C. Ghezzi and M. Jazayeri.
Programming Language Concepts, 2/E.
Wiley, 1987.

9

An Introduction to Formal Semantics

According to the definition given in Chapter 3, the semantics of a language describes "the meaning" of any syntactically correct program in that language. In Chapter 3 we described the meaning of basic language concepts, such as variables, bindings, and run-time structures, in terms of a virtual machine. That approach is, what we have called *operational semantics.*

In this chapter we introduce two formal—that is, mathematical—ways of defining the semantics of a programming language: *axiomatic semantics* and *denotational semantics.* Our purpose is to illustrate the spirit of how language semantics can be defined formally, and how the two styles differ in flavor from one another and from operational semantics. We do not intend to cover the details of the definition of complete and realistic programming languages. We also warn the reader that treating additional constructs and details of real programming languages is not simply a matter of adding more details to the formal description. Depending on the nature of the construct or detail, new problems may arise that may require new and perhaps as yet unknown mathematical solutions.

Also, we restrict our attention to the formal semantics of imperative languages, which constitute the main body of this text. Dealing with nontraditional languages would require the development of additional mathematical background, which is out of the scope of this text.

9.1 THE NEED FOR FORMAL SEMANTICS

Why is formal semantics useful? Why is a natural language description, such as the Pascal Report, or the Ada Reference Manual, or an operational view such as that presented in Chapter 3, inadequate? Informal natural language descriptions, even if they refer to well-understood abstract concepts, such as the SIMPLESEM virtual machine of Chapter 3, lack precision; very often they are ambiguous, incomplete, and inconsistent. Worse yet, determining that such a definition is unambiguous, complete, and consistent is a nontrivial and certainly not a mechanical task.

These are the most important benefits of formal semantics:

1. Rigorous and unambiguous definition. The language that is used to formally specify semantics, called *metalanguage,* is based on well-understood and simple mathematical concepts. This is similar to what we have seen for formal descriptions of the syntax of programming languages. BNF or syntax diagrams are the metalanguages used to specify the syntax of programming languages. In our case, giving semantics to language L by using metalanguage M can be viewed as translating from L to M. If M is a complex, nonmathematical language, we would have a circular problem, because M itself would require a formal specification of its semantics. Instead, no such need arises when M is simple and based on mathematical concepts. The resulting description is rigorous and unambiguous.

2. Basis for language comparison. The ability to compare different features, perhaps in different languages, to choose a language or evaluate alternative design decisions is often an elusive goal. The operational model developed in Chapter 3 is certainly of great help, but one hopes that formal semantics provides an even more detailed and precise basis for comparisons.

3. Independence from implementation. When a concept is described in terms of an implementation—be it abstract or concrete—one uses a number of exogenous, ancillary, implementation-dependent concepts. These do not deal with the essence of the concept, which is independent of any implementation. To circumvent this problem, one might say that a given implementation represents the equivalence class of all correct implementations (i.e., implementations that give the same results for any program and any given input). For example, our SIMPLESEM machine described in Chapter 3 uses static links to access nonlocal environments, and details have been given to show how static links are installed and used at run-time. Exercise 3.21 shows another implementation in terms of a "display." The two implementations belong to the same equivalence class of correct implementations of nonlocal environments. The essence of the concept they implement is the same, but numerous (and different) implementation-dependent details must be given in the specification. In conclusion, implementation-based semantic definitions, such as the operational definition of Chapter 3, cannot distinguish between what is the pure essence of a language concept and a specific implementation: they lack abstraction and easily can become cluttered with details.

4. Basis for correctness proof of implementation. A language implementation is defined correct if it conforms to the formal semantic specification. The formal description is *the* reference for solving any controversial issue. Implementations may differ from one another as far as efficiency is concerned, but the meaning of syntactically correct programs remains the same.

5. Basis for program correctness proofs. To prove that a given program is correct with respect to a given specification, the effect of the program must be clearly understood, that is, the language semantics should be specified rigorously. For example, as we have seen in Chapter 4, the original Pascal Report does not define variant records precisely. As a result, one may be unable to understand precisely the meaning of a given program that uses variant records, and thus to prove whether the program meets its specification.

