1 Introduction

The word 'semantics' was introduced in a book of Michel Bréal published in 1900; in that work it refers to the study of how words change their meanings. Subsequently 'semantics' has also changed its meaning, and it is now generally defined as the study of the attachment between the words and sentences of a language (written or spoken) and their meanings. The most established lines of investigation of semantics lie in the areas of linguistics and philosophy that study the meanings of sentences of natural language. A second area of study of semantics focuses on the meanings of sentences in the formal languages of mathematical logic, originally designed to serve as a foundation for general mathematics. This book is devoted to a discussion of topics from a third area of semantics that focuses on developing ways to express the semantics of languages used for programming electronic digital computers. We will be especially concerned with the application of ideas from semantics in mathematical logic toward achieving the goals of the semantics of programming languages.

1.1 Semantics

The methods and objectives of semantics of programming languages are different in many ways from those of the semantics of natural language and mathematical logic but borrow a great deal from the work in these areas. Traditionally, computer languages have been viewed as based on imperative sentences for issuing commands. In natural language the analogous sentences are like those that might be found in a recipe book: 'beat the egg whites until they are stiff'. By contrast, the sentences of mathematical logics are intended to assert timeless truths: 'if beaten, egg whites stiffen'. Much of the current research in methods for reasoning about programs seeks to formalize the relationship between examples such as these—after all, the assertion about the beating of egg whites means that the chef obeying the command from the recipe book will see a successful termination to his labors (although nothing is explicitly said about how tired his arm will be at the end). Moreover, a great deal of current research in programming language design aims at blurring the distinction between a program as a set of instructions and a program as an assertion about the desired answer. This idea is embodied in terms like 'declarative programming' and 'executable specification'. For example, the following pair of assertions (clauses) are a definition of a relation in Prolog syntax:

```
gcd(X,0,X).
gcd(X,Y,Gcd) :- mod(X,Y,Z), gcd(Y,Z,Gcd).
```

Given that `mod(X,Y,Z)` means that `Z` is the remainder of `X` divided by `Y`, the usual semantics of first-order logic provides a meaning for `gcd` as the unique relation that satisfies these two formulas (for non-negative values of the identifiers). It is an aim of Prolog programming methodology that one think of the program in these terms whenever possible. In this case the evaluator for Prolog makes it possible to use the program above to calculate a binary function on numbers using the relation **gcd**. Thus these clauses are a piece of logic that has an algorithmic content. Unfortunately, it will not be the case that every assertion in first order logic can be given a similar algorithmic interpretation derived by the Prolog interpreter. The explanation of what a program is required to compute does not necessarily provide an algorithm for computing it. In particular, the *greatest common divisor* of x and y is typically defined as the largest number z such that z divides both x and y. It is not hard to show that this definition is meaningful in the sense that there is a unique relation on numbers that satisfies it. However, this definition does not easily admit the efficient algorithmic interpretation that Prolog endows on the definition of gcd despite the fact that both describe the same relation. What Prolog offers beyond a syntax for defining a relation is an algorithm for computing values that satisfy it. The semantics of the Prolog clauses as first-order formulas does not always coincide with the results of this algorithm, but when they do coincide, they offer two different views of the meaning of the program, each having its own advantages in how the programmer wants to understand what has been coded.

Other examples of programs that can be dually viewed as commands or assertions arise as recursive definitions of procedures. The following program codes the greatest common divisor function in the Scheme programming language:

```
(define (gcd x y)
  (if (= y 0)
      x
      (gcd y (rem x y))))
```

Rather than two formulas, the definition has the appearance of an equation

$$\mathrm{gcd}(x, y) = \cdots \text{ gcd } \cdots$$

in which the defined function appears on the right-hand side as well as the left. Assuming that the arguments are non-negative, this equation has a unique solution; this solution can be viewed as the meaning of gcd. The defining equation is special though since it provides a way to calculate the greatest common divisor following an algorithm defined by the Scheme interpreter. This provides two alternate semantics for the program, each describing the same function but in different ways.

It is also possible to view 'imperative' programming language commands as assertions by thinking of them as asserting a relation between the state of the computer before the execution of the command and the state after its execution. For example, the command `x := x+1` can be viewed as an order to the machine: 'take the value in the location `x` and add one to it, then place the resulting value in the location `x`'. But it may also be viewed as a relation between the contents of memory locations: an input state is related to an output state via the relation defined by this assignment if the value contained in `x` in the output is one plus the value stored there in the input (and the state is otherwise unaltered). This relational interpretation of program fragments can be generalized by interpreting a command as a relation between *properties* of input and output values.

The languages of mathematical logic and programming languages share the feature that they are 'artificial' languages distinct from the 'natural' language of everyday writing and discourse. This has an effect on the methodology that is employed in these areas. Natural language research draws from the existing body of spoken and written languages; work in natural language semantics seeks to understand how native speakers attach meaning to their sentences—the ear and intuition of the native speaker is the ultimate authority on what is a correct sentence and what the sentence means. Although grammarians can advise on the proper use of words and their arrangement into pleasing and correct sentences, even the *Académie Française* cannot be credited with designing a natural language. By contrast, the formal languages of logic and programming are synthesized by logicians and computer scientists to meet more or less specific design criteria.

Before discussing some of the consequences of this difference, let us first consider how much similarity there is between work on designing artificial languages and work on understanding natural languages. It is reasonable to view the aim of research in logic (at least during the middle of the twentieth century) as an attempt to formalize the language of mathematicians and formulate axioms and rules capturing the methods of proof in mathematics. If the arguments of mathematicians are viewed as a specialized form of the use of natural language, then this provides some analogy with the work of linguists. At this point in time, it is possible to take a proof written by a mathematician in English (say) and formulate *essentially* the same proof in first-order logic, second-order logic, Church's higher-order logic, or some similar language. Moreover, there are successful efforts to design formal languages capable of turning a mathematical textbook into something that looks like a computer program (although this cannot currently be done automatically). Such a translation ignores the motivations and intuitions as well as the jokes (of which there are few in most mathematics texts anyway) but captures a degree of detail that any mathematician would consider to be sufficient for a proof.

Research on the semantics of programming languages has also had its similarities to the study of natural language. Many approaches to the semantics of programming languages have been offered as tools for language design, but rather often these sometimes meritorious and sometimes over-idealistic approaches have been ignored by the committees responsible for language designs. This has often led to post-design analyses of the semantics of programming languages wherein the syntax of the language is treated as given. A semantics is also given, in some more or less reasonable form. The goal then is to understand the semantics of the language in another way or to sort out anomalies, omissions, or defects in the given semantics. Examples of this kind of analysis have occurred for virtually every programming language in widespread use, and a partial list of papers published in pursuit of such endeavors could fill fifty pages of references at the end of this book. As a result, it is sometimes felt that research in semantics has had less impact on language design than it should have had. While there is probably truth in this, it is also the case that no successful language has been designed without some serious consideration of how the meanings of programs will be best described. Every programmer must understand the meanings of the language constructs he uses at *some* level of abstraction and detail. Post-design analysis of the semantics is reasonable and necessary because there are many different forms of semantic description serving different purposes; it is unlikely that any language can be designed that is optimally understandable in terms of all of them, and it is probably infeasible to investigate all of the myriad descriptive techniques in the course of a given language design. Hence work on the semantics of programming languages will often share with that on natural language the characteristics that the language is given and its explanation is the goal of research.

Despite these similarities, the difference between the studies of natural and artificial languages is profound. Perhaps the most basic characteristic of this distinction is the fact that an artificial language can be fully circumscribed and studied in its entirety. Throughout this book there are tables that hold the grammar, rules, and semantics of artificial languages. These tables furnish the bottom line for the definition of the language; assertions about the language must be justified in terms of them. For this reason, many of the proofs that will be given proceed by cases (usually with an inductive hypothesis) demonstrating the truth of some claim about the language as a whole by justifying it for each of the constructions allowed by its rules for legitimate formation of expressions.

