more depth.

1.4.1 Syntax

The basic syntactic unit of a PostFix program is the **command**. Commands are of the following form:

- Any integer numeral. E.g., `17`, `0`, `-3`.
- One of the following special command tokens: `add`, `div`, `eq`, `exec`, `gt`, `lt`, `mul`, `nget`, `pop`, `rem`, `sel`, `sub`, `swap`.
- An **executable sequence** — a single command that serves as a subroutine. It is written as a parenthesized list of subcommands separated by whitespace.[1] E.g., `(7 add 3 swap)` and `(2 (5 mul) exec add)`.

Since executable sequences contain other commands (including other executable sequences), they can be arbitrarily nested. An executable sequence counts as a single command despite its hierarchical structure.

A PostFix **program** is a parenthesized sequence consisting of (1) the token `postfix` followed by (2) a natural number (i.e., non-negative integer) indicating the number of program parameters followed by (3) zero or more PostFix commands. For example, here are some sample PostFix programs:

```
(postfix 0 4 7 sub)

(postfix 2 add 2 div)

(postfix 4 4 nget 5 nget mul mul swap 4 nget mul add add)

(postfix 1 ((3 nget swap exec) (2 mul swap exec) swap)
           (5 sub) swap exec exec)
```

In PostFix, as in all the languages we'll be studying, all parentheses are required and none are optional. Moving parentheses around changes the structure of the program and most likely changes its behavior. Thus, while the following PostFix executable sequences use the same numerals and command tokens in the same order, they are distinguished by their parenthesization, which, as we shall see below, makes them behave differently.

```
((1) (2 3 4) swap exec)
((1 2) (3 4) swap exec)
((1 2) (3 4 swap) exec)
```

[1]Whitespace is any contiguous sequence of characters that leave no mark on the page, such as spaces, tabs, and newlines.

1.4.2 Semantics

The meaning of a PostFix program is determined by executing its commands in left to right order. Each command manipulates an implicit stack of values that initially contains the integer arguments of the program (where the first argument is at the top of the stack and the last argument is at the bottom). A value on the stack is either (1) an integer numeral or (2) an executable sequence. The result of a program is the integer value at the top of the stack after its command sequence has been completely executed. A program signals an error if (1) the final stack is empty, (2) the value at the top of the final stack is not an integer, or (3) an inappropriate stack of values is encountered when one of its commands is executed.

The behavior of PostFix commands is summarized in Figure 1.1. Each command is specified in terms of how it manipulates the implicit stack. We use the notation $P \xrightarrow{args} v$ to mean that executing the PostFix program P on the integer argument sequence $args$ returns the value v. The notation $P \xrightarrow{args}$ `error` means that executing the PostFix program P on the arguments signals an error. Errors are caused by inappropriate stack values or an insufficient number of stack values. In practice, it is desirable for an implementation to indicate the type of error. We will use comments (delimited by squiggly braces) to explain errors and other situations.

To illustrate the meanings of various commands, we show the results of some simple program executions. For example, numerals are pushed onto the stack, while `pop` and `swap` are the usual stack operations.

`(postfix 0 1 2 3)` $\xrightarrow{[]}$ 3 {*Only the top stack value is returned.*}
`(postfix 0 1 2 3 pop)` $\xrightarrow{[]}$ 2
`(postfix 0 1 2 swap 3 pop)` $\xrightarrow{[]}$ 1
`(postfix 0 1 swap)` $\xrightarrow{[]}$ `error` {*Not enough values to swap.*}
`(postfix 0 1 pop pop)` $\xrightarrow{[]}$ `error` {*Empty stack on second pop.*}

Program arguments are pushed onto the stack (from last to first) before the execution of the program commands.

`(postfix 2)` $\xrightarrow{[3,4]}$ 3 {*Initial stack has 3 on top with 4 below.*}
`(postfix 2 swap)` $\xrightarrow{[3,4]}$ 4
`(postfix 3 pop swap)` $\xrightarrow{[3,4,5]}$ 5

It is an error if the actual number of arguments does not match the number of parameters specified in the program.