Points 4 and 5 also show that formal semantics have the potential of providing mechanical support to correctness proofs. The only way for a computer to aid in the verification of a language implementation or the correctness of a program is to start from a precise, formal language definition.

Having seen the advantages of formal semantics, one wonders whether there is a need for informal semantic descriptions, such as those of traditional language standards. In practice, both formal and informal descriptions are needed. The interplay between formal and informal semantics is as follows: When a language is being designed, one usually starts from an informal definition. The informal description then is converted into a formal description, and this may uncover inconsistencies, ambiguities, and incompletenesses in the informal description. The process is iterated until the design is satisfactory. The final formal semantics description then is used as a basis for deriving more informal reports on the language. These definitely are needed in practice. Because they are more readable then their formal counterparts, the reader only needs to refer to the formal definition when the informal documents do not provide a satisfactory answer. Also, the practical use of the language should not require as a prerequisite the mathematical skills needed to manage formal semantics.

9.2 STYLES OF FORMAL SEMANTICS

As we have seen in the previous section, formal semantics of a programming language maps every syntactically correct language construct into a metalanguage that is based on a well-understood mathematical notation. Consequently, formal semantics can be specified as a set of translation rules from the domain of language constructs to the range of well-formed formulas of the formalism.

We distinguish between two kinds of formal semantics: axiomatic semantics and denotational semantics. *Axiomatic semantics* describes the meaning of each syntactically correct program by associating to it properties of variables (in terms of predicate calculus) that hold before execution starts and after the program halts. Thus the metalanguage of axiomatic semantics is a logic language, such as predicate calculus. *Denotational semantics* describes the meaning of a program by associating to it a function from the input domain to the output domain. In this case the metalanguage is that of functional calculus.

The fundamental concepts needed to model the semantics of a programming language are the *state of computation*, and how this state is *transformed* by various language constructs. In the informal operational semantics of Chapter 3, the state was described in terms of data structures of the abstract SIMPLESEM processor, and state transformations were described in terms of changes to the processor's data structures. In Sections 9.3 and 9.4 we will see how this can be done formally.

The rest of this chapter is organized as follows: Our discussion of formal semantics starts in Section 9.3 with axiomatic semantics, which is highly intuitive and does not require deep mathematical knowledge to be appreciated in practice. In Section 9.3.1 we also will give a glimpse of how axiomatic semantics support formal proofs of program correctness. Section 9.4 will outline the basics of denotational semantics. Finally, in Section 9.5 we will contrast the two styles and draw some conclusions.

9.3 AN INTRODUCTION TO AXIOMATIC SEMANTICS

Axiomatic semantics is based on mathematical logic. The state of computation is described by a logical expression (called a *predicate or assertion*) on program variables that must be true in that state. Intuitively, whereas in operational semantics the state of computation for a program is determined by the data structures of the modeling machine, in axiomatic semantics it is determined by a predicate on the values of the program variables.

Because in this chapter we are only interested in the flavor of formal semantics, our use of the concepts of logic will be rather informal and intuitive. To go deeper into the subject, the reader needs a background of mathematical logic, which is not required to understand the spirit of the method and its use. The Further Reading Section at the end of the chapter provides references to the literature on this subject.

A predicate P that is required to hold after a statement S is called a *postcondition* for S. A predicate Q such that the execution of S terminates and postcondition P holds upon termination is called a *precondition* for S and P. For example, $y=3$ is one possible precondition for statement $x:=y+1$, that leads to postcondition $x>0$. The predicate $y\geq 0$ is also a precondition for statement $x := y+1$ and postcondition $x>0$. Actually, $y \geq 0$ is the *weakest precondition*, that is, the *necessary and sufficient* precondition for statement $x := y+1$ that leads to postcondition $x>0$. A predicate W is called the weakest precondition for a statement S and a postcondition P, if any precondition Q for S and P implies W, that is, W holds for any precondition Q. Among all possible preconditions for statement S and postcondition P, W is the weakest: it specifies the fewest constraints. If we write implication as "$\Rightarrow$", for our example we have

$$y=3 \Rightarrow y \geq 0$$

In general, given an assignment statement $x := E$ and a postcondition P, the weakest precondition is obtained by replacing each occurrence of x in P with expression E. We express this weakest precondition with the notation $P_{x \rightarrow E}$. In the example, $P_{x \rightarrow y+1}$ is $y+1>0$, that is, $y \geq 0$.