A particularly beneficial aspect of the ability to design a language comes from the opportunity to separate concerns in a way that will simplify the analysis of the language. What is lost by such separations is open to debate, but the techniques have been developed through extensive experience. Perhaps the most important example of a simplification of this kind is the separation of the grammar of a programming language from its semantics. In the simplest case, a language is specified as a sequence of characters, and it is specified how such a sequence of characters is to be divided into tokens (basic lexical units). A set of rules is given for how a sequence of tokens is formed into a parse tree (or rejected if the sequence is illegitimate in some way). Then, finally, a semantics is assigned to parse trees built in this way.[1] This all sounds very reasonable to most computer scientists, but it is worthwhile to reflect that one of the real challenges in the parsing of natural language sentences comes from the fact that the meanings of words seem to influence the correct parsing of a sentence. A well-known example illustrating the problem is the following pair of grammatically correct and sensible English sentences:

Time flies like an arrow.
Fruit flies like a banana.

In the first sentence, 'flies' is a verb, and the subject of the sentence is 'time' whereas in the second sentence, 'flies' is a noun. In the first sentence 'like an arrow' is an adverbial phrase indicating how time flies whereas 'like' is a verb in the second sentence with 'a banana' as its object. Why do I think this is the right way to parse these sentences? Reversing the way the sentences are parsed brings up semantic questions: what are 'time flies' (some things that like an arrow, I suppose) and what does it mean to fly like a banana? The problem of what is the interaction between syntax and semantics in natural language is an ongoing topic of research. Perhaps the mechanisms whereby natural language combines parsing with semantics could be useful in artificial language design, but for now it seems that logicians and the designers of programming languages are happy to steer clear of this complexity.

Since artificial languages are engineered rather than given, choosing the best constructions from a wide range of possibilities is a central objective. To do this, it is essential to develop ways to compare different languages and explore how one language may serve a given purpose better than another. Work on the parsing of artificial languages has achieved considerable success in understanding the alternatives for syntax. The most useful methods for specifying grammars have been identified, and algorithms for constructing parse trees have been carefully studied. This generality has benefited the engineering side of language design by providing widely-used tools for the construction of quality lexers and parsers automatically from specifications of the tokens and grammar of a language. The semantic side of language design is now the primary challenge. The development of abstractions for describing the range of possibilities for the semantics of languages has been much harder to achieve than the corresponding development for syntax. Many semantic frameworks have been developed, but there is no universal acceptance of any particular approach.

[1]This is an approximation of what happens in practice. For example, it may be that only those parse trees that satisfy certain typing rules are considered to have a semantics. Moreover, the situation can be even more complex if the rules for verifying that a program is well-typed are also used in giving the semantics. These subtleties will later be discussed in detail.

What purpose is served by the semantics of a language? For natural language, the answer seems almost too obvious to state except as another question: what use could be made of a sentence without meaning? If I am told, 'turn right at the next light and you'll see a sign on your left', then my subsequent behavior will depend on what object is denoted by the 'next light', what I think the directions 'right' and 'left' are, what the acts of turning and seeing are, and so on. In mathematical logic, meaning is formalized in the notion of a *model,* which is a mathematical entity like the natural numbers, the complexes, or the quaternions. A logic provides a syntax for asserting facts about the model based on an interpretation of the symbols of the syntax in the model. Ordinarily, the model is the primary object of attention, and the syntax is a tool for its study. For example, the first-order theory of *real closed fields* is a set of axioms from which every first-order fact true of the real numbers can be derived using the laws of first-order logic. This theory is decidable, so any assertion about the reals that can be expressed with a first-order sentence could be established or refuted by submitting it to a computer program that can decide, in principle,[2] whether it is a consequence of the real closed field axioms. Another example of an important model is that of the natural numbers, but a theorem of Kurt Gödel has shown that the first-order theorems that are true of this model (in a language that includes basic operations like addition and multiplication) cannot be decided by a computer. This result has had profound implications, but it is counterbalanced by the existence of a first-order set of axioms, called *Peano arithmetic,* from which many facts about the numbers can be derived—indeed, simple examples of truths about the natural numbers that are not provable from the Peano axioms have been uncovered only recently. Unfortunately, there is no computer program that can tell of a proposition ϕ whether or not it follows from the Peano axioms—any algorithm that seeks to do this will provide wrong answers or fail to terminate on some of its inputs. As a slight compensation for this problem, there is another theory, called *Presburger arithmetic*, that can decide certain propositions about addition. Moreover, there are good algorithms for this decision procedure that are used in practice.

The case in which axioms are intended to describe properties of a given model, the *standard* model, is common in mathematical logic, but there are many instances in which the study of a particular theory is the goal, and no standard model is intended. An example of this from abstract algebra is the the theory of *groups,* which are algebras with a constant for a unit and a single binary operation that is associative and has inverses. There is no 'standard group' that these axioms are intended to describe;[3] instead they describe a collection of properties common to many mathematical structures. Now, a

[2]The algorithm is hyper-exponential, so its performance in practice is a different matter!

[3]It is worth noting, however, that the group axioms were originally intended to axiomatize permutations, so it could be argued that permutation groups are the standard models of the group axioms.

valid proposition about groups is a proposition that is true for each group. Since the group axioms can be formulated in first-order logic, if a first-order proposition in this language is valid, then by the completeness theorem it is also *provable* in the formal language of first-order logic. However, in a book on group theory, such facts are often proven 'model-theoretically' rather than through something that looks like a formal first-order proof. For example, a proof is likely to begin with something like 'let G be a group and suppose x, y are elements of G'. Of course, the group axioms will surely be used in the proof (if anything interesting about groups is being proved), but they will be treated as facts satisfied by the model. A second characteristic of such proofs is the fact that they are *informal.* This is not to say that the proof is somehow imprecise; it is crucial to distinguish between a rigorous but informal proof and a formal one: *formal* proofs are those formulated in the syntax of a particular logic and using its axioms and rules, whereas *informal* proofs are embedded in a specialized part of natural language spoken in mathematical discussions.[4] For a given logic, it may be that some rigorous, informal, semantic proofs can be easily converted to formal ones, but it is likely that other proofs will not be so easy. The point here is that semantic arguments are the norm in mathematics, even where formal syntactic proof is quite possible.

By contrast, the literature on proving properties about computer programs places a great deal of emphasis on formal axiom systems and formal proofs of properties of programs. There are several explanations that could partially account for this phenomenon. First of all, computer programs are themselves formal objects. A formal axiom system for reasoning about programs can take advantage of this intrinsic characteristic by making programs into parts of the formulas of the language. *Hoare triples,* which are generally written in the form $\phi\{C\}\psi$, are an example of this idea: in such triples, ϕ and ψ are typically propositions about the computer memory, and C is a program fragment. This may have the practical consequence that assertions can be used to annotate actual computer programs, and these assertions can themselves be processed by an analytic program such as the compiler. A second explanation for the emphasis on formal proof arises from the complexity of computer programs. Large portions of typical programs are filled with long and tedious lists of cases. Informal arguments 'by hand' may be unequal to the task of checking that these cases all satisfy the desired conditions. Often, it is not possible to automate the verification of a condition, but when it is, this could free the human certifier of a program's correctness from a very unpleasant and difficult chore. Since complex programs are often not rigorously proved correct (either formally or informally), good programs for automating parts of such a task might lead to more reliable programs. A

[4]In fact, formality is often viewed as a relative matter. A careful, step-by-step proof in precise mathematical language is sometimes called a formal proof when compared to a 'hand-waving' proof that has been carried out through convincing pictures and gesticulations.

third reason for emphasis on formal proof is a poor understanding of how the semantics of a program can be used directly for proving properties of programs. The semantics of a programming language can be very subtle; attempts to avoid semantic arguments by resorting to formal systems of axioms (proved sound for the semantics, of course) are a common way to help deal with this subtlety.