`(postfix 2 swap)` $\xrightarrow{[3]}$ `error` {*Wrong number of arguments.*}
`(postfix 1 swap)` $\xrightarrow{[3]}$ `error` {*Not enough values to swap.*}

- N : Push the numeral N onto the stack.
- `sub` : Call the top stack value v_1 and the next-to-top stack value v_2. Pop these two values off the stack and push the result of $v_2 - v_1$ onto the stack. If there are fewer than two values on the stack or the top two values aren't both numerals, signal an error. The other binary arithmetic operators — `add` (addition), `mul` (multiplication), `div` (integer division[a]) and `rem` (remainder of integer division) — behave similarly. Both `div` and `rem` signal an error if v_1 is zero.
- `lt` : Call the top stack value v_1 and the next-to-top stack value v_2. Pop these two values off the stack. If $v_2 < v_1$, then push a `1` (a true value) on the stack, otherwise push a `0` (false). The other binary comparison operators — `eq` (equals) and `gt` (greater than) — behave similarly. If there are fewer than two values on the stack or the top two values aren't both numerals, signal an error.
- `pop` : Pop the top element off the stack and discard it. Signal an error if the stack is empty.
- `swap` : Swap the top two elements of the stack. Signal an error if the stack has fewer than two values.
- `sel` : Call the top three stack values (from top down) v_1, v_2, and v_3. Pop these three values off the stack. If v_3 is the numeral `0`, push v_1 onto the stack; if v_3 is a non-zero numeral, push v_2 onto the stack. Signal an error if the stack does not contain three values, or if v_3 is not a numeral.
- `nget` : Call the top stack value v_{index} and the remaining stack values (from top down) $v_1, v_2, \ldots, v_n$. Pop v_{index} off the stack. If v_{index} is a numeral i such that $1 \leq i \leq n$ and v_i is a numeral, push v_i onto the stack. Signal an error if the stack does not contain at least one value, if v_{index} is not a numeral, if i is not in the range $[1, n]$, or if v_i is not a numeral.
- `(`$C_1 \ \ldots \ C_n$`)` : Push the *executable sequence* `(`$C_1 \ \ldots \ C_n$`)` as a single value onto the stack. Executable sequences are used in conjunction with `exec`.
- `exec` : Pop the executable sequence from the top of the stack, and prepend its component commands onto the sequence of currently executing commands. Signal an error if the stack is empty or the top stack value isn't an executable sequence.

[a] The integer division of n and d returns the integer quotient q such that $n = qd + r$, where r (the remainder) is such that $0 \leq r < |d|$ if $n \geq 0$ and $-|d| < r \leq 0$ if $n < 0$.

Figure 1.1: English semantics of PostFix commands.

Note that program arguments must be integers — they cannot be executable sequences.

Numerical operations are expressed in postfix notation, in which each operator comes after the commands that compute its operands. `add`, `sub`, `mul`, and `div` are binary integer operators. `lt`, `eq`, and `gt` are binary integer predicates returning either `1` (true) or `0` (false).

`(postfix 1 4 sub)` $\xrightarrow{[3]}$ -1
`(postfix 1 4 add 5 mul 6 sub 7 div)` $\xrightarrow{[3]}$ 4
`(postfix 5 add mul sub swap div)` $\xrightarrow{[7,6,5,4,3]}$ -20
`(postfix 3 4000 swap pop add)` $\xrightarrow{[300,20,1]}$ 4020
`(postfix 2 add 2 div)` $\xrightarrow{[3,7]}$ 5 {*An averaging program.*}
`(postfix 1 3 div)` $\xrightarrow{[17]}$ 5
`(postfix 1 3 rem)` $\xrightarrow{[17]}$ 2
`(postfix 1 4 lt)` $\xrightarrow{[3]}$ 1
`(postfix 1 4 lt)` $\xrightarrow{[5]}$ 0
`(postfix 1 4 lt 10 add)` $\xrightarrow{[3]}$ 11
`(postfix 1 4 mul add)` $\xrightarrow{[3]}$ `error` {*Not enough numbers to add.*}
`(postfix 2 4 sub div)` $\xrightarrow{[4,5]}$ `error` {*Divide by zero.*}