To characterize the semantics of a programming language, we would like to describe the effect of each language construct in terms of the transformation of predicates it implies. To this end, we will define a function *asem* (for axiomatic *semantics*, also called a *predicate transformer*) that for any statement S and any postcondition P, has as its value the weakest precondition W. It is written as

$$asem\,(S, P) = W$$

In the case of an assignment statement $x := E$, we have

$$asem\,(x := E, P) = P_{x \rightarrow E}$$

This characterization of assignments is correct under the assumption that the evaluation of the right-hand side of the statement does not have any side-effects. Moreover, the variable being assigned cannot be an alias of other program variables. In such cases the execution of the assignment also may affect variables not appearing at the left-hand side of the statement, and the given semantics would be incorrect. This is an example of how side-effects and aliasing can complicate formal as well as informal reasoning about programs.

side-effect + aliasing: example on p. 324

Let us give an intuitive explanation of how *asem* (S,P) can be described as giving the semantics of S. Consider how we can use the function we have just given for assignment statements. Suppose we want to know whether the statement $x := x+1$ will produce the result $x>5$. The function *asem* can be used to tell us that $asem(x := x+1, x>5) = x>4$, that is, as long as we start with a computation state that ensures the truth of $x>4$, then the given assignment statement will achieve the desired effect. In other words, we can predict the exact effect or meaning of statements if we have the *asem* function.

Simple statements, such as assignment statements, can be combined into more complex actions by statement-level control structures. Therefore, *composition rules* are needed to characterize the semantic effect of combining individual statements into a program segment. For example, for sequencing, if we know that

$$asem\,(S1, P) = Q$$

and

$$asem\,(S2, Q) = R$$

then

$$asem\,(S2; S1, P) = R$$

The cases of selection and iteration are more complex. If B is a boolean expression and $L1$, $L2$ are two statement lists, then let **if**-stat be the following statement:

```
if B then L1 else L2 fi
```

If P is the postcondition that must be established by if-stat, then the weakest precondition is given by

$$asem(\text{if-stat}, P) = B \Rightarrow asem(L1, P) \textbf{ and not } B \Rightarrow asem\,(L2, P)$$

For example, given the following program fragment (x, y, and *max* are integers)

```
if x>= y then max:= x else max:= y fi
```

and the postcondition

$$(max=x \textbf{ and } x \geq y) \textbf{ or } (max=y \textbf{ and } y>x)$$

the weakest precondition is easily proven to be *true*(*), that is, the statement satisfies the postcondition without any constraints on variables.

Suppose now that P is the postcondition that must be established by the following loop

```
while B do L od
```

where B is a boolean expression and L is a statement list. The problem is that we do not know how many times the body of the loop is iterated. Indeed, if we knew, for example, that the number of iterations were n, the construct would be equivalent to the sequential composition

```
L;L; . . . ; L
```

of length n. Thus the semantics of the statement would be straightforward.

To overcome this difficulty, let us first reconsider why we have chosen to refer to weakest preconditions in our formalization of semantics. The reason is that weakest preconditions give an *exact* characterization of seman-

(*) B ⇒ asem(max:=x, (max=y and x≥y) or (...)) and (not B ⇒ asem(max:=y, (...)))
x≥y ⇒ [x=x and x≥y] or ... and y>x ⇒ [y=y and y>x] or ...]
true true

tics. If we choose to refer to any other preconditions, they provide only an *approximate* characterization of semantics: they are only a *sufficient* precondition that can be derived for a given statement and a given postcondition. In fact, if any precondition for a given statement and a given postcondition holds, then the weakest precondition also holds. In other words, the constraints on the state specified by a nonweakest precondition are stronger than what is needed to ensure that a certain postcondition holds after execution of a statement. If we relax our requirement and accept nonweakest preconditions as an (approximate) specification of semantics, here is how **while** statements can be handled.

Given the **while** statement and a postcondition P, we wish to determine a precondition Q for the **while** statement and predicate P. Q must be such that

(a) The loop terminates.
(b) At loop exit, P holds.