One thing offered by the semantics of a language is a theory of *equivalence* between expressions. Everyone uses the idea that there are many different ways to express the same meaning, but providing a theory that explains this intuitive equivalence relation is no easy task for the semantics of natural language. Theories of semantic equivalence are equally important for programming. In the case of computer programs, the theory of equivalence is based on what is often called the *level of abstraction* of the semantic analysis. To see this at work in a concrete example, consider the following mathematical definition of the Fibonacci function:

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{otherwise.} \end{cases}$$

This definition can be coded directly as a recursive procedure declaration in a language like Scheme:

```
(define (fib n) (if (= n 0) 1
                    (if (= n 1) 1
                        (+ (fib (- n 1))
                           (fib (- n 2))))))).
```

Although this clearly implements the desired mathematical function, the computational properties of this way of coding it are disastrous. Indeed the number of calls to the `fib` routine will be exponential in the size of the argument n. This program does a great deal of redundant calculation, since the value of `(fib (- n 2))` will always be needed in calculating `(fib (- n 1))`, but this value is recalculated in a separate recursive call. A much better way to code the Fibonacci function is to 'remember' the relevant calculation and pass it as a parameter to the recursive call. An auxiliary function can aid in achieving the desired optimization:

```
(define (fib n)
  (define (fib-iter a b count)
    (if (= count 0)
        b
        (fib-iter (+ a b) a (- count 1))))
  (fib-iter 1 0 n)).
```

In this new coding the number of recursive calls is linear in n, a vast improvement over the performance of the previous implementation. It is possible to understand this improvement in terms of a simple rewriting semantics that models the recursive definition through repeated 'unwinding' of the body of the definition. The two programs given previously are equivalent at one level of abstraction, that which considers only the value of the output on a given input, but inequivalent at another level, in which the number of computational steps are considered. Showing that the efficient implementation is 'correct' in the sense that it really does compute the Fibonacci function involves considering its meaning at an abstract level; the justification that it is efficient is based on a more 'low-level' semantics. Given a still lower-level semantics, a programmer may be able to make conclusions about the use of space that programs will make; for example, in a Scheme compiler the second program will require only a constant amount of space.

1.2 Semantics of Programming Languages

What is meant by the 'semantics' of a programming language? Although researchers in mathematical logic have a clear idea of what semantics is for their languages, this level of clarity is not present in the programming languages literature. A crude view is that the semantics of a programming language as defined by a compiler is the mapping that the compiler defines from the program written by a human to the target code executed by the computer. This view is sometimes refined by separating the syntactic analysis done by the compiler from the semantic analysis by classifying syntax as the context-free phase of the compilation, which builds the parse tree for the program, and semantics as the remaining phases, which check types, generate code, perform optimizations, and so on.

If we think of the programming language as something whose semantics we must define and the instruction set for machine architecture as something whose meaning we understand well, then the compiler provides a sort of semantics through translation. The general idea of semantics through translation is an important and pervasive one in the study of programming languages; it will often arise as a theme in later chapters of this book. Nevertheless, it does not work well to specify the semantics of a programming language through a compiler to a specific piece of commercial hardware. There are at least two problems. First of all, such a specification is not good for portability. If the language has been specified by a compiler for machine M and it is to be implemented on another machine M', then either the machine M must be simulated on M' or the compiler must be translated in a 'compatible' way. The use of the word 'compatible' here begs the question of what invariants should be preserved under the translation. That the user should observe no difference in running his programs is not yet a precise condition since it may not be clear what the user can actually observe. A second problem is that a compiler is likely to provide the wrong level of abstraction for a programmer. It may provide useful insights into which commands of the language will run most efficiently (given an understanding of what runs well on the underlying machine), but it is also likely to provide a great deal of arbitrary detail that a programmer does not need and could not really use.

A common approach to resolving this problem is to formulate an abstract machine that can be easily and efficiently simulated on machines with different instruction sets. A compiler for the language for this abstract machine can then be offered as specification of the semantics of the language. The instructions that a program produces for the abstract machine are sometimes called 'intermediate code' since they stand between the program the user wrote and the basic machine instruction set that is the eventual target of the compilation. This may go far in resolving the problem of portability if the abstract machine is chosen well—just abstract enough to be general but low-level enough to be efficient and easy to implement. In short, a good abstract machine for this purpose is the least common denominator for the class of machines over which it is an abstraction. Unfortunately, an abstract machine description of this kind is unlikely to be as useful to a programmer as it is to an implementor of the language, since the compiler for the abstract machine will still include many complexities that the general programmer does not need (or want) to know.

Informal semantics.

The problem with many abstract machines, therefore, is that they may be abstract enough for some but still too low-level for others. Every programmer who uses a higher-level language must understand its semantics at *some* level of abstraction. A compiler for an abstract machine is likely to be hard to understand in detail, even for the clearest language and simplest abstract machine. Hence programmers ordinarily understand the semantics of a language through examples, intuitions about the underlying machine model, and some kind of *informal* semantic description. By way of illustration, an excerpt from one such specification appears in Table 1.1. The example is taken from the ALGOL 60 Revised Report and describes the semantics of the assignment statements in that language. It is not primarily chosen to point out any particular strength or failing of such informal semantic descriptions but merely to give a good example of one.

The ALGOL 60 report uses English sentences to provide such essential information as the order in which expressions are evaluated. For example, it is indicated that the subscript expressions occurring on the left side of the assignment statement are evaluated from left to right, *before* the expression of the assignment is evaluated. The brevity and

Table 1.1
ALGOL 60 Semantics for Assignments

4.2.2. Examples

$$\begin{aligned} s &:= p[0] := n := n+1+s \\ n &:= n+1 \\ A &:= B/C - v - q \times S \\ S[v,k+2] &:= 3 - \mathit{arctan}(s \times \mathit{zeta}) \\ V &:= Q > Y \wedge Z \end{aligned}$$

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 4.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1 Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2 The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

apparent simplicity of the description is partly based on assumptions about what the reader of the specification already understands. For example, the sentence

> Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers.

will mean nothing to someone who does not already understand the idea of assigning a value to a variable. Such assumptions can, and have, led to misunderstandings. In the best circumstances, an experienced programmer's intuition about the meanings of the phrases fills in any omissions in the description, and it can be viewed as a simple, succinct, and readable account of the meanings of language constructs. At worst, however, the description can be fatally ambiguous or misleading, and programming errors or compiler implementation errors can result.

Indeed, the original definition of ALGOL 60 did contain ambiguities, and a revised report was put out in 1963. Work on understanding the ambiguities in the definition continued for some years further; in 1967, Donald Knuth wrote a summary of the known ambiguities and errors in the 1963 Revised Report. He wrote,

Table 1.2
An ALGOL 60 Program

```
begin integer a;
  integer procedure f(x,y);  value y,x;  integer y,x;
    a := f := x + 1;
  integer procedure g(x);  integer x;  x := g := a + 2;
    a := 0; outreal(1, a + (f(a, g(a)) / g(a))) end
```

When ALGOL 60 was first published in 1960, many new features were introduced into programming languages.... It was quite difficult at first for anyone to grasp the full significance of each of the linguistic features with respect to other aspects of the language, and therefore people commonly would discover ALGOL 60 constructions they had never before realized were possible, each time they reread the Report. Such constructions often provided counterexamples to many of the usual techniques of compiler implementation, and in many cases it was possible to construct programs that could be interpreted in more than one way.

Let us say that a feature of ALGOL 60 is *ambiguous* if at least two implementations yielding different behaviors are consistent with its definition in the Report. One of the features that led to questions about ambiguity was the possibility of *side effects,* that is, changes to variables resulting from the evaluation of an expression for which the variable is not local. For example, consider the expression $x + f(x)$. Is it interchangeable with the expression $f(x) + x$? The answer depends on whether the evaluation of f can alter ('side effect') the value of x; if this is possible, then the two expressions may not be interchangeable. For the language implementor, this question is intimately connected with the order in which the parts of an expression are to be evaluated. Consider the program in Table 1.2. This program defines procedures `f` and `g` and calculates the value of the expression

$$\mathtt{a} + (\mathtt{f}(\mathtt{a}, \mathtt{g}(\mathtt{a}))/\mathtt{g}(\mathtt{a})) \tag{1.1}$$

beginning with zero as the initial value of a. For readers not familiar with the syntax of ALGOL-like languages, it suffices simply to understand that the evaluation of procedures `f` and `g` changes the state by modifying the global integer variable a. As a result of this modification, the order in which parts of 1.1 are evaluated will have a significant effect on its value. For example, if it was felt that an optimization could be achieved by evaluating the denominator of a fraction first, then an implementation might proceed by evaluating `g(a)`, then `f(a, g(a))`, and then `a + (f(a, g(a)) / g(a))`. If the parameters for the application of `f` are evaluated in the order `a`, `g(a)`, then the result of this evaluation is $4\frac{1}{2}$. On the other hand, if the expression is evaluated 'from left to right', then the final result will be $\frac{1}{3}$. The ALGOL 60 definition is therefore ambiguous, since it does not specify a fixed order of evaluation for expressions or rule out the possibility of side effects in programs.