In all the above examples, each stack value is used at most once. Sometimes it is desirable to use a number two or more times or to access a number that is not near the top of the stack. The `nget` command is useful in these situations; it puts at the top of the stack a copy of a number located on the stack at a specified index. The index is 1-based, from the top of the stack down, not counting the index value itself.

`(postfix 2 1 nget)` $\xrightarrow{[4,5]}$ 4 {*4 is at index 1, 5 at index 2.*}
`(postfix 2 2 nget)` $\xrightarrow{[4,5]}$ 5

It is an error to use an index that is out of bounds or to access a non-numeric stack value (i.e., an executable sequence) with `nget`.

`(postfix 2 3 nget)` $\xrightarrow{[4,5]}$ `error` {*Index 3 is too large.*}
`(postfix 2 0 nget)` $\xrightarrow{[4,5]}$ `error` {*Index 0 is too small.*}
`(postfix 1 (2 mul) 2 nget)` $\xrightarrow{[3]}$ `error` {*Value at index 2 is not a number.*}

The `nget` command is particularly helpful for expressing numerical programs, where it is common to reference arbitrary parameter values and use them multiple times.

`(postfix 1 1 nget mul)` $\xrightarrow{[5]}$ 25 {*A squaring program.*}

`(postfix 4 4 nget 5 nget mul mul swap 4 nget mul add add)` $\xrightarrow{[3,4,5,2]}$ 25
{*Given a, b, c, x, calculates* $ax^2 + bx + c$.}

As illustrated in the last example, the index of a given value increases every time a new value is pushed on the stack.

Executable sequences are compound commands like `(2 mul)` that are pushed onto the stack as a single value. They can be executed later by the `exec` command. Executable sequences act like subroutines in other languages; execution of an executable sequence is similar to a subroutine call, except that transmission of arguments and results is accomplished via the stack.

`(postfix 1 (2 mul) exec)` $\xrightarrow{[7]}$ 14 {`(2 mul)` *is a doubling subroutine.*}

`(postfix 0 (0 swap sub) 7 swap exec)` $\xrightarrow{[]}$ -7
{`(0 swap sub)` *is a negation subroutine.*}

`(postfix 0 (7 swap exec) (0 swap sub) swap exec)` $\xrightarrow{[]}$ -7

`(postfix 0 (2 mul))` $\xrightarrow{[]}$ `error` {*Final top of stack is not an integer.*}

`(postfix 0 3 (2 mul) gt)` $\xrightarrow{[]}$ `error`
{*Executable sequence where number expected.*}

`(postfix 0 3 exec)` $\xrightarrow{[]}$ `error` {*Number where executable sequence expected.*}

```
(postfix 1 ((3 nget swap exec) (2 mul swap exec) swap)
           (5 sub) swap exec exec)
```
$\xrightarrow{[7]}$ 9
{*Given n, calculates 2n - 5.*}

The last example illustrates that evaluations involving executable sequences can be rather contorted.

The `sel` command selects between two values based on a test value, where zero is treated as false and any non-zero integer is treated as true. It can be used in conjunction with `exec` to conditionally execute one of two executable sequences.