Thus predicate Q can be written as $Q = T$ and R, where T implies termination of the loop and R implies the truth of P at loop exit. Determining two predicates T and R that satisfy these properties is not straightforward and requires ingenuity. For simplicity's sake, we ignore the problem of termination and focus our attention on inventing R. Suppose we are able to identify a predicate I that holds both before and after each loop iteration and, when the loop terminates (i.e., when the boolean expression B is false), I implies P. I is called an *invariant predicate* for the loop. Formally, I satisfies the following conditions.

$$\text{(i)}\quad I \text{ and } B \Rightarrow \text{asem}(L, I)$$
$$\text{(ii)}\quad I \text{ and not } B \Rightarrow P$$

If we are able to identify a predicate I that satisfies both i) and ii), then we can take I as the desired predicate R, because P holds upon termination if $R = I$ holds before executing the loop.

In conclusion, the method of loop invariants allows us to approximate the evaluation of semantics of a **while** statement; the precondition is one possible valid precondition, not necessarily the weakest. Because the major use of axiomatic semantics is in providing programs correct (or deriving correct programs), as we will see in the next section this inconvenience does not cause us much trouble.

9.3.1 Axiomatic Semantics and Program Correctness

An in-depth study of the issues involved in the study of program correctness is beyond the scope of this book. However, we are now in a position to state precisely what we mean by *a correct program*, and to give a glimpse of how programs can be proven correct. We also will mention an interesting methodical approach that uses axiomatic semantics to guide in the derivation of programs that are correct in the first place.

First of all, the correctness requirements of the program must be specified formally by giving two predicates: a precondition IN (or *input assertion*) on input variables and a postcondition OUT (or *output assertion*) on input and output variables. The job of verification is to show that if IN holds before executing the program, execution terminates in a state where OUT holds. This proof requires the use of the semantic characterization of statements such as the one described in the previous section.

Program verification is illustrated here with the aid of a simple example. Consider the following program fragment, in which all variables are assumed to be integers.

```
i:= k; sum:= k;
while i> 1 do
      i:= i-1;
      sum:= sum+ i
od
```

Let the input assertion be

$$\text{IN}: k > 0$$

and let the output assertion be

$$\text{OUT}: sum = \sum_{1}^{k} j$$

We want to prove the fragment correct with respect to IN and OUT.

[illegible]

The termination of the loop is obviously assured, because variable i is altered only by instruction $i := i - 1$ and, thus, assumes a decreasing sequence of values, which would eventually make "$i > 1$" false. For the invariant, the predicate

$$\text{Error!}\ \dashrightarrow\ \text{I}: sum = \sum_{\underset{i}{1}}^{k} j \text{ and } 0 < i \leq k$$

can be proven easily to satisfy conditions i) and ii) of the previous section (we leave this proof to the reader). Thus, starting the execution of the loop with variables satisfying I assures termination in a state satisfying OUT. Finally, it is also easy to prove that I holds after the instructions

```
i:= k; sum:= k;
```

[illegible]

if the precondition $k > 0$ holds before the two instructions. In conclusion, if IN holds before executing the fragment, execution terminates in a state

[illegible]

where OUT holds, that is, the program is correct with respect to IN and OUT.

Given the input and output assertions, program verification proceeds by deriving *intermediate assertions* that must be proven to hold at various points in the program. In particular, as we have seen, intermediate assertions must be supplied by the verifier (human or machine) in the form of loop invariants. Other examples of intermediate assertions are the legality assertions generated by the Euclid compiler, which we mentioned in Chapter 5. In addition, Euclid allows the programmer to specify **assert** statements in the program. The Euclid's **assert** statement is used to supply intermediate assertions to be proven by the verifier as an integral part of the program. In this way, assertions become part of the documentation of the program. Moreover, compiler options allow the programmer to transfom assert statements into run-time checks during the testing phase or, alternatively, to suppress their evaluation. Finally, if a verifier is part of the set of tools provided by the programming environment, assertions can be proven by the verifier and the program is fully certified statically.