Further questions about order of evaluation arise in innocent places. For example, the clause 4.2.3.2 in Table 1.1 says that the evaluation of subscripts (that is, array indices) should be done from 'left to right'. In the expression

```
A[a + B[f(a)] + g(a)] := C[a] := 0
```

there are three subscripts. Should they be evaluated by doing `a + B[f(a)] + g(a)` first, followed by `f(a)` *again*, and finally `a`?

From the standpoint of mathematical notation, the idea that an expression such as $x+f(x)$ may be unequal to $f(x)+x$ is unsettling, but the consequence of this distinction has ramifications for the efficiency of evaluation as well. For example, the code generated for $x+f(x)$ must save x before evaluating $f(x)$ because of the (probably unlikely) prospect that the evaluation of $f(x)$ will alter the value of x. Although there might be some advantage therefore in requiring that $x + f(x)$ have the same value as $f(x) + x$, such a requirement is less clearly beneficial in some analogous cases. For instance, it would probably not be a boon for efficiency to require that $x = 0 \vee f(x) = 0$ be equivalent to $f(x) = 0 \vee x = 0$ since the evaluation of $f(x)$ might be avoided if x is zero. Moreover, it might be useful to write $x = 0 \vee f(x) = 0$ in instances where $f(0)$ is undefined.

It is now common for language definitions to explicitly allow the same program to yield different results under different implementations. Ideally, the definition describes clearly how such a situation can arise, and programmers must themselves take responsibility for writing code that will behave differently for different implementations. A typical example of this arises in the definition of the Scheme programming language. The informal description of the language indicates that the operator and operands of a procedure application are evaluated *in an indeterminate order*. Hence programmers cannot rely on the portability of their code if it depends on the choice of evaluation order made by a specific Scheme compiler. Presumably this latitude does not leave open the possibility that a given implementation is non-deterministic but does allow the possibility that the order of evaluation for an expression is different for different occurrences of the expression in a given program. Such nuances are difficult to explain rigorously and even more difficult to formalize. Indeed, the formal definition of Scheme is not quite true to its informal explanation. The following passage from the formal definition indicates the compromise in the formalization:

> The order of evaluation within a call is unspecified. We mimic that here by applying arbitrary permutations *permute* and *unpermute,* which must be inverses, to the arguments in a call before and after they are evaluated. This still requires that the order of evaluation be constant throughout a program (for any given number of arguments), but it is a closer approximation to the intended semantics than a left-to-right evaluation would be.

There has been a great deal of effort to ensure that language definitions are clear and do not admit unexpected possibilities in their interpretation. Much effort has been devoted to showing that formal methods of definition can help to achieve this. But, as the Scheme example shows, capturing the exact nuance of the intended specification in the formalism is not an easy task. There are many approaches to the formal specification of programming languages. In the rest of this section I will attempt to survey some significant classes of these.

Transition semantics.

An intuitive model of the execution of a program is to think of it in terms of a machine state consisting of a *control* part representing the instructions to be executed and a *data* part consisting of the structures being manipulated by the instructions of the program. This picture is augmented by some idea of the *point of control* indicating the instruction currently being executed. The computation then is a sequence of transitions in which the data is altered and the point of control moves through the instructions of the program. The semantics of a programming language is then described by saying how it is converted into instructions and what transitions are engendered by instructions. To make such an account more abstract, it is helpful to avoid translation into some low-level instruction set and instead work directly in terms of the syntax of the higher-level programming language. Two abstractions can now lead us to a clear and simple approach to explaining the semantics of programming languages. First, we eliminate the need for a point of control by thinking of the transitions as converting the program before the execution of an instruction into a new program to be executed afterwards. Second, we explain how such transitions are defined by following the *structure* of the programs of the language as given by its grammar and providing one or more transitions for each possible form of program.

Let us illustrate this approach by considering a simple imperative programming language. Its syntax is given as a BNF grammar in Table 1.3 where I ranges over a syntax class of *identifiers* and N ranges over a syntax class of *numerals.* The command **skip** is called *skip.* Commands of the form $C_1; C_2$ are called *sequences.* Those having the forms $I := E$ are called *assignments.* Those having the form **if** B **then** C_1 **else** C_2 **fi** are called

Table 1.3
Syntax for the Simple Imperative Programming Language

I	ϵ	Identifier
N	ϵ	Numeral
B	$::=$	**true** | **false** | B **and** B | B **or** B | **not** B | $E < E$ | $E = E$
E	$::=$	$N \mid I \mid E + E \mid E * E \mid E - E \mid -E$
C	$::=$	**skip** | $C; C$ | $I := E$ | **if** B **then** C **else** C **fi** | **while** B **do** C **od**

conditionals and those of the form **while** B **do** C **od** are called *while loops*..

The notion of the data part of a machine state will be represented simply as a function assigning integer values to identifiers. More precisely, we define the domain $\mathbb{M}$ of *memories* to be the set of functions f : Identifier $\rightarrow \mathbb{Z}$ from identifiers to the integers $\mathbb{Z}$. To simplify matters, let us assume that semantic functions for arithmetic expressions and boolean expressions are already known, so we can focus our attention on the semantics of commands. It is traditional to write the syntax of a program within 'semantic brackets' $[\![\cdot]\!]$ to separate terms of the object language (that is, the formal language whose semantics is being described) from the surrounding mathematical metalanguage (that is, the possibly informal language being used to explain the semantics of the object language). When it is important to distinguish one semantic function from another, a calligraphic letter is used to indicate the function in question, and we write something like $\mathcal{A}[\![B]\!]$ and $\mathcal{B}[\![B]\!]$ for the $\mathcal{A}$ and $\mathcal{B}$ meaning functions. When only one semantic function is under discussion, it is simplest to drop the distinguishing letter so long as no confusion is likely to arise. So, for the discussion at hand, let us assume we are given the definition of a meaning function $[\![\cdot]\!]$ on booleans and arithmetic expressions:

$$[\![B]\!] : \mathbb{M} \rightarrow \{\mathsf{true},\ \mathsf{false}\}$$
$$[\![E]\!] : \mathbb{M} \rightarrow \mathbb{Z}$$

(It can be assumed—for the sake of uniformity—that these semantic functions were also defined by a means similar to that being used here for commands.) The meaning $[\![C]\!]$ of a command C will be defined as a partial function from $\mathbb{M}$ (input) to $\mathbb{M}$ (output). To achieve the desired level of abstraction in the description of its semantics, the state

Table 1.4
Transition Semantics for the Simple Imperative Programming Language

$$(I := E, m) \to m[I \mapsto [\![E]\!]m] \qquad (\mathbf{skip},\ m) \to m$$

$$\frac{(C_1, m) \to (C_1', m')}{(C_1; C_2, m) \to (C_1'; C_2, m')} \qquad \frac{(C_1, m) \to m'}{(C_1; C_2, m) \to (C_2, m')}$$

$$\frac{[\![B]\!]m = \mathsf{true}}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, m) \to (C_1, m)} \qquad \frac{[\![B]\!]m = \mathsf{false}}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, m) \to (C_2, m)}$$

$$\frac{[\![B]\!]m = \mathsf{false}}{(\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, m) \to m} \qquad \frac{[\![B]\!]m = \mathsf{true}}{(\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, m) \to (C; \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, m)}$$

of a machine is represented as a pair (C, m) consisting of a program fragment C and a memory m recording the values of identifiers. When the control part of the program is empty (that is, the program has terminated and there are no more instructions to be executed), we simply write m. More precisely, a *configuration* is either

- a pair (C, m) consisting of a command C and a memory m, or
- a memory m.