`(postfix 1 2 3 sel)` $\xrightarrow{[1]}$ 2

`(postfix 1 2 3 sel)` $\xrightarrow{[0]}$ 3

`(postfix 1 2 3 sel)` $\xrightarrow{[17]}$ 2 {*Any non-zero number is "true".*}

`(postfix 0 (2 mul) 3 4 sel)` $\xrightarrow{[]}$ `error` {*Test not a number.*}

`(postfix 4 lt (add) (mul) sel exec)` $\xrightarrow{[3,4,5,6]}$ 30

`(postfix 4 lt (add) (mul) sel exec)` $\xrightarrow{[4,3,5,6]}$ 11

`(postfix 1 1 nget 0 lt (0 swap sub) () sel exec)` $\xrightarrow{[-7]}$ 7
{*An absolute value program.*}

`(postfix 1 1 nget 0 lt (0 swap sub) () sel exec)` $\xrightarrow{[6]}$ 6

▷ **Exercise 1.1** Determine the value of the following PostFix programs on an empty stack.

a. `(postfix 0 10 (swap 2 mul sub) 1 swap exec)`

b. `(postfix 0 (5 (2 mul) exec) 3 swap)`

c. `(postfix 0 (() exec) exec)`

d. `(postfix 0 2 3 1 add mul sel)`

e. `(postfix 0 2 3 1 (add) (mul) sel)`

f. `(postfix 0 2 3 1 (add) (mul) sel exec)`

g. `(postfix 0 0 (2 3 add) 4 sel exec)`

h. `(postfix 0 1 (2 3 add) 4 sel exec)`

i. `(postfix 0 (5 6 lt) (2 3 add) 4 sel exec)`

j.
```
(postfix 0 (swap exec swap exec) (1 sub) swap (2 mul)
           swap 3 swap exec)
```
◁

▷ **Exercise 1.2** Write executable sequences that compute the following logical operations. Recall that `0` is false and all other numerals are treated as true.

a. *not*: return the logical negation of a single argument.

b. *and*: given two numeric arguments, return `1` if their logical conjunction is true, and `0` otherwise.

c. *short-circuit-and*: return 0 if the first argument is false; otherwise return the second argument.

d. Demonstrate the difference between *and* and *short-circuit-and* by writing a PostFix program that has a different result if *and* is replaced by *short-circuit-and*. ◁

▷ **Exercise 1.3**

a. Without `nget`, is it possible to write a PostFix program that squares its single argument? If so, write it; if not, explain.

b. Is it possible to write a PostFix program that takes three integers and returns the smallest of the three? If so, write it; if not, explain.

c. Is it possible to write a PostFix program that calculates the factorial of its single argument (assume it's non-negative)? If so, write it; if not, explain. ◁

1.4.3 The Pitfalls of Informal Descriptions

The "by-example" and English descriptions of PostFix given above are typical of the way that programming languages are described in manuals, textbooks, courses, and conversations. That is, a syntax for the language is presented, and the semantics of each of the language constructs is specified using English prose and examples. The utility of this method for specifying semantics is apparent from the fact that the vast majority of programmers learn to read and write programs via this approach.

But there are many situations in which informal descriptions of programming languages are inadequate. Suppose that we want to improve a program by substituting one phrase for another throughout the program. How can we be sure that the substitution preserves the meaning of the program?

Or suppose that we want to prove that the language as a whole has a particular property. For instance, it turns out that every PostFix program is guaranteed to terminate (i.e., a PostFix program cannot enter an infinite loop). How would we go about proving this property based on the informal description? Natural language does not provide any rigorous framework for reasoning about programs or programming languages. Without the aid of some formal reasoning tools, we can only give hand-waving arguments that are not likely to be very convincing.

Or suppose that we wish to extend PostFix with features that make it easier to use. For example, it would be nice to name values, to collect values into arrays, to query the user for input, and to loop over sequences of values. With each new feature, the specification of the language becomes more complex, and it becomes more difficult to reason about the interaction between various features. We'd like techniques that help to highlight which features are orthogonal and which can interact in subtle ways.

Or suppose that a software vendor wants to develop PostFix into a product that runs on several different machines. The vendor wants any given PostFix program to have exactly the same behavior on all of the supported machines. But how do the development teams for the different machines guarantee that they're all implementing the "same" language? If there are any ambiguities in the PostFix specification that they're implementing, different development teams might resolve the ambiguity in incompatible ways. What's needed in this case is an unambiguous specification of the language as well as a means of proving that an implementation meets that specification.