Euclid's assert statement

The mechanical verifier cannot proceed in a purely automatic fashion, but must interact with the user. In particular, the user must use ingenuity to invent loop invariants that are needed in the derivation of a precondition for a given program containing loops and a given output assertion.

mechanical verifier + loop invariants

The influence of programming language features on the process of reasoning about programs is felt both by human readers and mechanical program verifiers. Many features that make reasoning about programs difficult for humans also are hard to deal with for a program verifier. For example, side-effects in functions complicate the evaluation of the weakest precondition for assignments, as the following example shows. Let $y := f(x) + z$ be an assignment statement and $P(z)$ be a predicate on variable z that must hold as a postcondition for the assignment. The absence of side-effects guarantees that $P(z)$ also is the weakest precondition. The possibility of side-effects, however, requires examining the function f, which might modify z. Moreover, the verifier—be it human or automatic—must be careful if aliasing is permitted by the language. In the example, $P(z)$ would not be the weakest precondition if z and y are aliases.

[illegible]

What is the practical influence of program verification on the programming activity? Unlike what many computer scientists foresaw in the past, program verification has not become common practice in everyday programming and probably never will. Nevertheless, program verification issues have a deep influence on programming and programming languages. They stimulate a rigorous approach to programming and provide a formal definition of programming languages. Even in the absence of a mechanical program verifier, rigorous reasoning about program correctness can help the programmer in discovering possible errors. Intermediate assertions that should be proven by the verifier (e.g., loop invariants) can be expressed as run-time checks; this is a useful way of certifying programs via systematic testing.

Another important use of axiomatic semantics is in the methodologies that try to derive programs that are correct in the first place. This approach has been exemplified by Dijkstra, who illustrated a calculus that, given input and output predicates, can be used to synthesize correct programs. This is a *constructive approach*: programs are not proven correct *a posteriori*, after being written, but are derived correct by the calculus. The approach has been demonstrated to work on examples of low-to-moderate complexity, but it is not clear if (and how) it can be used in more complex and larger applications. Further discussion of this topic is beyond the scope of this text, and belongs in the area of programming methodology. The reader interested in the subject will find references to the literature in the Further Reading section.

9.4 AN INTRODUCTION TO DENOTATIONAL SEMANTICS

As we said in Section 9.2, denotational semantics associates to each program a function from the input domain to the output domain. To do so, as we did for axiomatic semantics, it is necessary to formalize the notion of *state*, that is, the concepts of *memory* (that binds identifiers to values), *input*, and *output*. Instructions of our programming language will be modeled through the state transformation they imply. For simplicity, we assume that our programs deal only with simple integer values and booleans, which may result from relational expressions. Symbol Z stands for the set of integers; symbol "*undef*" stands for the undefined value.

For any given program P, P's state s_P is formalized by a triple

$$<mem_P, i_P, o_P>$$

where:

- mem_P is a function that gives the value of each identifier. If Id_P is the set of P's identifiers, we can write:

$$mem_P: Id_P \rightarrow Z \cup \{undef\}$$

- i_P and o_P are the input and output streams, respectively. Both are strings of integers, that is, i_P and o_P are elements of Z^* (symbol * denotes the reflexive and transitive closure of a set. Thus Z^* is the set of all sequences of integers, including the null sequence.)

Each language instruction will be specified now in terms of a state transformation. To this end, we define a function *dsem* (for *d*enotational *sem*antics) for each construct of the language and we define how *dsem* can be constructed for each program in terms of function *dsem* for individual statements. To make the notation more readable, an abbreviation for the name of the construct will be used as a subscript of *dsem* in our formal definitions; S will denote the set of states.

Let us start our analysis with *arithmetic expressions*. Assuming that expression evaluation does not produce any side-effects, arithmetic expressions do not cause any state change; thus semantics of arithmetic expressions simply describe how a value is produced by expression evaluation. If EX is the set of all legal arithmetic expressions, we can write:

[illegible]

$$dsem_{EX}: EX \times S \to Z \cup \{error\}$$

[illegible]

where

$dsem_{EX}(E, s) = error$ if $s = <mem, i, o>$ and $mem(v) = undef$ for some variable v occurring in E; otherwise
$dsem_{EX}(E, s) = e$ if $s = <mem, i, o>$ and e is the result of evaluating E after replacing each variable v occurring in E with $mem(v)$.