Rules for evaluating a program of the Simple Imperative Language are given in terms of a binary relation $\to$ on configurations; this is defined to be the least relation that satisfies the rules in Table 1.4. In the transition rules for assignment, the evaluation of the command results in a new memory in which the value associated with the identifier I is bound to the value of the expression E in memory m. The notation used there is defined as follows for a value $z \in \mathbb{Z}$:

$$m[I \mapsto z](J) = \begin{cases} z & \text{if } I \text{ is the same as } J \\ m(J) & \text{otherwise} \end{cases} \tag{1.2}$$

Now, let $\to^+$ be the transitive closure of the relation $\to$ (that is, $\to^+$ is the least transitive relation that contains $\to$). It is possible to prove the following:

1.1 Lemma. *Let γ be a configuration. If $\gamma \to^+ m$ and $\gamma \to^+ m'$ for memories m and m', then $m = m'$.* □

The meaning of a program C can therefore be defined as follows:

$$[\![C]\!]m \simeq \begin{cases} m' & \text{if } (C,m) \rightarrow^{+} m' \\ \text{undefined} & \text{otherwise} \end{cases}$$

where the symbol $\simeq$ is being used rather than $=$ to emphasize that $[\![C]\!]$ is a *partial* function, which may be undefined on some memories m. (This is called *Kleene* equality, and it is used in expressing a relationship between partially defined expressions. Given expressions E and E' where one or both may be undefined, writing $E \simeq E'$ means that either E and E' are both defined and equal or they are both undefined.)

This semantic description induces a set of equations between programs if we take C to be equivalent to C' when $[\![C]\!] = [\![C']\!]$. It is a *virtue* of this equivalence that it ignores important aspects of a program such as its efficiency. It is this separation of concerns that makes it possible to develop a theory of program transformations; we could not say when the replacement of one program by another is legitimate without saying what property is to be preserved under such a replacement.

Let me turn to another example of how transition rules can be used to describe the semantics of a familiar kind of programming language construct. To keep matters simple, let us work with a very terse fragment of a language such as Lisp that allows functions to be taken as arguments or returned as values (a feature sometimes referred to as 'functions as first class citizens'). The minimal grammar is given as follows:

$$M \;::=\; x \mid (M\ M) \mid (\textbf{lambda}\ x\ M)$$

where x is drawn from a primitive syntax class of identifier names.[5] I will use letters x, y, z, and these letters with primes, subscripts, and so on for identifiers. This language is called the *untyped λ-calculus,* and its expressions are called *λ-terms.* I will use letters L, M, and N to range over λ-terms. The expression $(\textbf{lambda}\ x\ M)$ is called an *abstraction,* and it is to be thought of as a functional procedure in which x is the *formal parameter* and M is the *body* of the procedure. In particular, the scope of the identifier x is the body of the abstraction. The expression $(M\ N)$ is called an *application,* and it should be thought of as the application of the function M to the argument N.

A λ-term is *closed* if each of its identifiers falls within the scope of an abstraction. For example, the λ-term

$$(\textbf{lambda}\ x\ (\textbf{lambda}\ y\ x))$$

[5]It would be tempting to use letters such as I for identifiers here to maintain uniformity with the notation in the Simple Imperative Language. However, some upper case letters are commonly used to denote various constants in the λ-calculus. For example, I is usually used for an expression of the form $(\textbf{lambda}\ x\ x)$.

is closed, whereas

(**lambda** z (**lambda** y x))

is not closed (because the identifier x does not lie in the scope of any abstraction whose formal parameter is x). We write $[M/x]N$ to denote the result of *substituting* the λ-term M for the identifier x in the λ-term N. Substitution is a tricky business really, but I will delay its precise definition until the beginning of the detailed discussion of the λ-calculus in Chapter 2. For now let us assume we understand the concept of substitution and look at a semantics for closed λ-terms. Two rules suffice to define how such terms can be viewed as programs. The first of these says that if a step of evaluation can be carried out on the operator of an application, then this step can be done as part of the application:

$$\frac{M \to M'}{(M\ N) \to (M'\ N)}$$

The second says that the result of applying an abstraction to an argument is obtained by substituting the operand for the formal parameter in the body of the abstraction:

$$((\mathbf{lambda}\ x\ M)\ N) \to [N/x]M$$

It is not hard to show the analog of Lemma 1.1 for this relation:

1.2 Lemma. *If M is a closed λ-term such that $M \to^+ N$ and $M \to^+ N'$ for abstractions N and N', then N and N' are the same term.* □

We define the value (meaning) of a closed λ-term M as follows:

$$[\![M]\!] \simeq \begin{cases} M & \text{if } M \text{ is an abstraction} \\ N & \text{if } M \to^+ N \text{ for some abstraction } N \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is easy to prove that the value of a closed λ-term is always a closed abstraction.

The rules above define a form of semantics for the evaluation of λ-terms called *call-by-name* because the rule for application may result in an unevaluated term being substituted for the formal parameter of the operator in its body. Another common evaluation order—the one used in Lisp and many similar languages—evaluates the operand of the application before it is substituted for the formal parameter. This is known as *call-by-value* since a value is always substituted for the formal parameter of the operator. It is possible to describe this with a transition semantics and an appropriate set of rules. The description is slightly more complex than the one for call-by-name since there are three rules rather than two. The first of these rules is the same as that given before:

$$\frac{M \circ\!\!\to M'}{(M\ N) \circ\!\!\to (M'\ N)}$$

The little circle on the arrow in the rule is meant to distinguish the call-by-value evaluation relation from the call-by-name one. To simplify notation, let us write V for values (that is, abstractions). The difference between the definitions of $\rightarrow$ and $\circ\!\!\rightarrow$ comes from the fact that only a value is substituted into a function body:

$$((\mathbf{lambda}\ x\ M)\ V) \circ\!\!\rightarrow [V/x]M$$

To obtain an operand that is a value, it is necessary to first evaluate the operand:

$$\frac{N \circ\!\!\rightarrow N'}{(V\ N) \circ\!\!\rightarrow (V\ N')}$$

The definition of the meaning of an expression is defined for call-by-value in basically the same way this was done for call-by-name, but with $\circ\!\!\rightarrow$ supplanting $\rightarrow$ in the defining equation. Differences between the semantics of call-by-name and call-by-value will appear frequently in the discussions of semantics in this book. Fortunately, both call mechanisms can be described in a clear and succinct manner using the formalisms at hand.

Natural semantics.

The transition semantics described in the previous section proceeds in two steps. We begin by defining the notion of a transition relation between configurations or λ-terms. This relation is then used to define a partial function from programs to values, which is taken to be the semantics of the language. Could we combine these two steps somehow and define the desired partial function directly? One approach to this is to axiomatize the relation $\rightarrow^+$ itself. This is indeed possible; the rules for doing this appear for the Simple Imperative Language appear in Table 1.5. They define a binary relation $\Downarrow$ between configurations of the form (C, m) on the left of side of the relation symbol and memories on its right side. The following lemma describes the desired property:

1.3 Lemma. *For any program C and memory m,*

$$(C, m) \rightarrow^+ m' \textit{ iff } (C, m) \Downarrow m'. \qquad \square$$

The proof of the lemma is somewhat tedious but routine. It shows that the rules in Tables 1.4 and 1.5 define essentially the same semantics for the Simple Imperative Language. A semantics described in the manner of Table 1.5 is sometimes called a *natural* or *relational* semantics. In a natural semantics, an evaluation is very much like the search for a proof. Because of the nature of the rules for the relation given in Table 1.5, it is possible to evaluate a program C in a memory m in the following way. First find a rule for which a conclusion of the form $(C, m) \Downarrow m'$ is possible. For some forms of command C—skip, sequence, and assignment—there is only one rule that could