The problem with informal descriptions of a programming language is that they're neither concise nor precise enough for these kinds of situations. English is often verbose, and even relatively simple ideas can be unduly complicated to explain. Moreover, it's easy for the writer of an informal specification to underspecify a language by forgetting to cover all the special cases (e.g., error situations in PostFix). It isn't that covering all the special cases is impossible; it's just that the natural language framework doesn't help much in pointing out what the special cases are.

It is possible to overspecify a language in English as well. Consider the PostFix programming model introduced above. The current state of a program is captured in two entities: the stack and the current command sequence. To programmers and implementers alike, this might imply that a language implementation *must* have explicit stack and command sequence elements in it. Although these would indeed appear in a straightforward implementation, they are not in any way *required*; there are alternative models and implementations for PostFix (see Exercise 3.12). It would be desirable to have a more abstract definition of what constitutes a legal PostFix implementation so that a would-be implementer could be sure that an implementation was faithful to the language definition regardless of the representations and algorithms employed.

In the remaining chapters of the first segment of this book, we introduce a number of tools that address the inadequacies outlined above. First, in Chapter 2 we present **s-expression grammars**, a simple specification for syntax that we will use to describe the structure of all of the mini-languages we explore. Then, using PostFix as our object of study, we introduce two approaches to formal semantics:

- An **operational semantics** (Chapter 3) explains the meaning of programming language constructs in terms of the step-by-step process of an abstract machine.

- A **denotational semantics** (Chapter 4) explains the meaning of programming language constructs in terms of the meaning of their subparts.

These approaches support the unambiguous specification of programming languages and provide a framework in which to reason about properties of programs and languages. This segment concludes in Chapter 5 with a presentation of a technique for determining the meaning of recursive specifications.

Throughout the book, mathematical concepts are formalized in terms of the **metalanguage** described in Appendix A. Readers are encouraged to familiarize themselves with this language by skimming the appendix early on and later referring to it in more detail on an "as needed" basis.

While we will emphasize formal tools throughout this book, we do not imply that formal tools are a panacea or that formal approaches are superior to informal ones in an absolute sense. In fact, informal explanations of language features are usually the simplest way to learn about a language. In addition, it's very easy for formal approaches to get out of control, to the point where they are overly obscure, or require too much mathematical machinery to be of any practical use on a day-to-day basis. For this reason, we won't dwell on nitty gritty formal details and won't cover material as a dry sequence of definitions, theorems, and proofs. Instead, our goal is to show that the *concepts* underlying the formal approaches are indispensable for understanding particular programming languages as well as the dimensions of language design. The tools introduced in this segment should be in any serious computer scientist's bag of tricks.

Reading

No single book can entirely cover the broad area of programming languages. We recommend the following books for other perspectives of the field:

- Mitchell has authored two relevant books: [Mit96] is a mathematical exploration of programming language semantics based on a series of typed lambda calculi, while [Mit03] discusses the dimensions of programming languages in the context of many modern programming languages.
- Friedman, Wand, and Haynes [FWH01] uses interpreters and translators written in SCHEME to study essential programming language features in the context of some mini-languages.
- Reynolds [Rey98] gives a theoretical treatment of many programming language features.
- Gelernter and Jaganathan [GJ90] discusses a number of popular programming languages in a historical perspective and compare them in terms of expressiveness.
- MacLennan's text [Mac99] stands out as one of the few books on programming languages to enumerate a set of principles and then analyze popular languages in terms of these principles.
- Kamin [Kam90] uses interpreters written in PASCAL to analyze the core features of several popular languages.
- Marcotty and Ledgard [ML86] cover a wide range of programming language features and paradigms by presenting a sequence of mini-languages.
- Gunter [Gun92] provides an in-depth overview of formal programming language semantics.

- Winskel [Win93] presents a mathematical introduction to formal programming language semantics.
- Horowitz [Hor95] has collected an excellent set of classic papers on the design of programming languages that every programming language designer should be familiar with.