According to this definition, an error can arise only because of undefined operands. For simplicity, we ignore the possibility of overflows and underflows during execution. Also, we implicitly assume here—and in what follows—that programs are statically correct; thus, for example, type errors can be ignored.

Let AS be the set of all legal *assignment statements*. Semantics of assignment statements can be defined as a state-transformation function.

$$dsem_{AS}: AS \times S \to S \cup \{error\}$$

where

$dsem_{AS}(x := E, s) = error$ if $dsem_{EX}(E, s) = error$; otherwise
$dsem_{AS}(x := E, s) = s'$ where $s' = <mem', i', o'>$, $s = <mem, i, o>$,
$i' = i$, $o' = o$, $mem'(y) = mem(y)$ for all $y \neq x$,
$mem'(x) = dsem_{EX}(E, s)$

Suppose that *input statements* are written in our language as $x :=$ **read** ( ), which means that the next input value read is assigned to x. Intuitively, the effect of such a statement is a state modification that affects both the memory and the input stream. Formally, let RD be the set of all legal **read** statements. Then

$$dsem_{RD}: RD \times S \to S \cup \{error\}$$

where

$dsem_{RD}(x :=$ **read**( ), $s) = error$ if $s = <mem, i, o>$ and i is empty; otherwise
$dsem_{RD}(x :=$ **read**( ), $s) = s'$ where $s = <mem, i, o>$, $s' = <mem', i', o'>$,
$o = o'$, $i = ii'$ for some i in Z and some i' in Z^*,
$mem(y) = mem'(y)$ for all $y \neq x$, and $mem(x) = i$

Similarly, if WR is the set of all *write statements*, we define

$$dsem_{WR}: WR \times S \to S \cup \{error\}$$

where

[illegible]

$dsem_{WR}($**write**$(x), s) = error$ if $s = <mem, i, o>$ and $mem(x) = undef$; otherwise
$dsem_{WR}($**write**$(x), s) = s'$ where $s = <mem, i, o>$, $s' = <mem', i', o'>$,
$mem = mem'$, $i = i'$, $o' = oO$, where $O = mem(x)$

Now, let us turn to compound statements. First, let SL be the set of all *statement lists*. Semantics can be specified by the function

$$dsem_{SL}: SL \times S \to S \cup \{error\}$$

We define $dsem_{SL}$ recursively as follows. First, if the statement list is the empty list, the state does not change:

$$dsem_{SL}(empty\_list, s) = s$$

Second, if the statement list is a statement T followed by a statement list L and *dsem* describes T's semantics:

[illegible] would [illegible] better [illegible]

$dsem_{SL}(T; L, s) = error$ if $dsem(T, s) = error$; otherwise
$dsem_{SL}(T; L, s) = dsem_{SL}(L, dsem(T, s))$

As far as *selection* is concerned, let us refer to a Pascal-like **if** . . . **then** . . . **else** . . . **fi** statement. If the statement list in the **else** branch is empty, then our selection statement can be abbreviated as **if** . . . **then** . . . **fi**. If IF is the set of all correct selections we can write in our language, we define

$$dsem_{IF}: IF \times S \to S \cup \{error\}$$

If B is a boolean valued relational expression, $L1$ and $L2$ are two statement lists; we have:

$dsem_{IF}($**if** B **then** $L1$ **else** $L2$ **fi**$, s) = error$ if $dsem_{BOOL}(B, s) = undef$; otherwise
$dsem_{IF}($**if** B **then** $L1$ **else** $L2$ **fi**$, s) = U$ where if $dsem(B, s) = true$, then
$U = dsem_{SL}(L1, s)$, else
$U = dsem_{SL}(L2, s)$

Semantic function $dsem_{BOOL}$ describes the boolean result of a relational expression. It can be defined exactly like $dsem_{EX}$ and its definition is left to the reader.