Table 1.5
Natural Semantics for the Simple Imperative Programming Language

$$(\textbf{skip},\ m) \Downarrow m$$

$$\frac{(C_1, m) \Downarrow m' \qquad (C_2, m') \Downarrow m''}{(C_1; C_2, m) \Downarrow m''}$$

$$(I := E, m) \Downarrow m[I \mapsto [\![E]\!]m]$$

$$\frac{[\![B]\!]m = \mathsf{true} \qquad (C_1, m) \Downarrow m'}{(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, m) \Downarrow m'} \qquad \frac{[\![B]\!]m = \mathsf{false} \qquad (C_2, m) \Downarrow m'}{(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, m) \Downarrow m'}$$

$$\frac{[\![B]\!]m = \mathsf{false}}{(\textbf{while } B \textbf{ do } C \textbf{ od}, m) \Downarrow m}$$

$$\frac{[\![B]\!]m = \mathsf{true} \qquad (C, m) \Downarrow m' \qquad (\textbf{while } B \textbf{ do } C \textbf{ od}, m') \Downarrow m''}{(\textbf{while } B \textbf{ do } C \textbf{ od}, m) \Downarrow m''}$$

possibly apply; for others—conditional and while loop—there are two possibilities. In each of the latter cases, if a conclusion of the form $(C, m) \Downarrow m'$ is indeed possible, then exactly one of these rules could be used to reach this conclusion. Both could not be applicable because of the boolean test, which cannot be both true and false for the same memory. To evaluate a program (C, m), therefore, find a rule that applies and then try to establish hypotheses for it by attempting to evaluate (find a proof for) each of the programs that appear in its hypotheses. If this effort fails because a hypothesis yields the wrong conclusion (for example, false is obtained when true is sought), then attempt the effort again with another applicable rule, if there is one. The search for a proof may, in fact, proceed without end in some cases; this is how non-termination is represented in this system. Evaluation using a natural semantics is very much like running a *logic program.* Indeed, the rules in Table 1.5 could easily be coded in a language such as Prolog. This connection to logic, in the form of natural deduction proof, is part of the reason the term 'natural semantics' is used for this form of specification.

If a relation $(C, m) \Downarrow m'$ can be established, then it can be shown that there is a unique proof of this fact. This property might be different if the language contains other features. For example, if we add a non-deterministic construct C_1 **or** C_2 that evaluates either C_1 or C_2, its semantics could be given through the following pair of rules:

$$\frac{(C_1, m) \Downarrow m'}{(C_1 \textbf{ or } C_2, m) \Downarrow m'} \qquad \frac{(C_2, m) \Downarrow m'}{(C_1 \textbf{ or } C_2, m) \Downarrow m'}$$

Either of the two rules could be used for reaching a conclusion about the result of evaluating an expression, so more than one outcome is possible.

In some ways, the natural semantics of the λ-calculus is simpler to describe than its transition semantics. Abstractions evaluate to themselves (since they are already values):

$$(\textbf{lambda } x\ M) \Downarrow (\textbf{lambda } x\ M)$$

and, for call-by-name, if the value of the operator is established, then the operand is substituted for the formal parameter and the result is evaluated:

$$\frac{M \Downarrow (\textbf{lambda } x\ M') \qquad [N/x]M' \Downarrow V}{(M\ N) \Downarrow V}$$

It can easily be seen from the rules that if $M \Downarrow U$ and $M \Downarrow V$, then U and V are the same. We therefore define the value of M to be the unique V, if there is one, such that $M \Downarrow V$. If there is no such V, then the value of M is undefined. As with the natural semantics for the simple imperative language, this definition is exactly the same as that given using transitions.

The call-by-value evaluation order has an identical rule for abstractions:

$$(\textbf{lambda } x\ M) \Downarrow^{\circ} (\textbf{lambda } x\ M)$$

but in the rule for application, it is required that the value of the operand be substituted for the formal parameter in the function body:

$$\frac{M \Downarrow^{\circ} (\textbf{lambda } x\ M') \qquad N \Downarrow^{\circ} U \qquad [U/x]M' \Downarrow^{\circ} V}{(M\ N) \Downarrow^{\circ} V} \tag{1.3}$$

Compositionality, fixed points, and denotation.

A common way to give the semantics of a language is to define semantic functions that describe how the semantics of an expression of the language can be obtained from the semantics of its syntactic components. The meaning of an expression can then be obtained by composing the semantic functions determined by its syntactic structure. A semantics given in this way is said to be *compositional.* Compositionality is a property common to the forms of semantic description used in most areas of logic. Consider, for example, the way the Tarski's semantics of first-order logic is given. If ϕ and ψ are first-order formulas and ρ is an assignment of meanings (in the universe of the model), then $[\![\phi \wedge \psi]\!]\rho$ is true if, and only if, $[\![\phi]\!]\rho$ and $[\![\psi]\!]\rho$ are both true. For a identifier x, $[\![\forall x.\ \phi]\!]\rho$ is true if, and only if, for every element a of the model, $[\![\phi]\!](\rho[x \mapsto a])$ is true. Other meanings are defined similarly.

By contrast, in both the transition and natural semantics as described above for the Simple Imperative Language, the meanings of programs were not described with such meaning functions. However, at least for some expressions, it seems that such a meaning function would not be hard to obtain. Let us look now at how we might go about giving a compositional description of the semantics of the Simple Imperative Language. As before, the meaning of a command C is a partial function $[\![C]\!]$ on memories. Clearly $[\![\mathbf{skip}]\!]m = m$. Sequencing is interpreted as composition of partial functions, $[\![C_1; C_2]\!]m \simeq [\![C_2]\!]([\![C_1]\!]m)$ and assignment is updating, $[\![I := E]\!]m \simeq m[I \mapsto [\![E]\!]m]$. We interpret branching by

$$[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}]\!]m \simeq \begin{cases} [\![C_1]\!]m & \text{if } [\![B]\!]m = \mathsf{true} \\ [\![C_2]\!]m & \text{if } [\![B]\!]m = \mathsf{false} \end{cases}$$

So far, so good. However, when we write down the straightforward explanation of the meaning of the loop, a problem appears:

$$[\![\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}]\!]m \simeq \begin{cases} m & \text{if } [\![B]\!]m = \mathsf{false} \\ [\![\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}]\!]([\![C]\!]m) & \text{if } [\![B]\!]m = \mathsf{true} \end{cases}$$

In particular, the expression **while** B **do** C **od** we are trying to define appears on *both* sides of the equation.

What is such an equation intended to define? One view holds that

$$[\![\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}]\!]$$

is some form of *canonical fixed point* of an operator defined by the expression on the right-hand side of the equation. In the case above, the loop construct is defined to be such a fixed point of the function $F : [\mathbb{M} \rightsquigarrow \mathbb{M}] \to [\mathbb{M} \rightsquigarrow \mathbb{M}]$ where

$$F(f)(m) = \begin{cases} m & \text{if } [\![B]\!]m = \mathsf{false} \\ f([\![C]\!]m) & \text{if } [\![B]\!]m = \mathsf{true} \end{cases}$$

and $[\mathbb{M} \rightsquigarrow \mathbb{M}]$ is the set of *partial* functions between memories. Now

$$[\![\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}]\!] = \mathsf{fix}(F)$$

really *is* a compositional description because the definition of F was made in terms of $[\![C]\!]$ and $[\![B]\!]$ only. This raises two questions: How do we know that such a fixed point exists, and, if it does, is there a way to choose it canonically?

There are several theories about how to solve equations in order to provide meanings for recursive definitions. The typical procedure is this. One defines a class of spaces in which programs take their meanings, and basic operators are interpreted as functions between such spaces. A recursive definition determines an operator $F : D \to D$ on such a space D; the term being defined is interpreted as a fixed point of F. In some theories the fixed point is unique; in others, the semantics is based on a choice of canonical fixed point for an operator. A way to choose a value canonically is to specify some property of the desired fixed point that uniquely specifies it. For instance, it may be the least one, the greatest one, the 'optimal' one (for some definition of optimality), or whatever might be appropriate for the purpose that the semantics is intended to serve. Members of the class of spaces over which recursive definitions are interpreted are usually called *domains.* The term originated from the idea that these spaces are the domains of the operators F used to make recursive definitions. The study of structures appropriate for such applications is called *domain theory.*

In this book, the use of domain theory and fixed points to interpret programs compositionally will be called *fixed-point semantics.* The technique goes by other names as well; it is often called a *denotational* semantics. This nomenclature is motivated by the idea that domains provide a realm of abstract values and that some of these values are denoted by programs. The usage is somewhat vague and sometimes misleading, though. For example, the transition and natural semantics of the Simple Imperative Language appear to have this characteristic (programs denote partial functions on memories). However, to appreciate the terminology better, consider the semantics of the λ-calculus. What mathematical object does a λ-term denote? Looking at the natural semantics, one might begin with the idea that V is the denotation of the term M if $M \Downarrow V$. But if this is what is meant by a denotation, then the semantics is a somewhat trivial one—after all, an abstraction denotes itself! This is far from our intuition about the meanings of such terms in which an abstraction is some kind of function. The domain of such a function might be the λ-terms themselves, but this idea is not *a priori* present in the rules for the natural semantics. In short, a further theory is required to provide a serious semantics in which λ-terms denote mathematical objects. To emphasize this distinction, natural semantics and transition semantics are usually said to be forms of *operational* (as opposed to denotational) semantics.

Abstract machines.

This survey of forms of semantic description began with a discussion of the limitations of abstract machines. But how do abstract machines differ from the kinds of semantics described above? For example, the transition semantics for the Simple Imperative Language could be mechanically executed to provide an interpreter for the Simple Imperative Language or for the λ-calculus. By an appropriate form of search, the natural semantics for these languages could also be viewed as an interpreter. It is less clear that a denotational semantics has computational content, but in many cases the semantic equations of a denotational semantics can be viewed as a program in a functional language whose execution serves as an interpreter for the language whose semantics it defines. (For this reason, languages such as Scheme and ML have often been used as notations for denotational specifications.) There are gray areas between the classification of forms of semantic specification given above. While overstating differences can mask important relationships, it is important to recognize distinctions in order to understand how semantic descriptions can serve different purposes.

The term 'machine' ordinarily refers to a physical device that performs a mechanical function. The term 'abstract' distinguishes a physically existent device from one that exists in the imagination of its inventor or user. The archetypical abstract machine is the *Turing* machine, which is used to explain the foundations of the concept of mechanical computability. The idea behind the Turing machine is that it is so concrete that it is evidently possible to execute its actions in a straightforward, mechanical manner while also being abstract enough that it is simple to prove theorems about it. The next action of a Turing machine is entirely determined by the value that it is currently examining, its current state, and the transitions that constitute its program. By contrast, the transition semantics for the Simple Imperative Language involved an additional idea: that transitions can also be specified by rules having hypotheses that use the transition relation itself. This use of rules introduces an implicit stack if one is to think of the transition relation mechanically. To execute one step in the evaluation of $(C_1; C_2, m)$, it is necessary to find a configuration γ such that $(C_1, m) \rightarrow \gamma$. If C_1 has the form $C_1'; C_1''$, then it will be necessary to look for a configuration γ' such that $(C_1', m) \rightarrow \gamma'$. If, eventually, the desired hypotheses are all established, then the evaluation of the configuration $(C_1; C_2, m)$ moves to the next appropriate configuration. In the case of a natural semantics, the details of evaluation can be even more implicit. For example, in seeking a value V in the conclusion of the call-by-value application rule 1.3, it has not been explicitly indicated whether the value of M should be sought first, whether this should instead be preceded by a search for the value of N, or whether these two searches might be carried out in parallel. Each approach would lead to the same answer in the end, but the semantic rule is somewhat abstract in omitting details of how the evaluation should be done mechanically. The details of how a fixed-point interpretation is given for the Simple Imperative Language were not provided, but the choice of a canonical fixed point to solve a recursive equation does not in itself indicate how the recursively defined term is to be evaluated. This level of abstraction has been offered as a virtue of fixed-point semantics, but the view is controversial since the defining semantic equations can sometimes be viewed as implicitly describing a particular evaluator.

To make this discussion more concrete, I will briefly describe one of the best-known examples of an abstract machine, the *SECD machine* of Landin. The machine carries out call-by-value evaluation of closed terms of the λ-calculus. It is simple enough that its basic instructions can be easily implemented on standard computer architectures; in particular, the operations of the machine explicitly carry out all needed stack manipulations. It can be described in many different ways, but one approach fairly similar to the way I have treated other semantics is to use λ-terms themselves as part of the instruction set of the machine. The acronym SECD stands for the typical names for the four components of the machine configuration, which is a four tuple (S, E, C, D), where S is the *Stack*, E is the *Environment*, C is the *Code*, and D is the *Dump*. The action of the machine is described by a collection of transitions driven by the next instruction in the code C. An *instruction* for the machine is either a λ-term M or a constant **ap**. The code is a list of instructions; let us write $M :: C$ for the list whose head is M and tail is a list C, and **nil** for the empty list. To describe a stack, we must define the notions of closure and environment. Each is defined in terms of the other: an *environment* E is a partial function that assigns closures to identifiers, and a *closure* is a pair (M, E) consisting of a λ-term M and an environment E. There is also a restriction that the environment in a closure is defined on all of the identifiers of M that are not in the scope of an abstraction; this property is implicitly maintained by the transitions of the SECD machine. Let us simply write $E(x)$ for the value of E on x and $E[x \mapsto Cl]$ for the function that is the same as E except for having value Cl on argument x. (This notation is the same as the one used for assignments in Equation 1.2 of the semantics of the Simple Imperative Language.) Now, the component S is a stack of closures, which we can represent as a list where the head of the list serves as the top of the stack. Finally, a dump is either **empty** or it is a four tuple (S', E', C', D'), where S' is a stack, E' an environment, C' a code (instruction list), and D' another dump.

The SECD machine is described by a collection of transitions that cover all of the cases that can arise starting from a tuple $(\mathbf{nil}, \emptyset, M, \mathbf{empty})$ where M is a closed λ-term. The list of transition rules appears in Table 1.6. Here are some notes on the rules:

- If there are no further instructions in the command component, then the dump is consulted. The end of the evaluation of a term using the machine is reached when the dump is **empty** and there are no further commands to be executed. In this case the final result of the evaluation is the closure on the top of the stack.
- If an identifier is next on the command list, then it is looked up in the environment and placed on the stack. An identifier appearing in this way will always be in the domain of definition of the environment E.

Table 1.6
Transition Rules for the SECD Machine

$$(Cl :: S,\ E,\ \mathbf{nil},\ (S', E', C', D')) \overset{\text{SECD}}{\longrightarrow} (Cl :: S',\ E',\ C',\ D')$$

$$(S,\ E,\ x :: C,\ D) \overset{\text{SECD}}{\longrightarrow} (E(x) :: S,\ E,\ C,\ D)$$

$$(S,\ E,\ (\mathbf{lambda}\ x\ M) :: C,\ D) \overset{\text{SECD}}{\longrightarrow} (((\mathbf{lambda}\ x\ M), E) :: S,\ E,\ C,\ D)$$

$$(((\mathbf{lambda}\ x\ M), E') :: Cl :: S,\ E,\ \mathbf{ap} :: C,\ D)$$
$$\overset{\text{SECD}}{\longrightarrow} (\mathbf{nil},\ E'[x \mapsto Cl],\ M,\ (S, E, C, D))$$

$$(S,\ E,\ (M\ N) :: C,\ D) \overset{\text{SECD}}{\longrightarrow} (S,\ E,\ N :: M :: \mathbf{ap} :: C,\ D)$$

- If an abstraction is the next command, it is paired with the environment E to form a closure and put on the top of the stack. The invariants of the evaluation relation will ensure that the pair built in this way is indeed a closure.
- If an application is next on the command list, it is broken into three instructions that replace it at the beginning of the command list. The idea is that the operator and operand must be evaluated before the actual application can take place.

The thing to notice about this SECD semantics is that there are no rules with hypotheses, just transitions. The representation of a term includes its own stacks for holding intermediate values during computation. At the opposite extreme, a natural operational semantics suppresses as much detail about this aspect of computation as possible. Instead of using rewriting as the driving force of computation, it uses search for a proof. The transition semantics provides a compromise between computation as rewriting and computation as proof search. If we think of transitions as a 'horizontal' kind of computation (represented as a sequence of arrows from left to right) and search as a 'vertical' kind of computation (represented as a tree), then the difference between abstract machines, transition semantics, and natural semantics can be graphically illustrated as in Figure 1.1.