Finally, we define the semantics of a Pascal-like **while** . . . **do** . . . **od** statement. If DO is the set of all syntactically correct loops, we define

$$dsem_{DO}: DO \times S \to S \cup \{error\}$$

If B is a boolean expression and L is a statement list, we can define $dsem_{DO}$ as a *recursive function*

$dsem_{DO}($**while** B **do** L **od**$, s) = error$ if $dsem_{BOOL}(B, s) = undef$; otherwise
$dsem_{DO}($**while** B **do** L **od**$, s) = s$ if $dsem_{BOOL}(B, s) = false$; otherwise

$dsem_{DO}$(**while** B **do** L **od**, s) = *error* if $dsem_{SL}(L, s)$ = *error*; otherwise
$dsem_{DO}$(**while** B **do** L **od**, s) = $dsem_{DO}$(**while** B **do** L **od**, $dsem_{SL}(L, s)$)

Finally, if PROG is the set of all statically correct programs in our language, the *language semantics* is defined by the following function:

$$dsem_{PROG}: PROG \times Z^* \rightarrow Z^* \cup \{error\}$$

(Z^* represents both the input—when it appears to the left of '→'—and the output domain). If L is the statement list that constitutes a program, function $dsem_{PROG}$ is defined as:

$$dsem_{PROG}(L, i) = out\,(dsem_{SL}\,(L, init\,(i)))$$

where

- init (i) = <*mem*0, *i*, *o*>, *mem*0(x) = *undef* for all identifiers x, o = empty
- out (*error*) = *error*
- out (<*mem*, *i*, *o*>) = *o*

If we wish to formalize additional constructs of the language, we may run into unexpected problems that cannot be handled within the current theoretical framework. For example, the current model does not allow us to deal with aliasing. Because in our model *mem* maps identifier names directly to values, there is no way to specify that two identifiers share the same object. Also, in our model the result of the function *dsem* associated with some construct is passed to the function associated to the construct that follows it in the program. This does not model the case where a jump or a procedure call causes a break in the sequential control flow. A denotational specification can be given to cover these cases too, but this requires additional mathematical sophistication.

Before closing this section, let us consider an example.

Example

Consider the following program P

```
read (n); fact:= 1; i:= 1;
while i <= n do
      fact:= fact * i;
      i:= i+1
od;
write (fact);
```

We wish to evaluate P's denotational semantics. Obviously, if the input stream is empty we have

$$dsem_{PROG}(P, empty) = error$$

[illegible]

(The formal proof of this is left to the reader.) Let us consider now an input string consisting of one integer value z. We have

(a) $dsem_{PROG}(P, z)$ = out ($dsem_{SL}$ (**read** (n); . . . ; **write**(*fact*), <*mem*0, z, empty>))
= out ($dsem_{SL}$ (*fact*:= 1; . . . , **write**(*fact*), <*mem*1, empty, empty>)),

where *mem*0 (x) = *undef* for all identifiers x, *mem*1 (x) = *undef* for all identifiers $x \neq n$, *mem*1 (n) = z.

Skipping a few trivial steps, we get [illegible]

(b) $dsem_{PROG}(P, z)$ = out ($dsem_{SL}$ (**while** . . . ; **write** (*fact*), <*mem*2, empty, empty>))

where *mem*2(n) = z, *mem*2(*fact*) = 1, *mem*2(i) = 1, *mem*2(x) = *undef* for any other identifier x.

Let us examine the **while** loop.

(c) $dsem_{DO}$(**while** . . . **od**, <*mem*2, empty, empty>) =
  if $dsem_{BOOL}$ (i <= n, <*mem*2, empty, empty>) = false
    then <*mem*2, empty, empty>
    else $dsem_{DO}$(**while** . . . **od**, $dsem_{SL}$(*fact*:= *fact* * i; i:= i + 1, <*mem*2, empty, empty>)) =
  if $dsem_{BOOL}$ (i <= n, <*mem*2, empty, empty>) = false
    then <*mem*2, empty, empty>
    else $dsem_{DO}$(**while** . . . **od**, <*mem*3, empty, empty>)
  where *mem*3(n) = *mem*2(n), *mem*3 (fact) = *mem*2 (*fact*) * *mem*2 (i), *mem*3(i) = *mem*2(i) + 1, *mem*3(x) = *undef* for any other identifier x

Equation (c) defines recursively a function $dsem_{DO}$ from S to $S \cup$ {error}. To solve it, we must rely upon the theory of recursive functions that underlies denotational semantics. Without entering into the details of the theory, we give the resulting function: [illegible]

$dsem_{DO}$(**while** . . . **do**, <*mem*2, empty, empty>) = <*mem*2', empty, empty>

where *mem*2'(*fact*) = *mem*2(n)! = z! From **b** and the above $dsem_{DO}$ we obtain
[illegible]

(d) $dsem_{PROG}(P, z)$ = out (<*mem*2', empty, z!>) = z!