Exercises.

1.1 Show that various other orders of evaluation for the ALGOL 60 program in Table 1.2 yield other possible results: $\frac{3}{5}$, $\frac{3}{2}$, $\frac{5}{2}$, $\frac{4}{3}$, $3\frac{3}{5}$, $3\frac{1}{3}$, $5\frac{3}{5}$, $3\frac{1}{2}$, and $7\frac{1}{2}$. (Suggestion: represent the expression $a + f(a, g(a))/g(a)$ as a tree and demonstrate the results obtained from various orders in which its nodes could be calculated.)

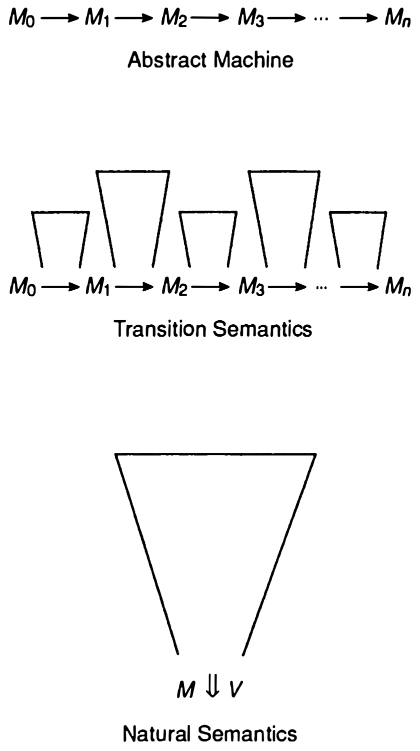
$M_0 \longrightarrow M_1 \longrightarrow M_2 \longrightarrow M_3 \longrightarrow \cdots \longrightarrow M_n$
Abstract Machine
$M_0 \longrightarrow M_1 \longrightarrow M_2 \longrightarrow M_3 \longrightarrow \cdots \longrightarrow M_n$
Transition Semantics
$M \Downarrow V$
Natural Semantics

Figure 1.1
Three Forms of Operational Semantics

1.2 Alter the Simple Imperative Language to replace the while command with a command **repeat** C **until** B **end**, which tests the boolean *after* executing the command C rather than *before* as the while command does. For the new command, provide

a. a careful informal description of its semantics,
b. a transition semantics,
c. a natural semantics.

1.3 Augment the λ-calculus with a non-deterministic choice construct. Give a grammar for your language and describe its transition semantics and natural semantics.

1.4 Formulate an analog of Lemma 1.3 for the Simple Imperative Language augmented by a non-deterministic choice construct C_1 **or** C_2.

1.5 Write a non-terminating program in the Simple Imperative Language and study how it runs in a transition or natural semantics. Carry out the same exercise for a non-terminating program in the call-by-value λ-calculus using transition, natural, or SECD semantics.

1.6 Give an example of a program in the Simple Imperative Language augmented by a non-deterministic choice construct C_1 **or** C_2 that has infinitely many possible outcomes. Show that your program can also diverge.

1.7* Formulate and prove a general conjecture about the phenomenon in Exercise 1.6.

1.8 Code the SECD machine in your favorite programming language.

1.9 Modify the definition of the SECD machine so that the evaluation is call-by-value rather than call-by-name.

1.3 Notes

The word 'semantics' was introduced by Micheal Bréal in 1897 and it is the subject of his book [35] on how words change their meanings. It is the most commonly used of a collection of terms derived from the Greek word *sēmainō* for 'to mean'. Some examples are: semiotic, semology, semasiology, and semiology. The last refers to the general subject of the relationship between signs and their meanings, whereas 'semantics' refers primarily to linguistic meaning.

Examples such as the Prolog and Scheme encodings of the gcd function given at the first part of the chapter and the programs for the Fibonacci function given later fill the pages of books that discuss programming methodology for these languages. Books by Sterling and Shapiro [249] and by Abelson and Sussman [2] are well-known examples for Prolog and Scheme respectively. The idea of viewing imperative programs as relations between properties of input and output states has been widely promoted in work of Hoare [116] and Dijkstra [73]. A discussion of the use of such relations together with hosts of references can be found in the survey article by Cousot [61]. Some works that have examined their use in higher-order languages include papers by Tennent [260], by Clarke, German, and Halpern [51; 85], by Goerdt [91], and a doctoral dissertation by O'Hearn [187].

There is a two-volume book by Lyons [153] on the semantics of natural language, and a variety of collections discuss the semantics of natural language and the philosophy of language [67; 135; 214; 248; 273]. There is a substantial literature on semantics in mathematical logic, an area generally known as *model theory.* A comprehensive bibliography is [229]. A standard reference on first-order model theory is the book by Chang and Keisler [50].

Attempts to formalize mathematics have been greatly intensified by the introduction of computers, since formalized proofs can be checked by machine. One successful effort in this direction is the Automath project of de Bruijn *et al.* [69], and another, which focuses on proofs in intuitionistic logic, is Nuprl project of Constable *et al.* [52].

Ever since the introduction of 'formal methods' for proving properties of programs, there has been a heated debate over their usefulness in practice. A well-known argument against formal methods was given in a paper by De Millo, Lipton, and Perlis [70], and a critique of this argument can be found in a paper by D. Scott and Scherlis [219]. A classic discussion of formality in mathematics that has not lost its relevance despite its age is Frege's book on arithmetic [81]; a translation of the work into English is [82].

Gödel's Incompleteness Theorem on the undecidability of truth for arithmetic has been clearly described in many sources. One such account appears in Boolos and Jeffrey [33]. Real closed fields are discussed in a paper by Ben-Or, Kozen, and Reif [23] where it is also possible to find further references.

An article by C. Gunter [98] drawn from this chapter discusses the terminology and history of some of the forms of denotational specification discussed in Section 1.2. The excerpt in Table 1.1 is taken from page 10 of the ALGOL 60 Revised Report [184]. The original report [183] appeared in 1960. The discussion of ALGOL 60 in Section 1.2 draws Exercise 1.1 and most of its examples from the 1967 article [139] of Knuth; his quote in Section 1.2 comes from the introduction of that article (page 611). The definition of the language Scheme [253] appeared first in 1975 followed by a revised report [254] three years later. The discussion of Scheme here is based on a revision to the revision, which is descriptively entitled *The revised^3 report on the algorithmic language Scheme* [199]. The quote is taken from the section on formal semantics in that article. A discussion of problems in the specification of the Pascal programming language appears in a paper of Welsh, Sneeringer, and Hoare [271].

Detailed treatments of the various forms of semantics for the Simple Imperative Language and some of its extensions can be found in books by Hennessy [111], Tennent [261], and Hanne and Flemming Nielson [185]. A substantial example of the use of natural semantics in programming language specification is the formal semantics for the Standard ML programming language given by Milner, Tofte, and Harper [173]. A commentary to aid the reader in understanding this definition is [172]. A paper [106] by Hannan and Miller discusses formal relationships between various forms of operational semantics. Fixed-point (denotational) semantics has its origins in the work of D. Scott and Strachey [230; 252]. An influential book on the topic was written by Stoy [252] in the 1970's. A more up-to-date exposition is the book by Schmidt [220]; it focuses on the denotational semantics of programming languages and has a substantial bibliography of further references. There is a handbook article on on denotational semantics by Mosses [179] and one on domain theory by C. Gunter and D. S. Scott [101]. The SECD machine was introduced by Landin [145; 146]. A good treatment of various extensions of the machine in this chapter can be found in the book of Henderson [109]. Some other approaches to specifying the semantics of programming languages include the Vienna Development Method, which is described by Wegner in [270], the evolving algebras of Gurevich [104], and the action semantics of Mosses [180].

The idea of automatically generating a compiler from a formal specification has been an area of active study. Pointers to most of the literature on the subject can be found in the book of Lee [151].