This example has given a denotational semantics to a very simple iterative Pascal-like program. In particular, it has shown that denotational semantics is founded on the theory of recursive functions; functions that define the semantics of a program are described through recursive equations and one must be able to solve them. For more information, the interested reader is referred to the literature in the Further Readings section of this chapter.

9.5 CONCLUSIONS

This chapter introduced the concept of formal semantics and illustrated the spirit of two different approaches: axiomatic semantics and denotational semantics. The mathematical prerequisites needed to develop the subject in more depth prevented us from discussing more details, particularly in the case of denotational semantics.

Although, in practice, languages are seldom described formally, formal definitions can be useful and will be used more in the future. As we anticipated in Section 9.2, we envisage an interplay between formal and informal specification techniques in the definition of semantics. In particular, we suggested that the informal reports on the use of the language should be derived systematically from the formal definition. The official definition of the language should be formal, and we should refer to it every time a conflict arises in the interpretation of the informal reports.

Besides providing a notation for describing languages, formal semantics supports rigorous reasoning about programs. In particular, it supports (mechanical) program verification. We have seen this for axiomatic semantics, which is more intuitive, but there also are proof techniques that derive from denotational semantics (see the Further Readings section). Verification issues, in turn, have influenced the design of programming languages (this is yet another reason why we have presented the topic here). Another area where formal semantics can be useful is the automatic production of language translators from the language's semantic description. Both axiomatic and denotational semantics have been used for this purpose.

Denotational and axiomatic semantics are based on different mathematical foundations. Denotational semantics is based on functions, in particular, recursively defined functions. Axiomatic semantics is based on logic, in particular, predicate calculus. Both are formal and can support correctness proofs; the choice of one or the other is mostly a matter of individual taste and mathematical background. Axiomatic semantics is more easily readable; its style is more closely related to a declarative specification style where one states the properties of the mechanism that is being defined without saying anything about the implementation. The style of denotational specifications, on the other hand, is more in terms of a highly stylized and very abstract implementation.

SUGGESTIONS FOR FURTHER READING AND BIBLIOGRAPHIC NOTES

Formal semantics of programming languages usually is treated in specialized texts and taught in specialized courses. We have only introduced the subject here because a more complete treatment cannot be done in a single chapter of a book.

Mandrioli and Ghezzi (1986) present the mathematical background needed to develop the formal description of semantics of programming languages and provide a discussion of axiomatic and denotational semantics. Tennent (1976) and Gordon (1979) are excellent introductions to denotational semantics. A denotational technique is described by Bjórner and Jones (1978).

Dijkstra (1976) defines programming language semantics in terms of weakest preconditions and illustrates the use of the concept in a methodology for developing correct programs. Given the input and output predicates, Dijkstra illustrates a calculus that allows one to derive a program that is correct with respect to the predicates. The approach is further developed by Gries (1981).

Hoare and Wirth (1973) and Alagić and Arbib (1978) present an axiomatic description of Pascal semantics. Tennent (1973) presents a formal definition of SNOBOL4. Kahn et al. (1980) give a preliminary report on the formal definition of Ada; a complete formal definition of semantics is presently under development. ALGOL 68 has been among the first languages for which a formal definition has been given (van Wijngaarden et al. 1976). The formalism used there is based on so-called two-level grammars. An axiomatic definition of ALGOL 68 has also been given (Schwartz 1978b).

The mathematical foundations of axiomatic semantics and program verification were laid by Floyd (1967) and Hoare (1969). A presentation of the Floyd–Hoare theory is contained in (Manna 1973). Manna (1973) discusses the method of computational induction for proving functional programs correct using denotational semantics. The theory of program correctness also is studied in (DeBakker 1980).

The evolving state-of-the-art of the field of automatic program verification is surveyed in (Yeh 1977b), by London's paper in (Wegner 1979), and by Good (1985). DeMillo et al. (1979) argue that program verification cannot be used in practice to guarantee software reliability, because of the nature of proofs. It reports examples from mathematics in which proofs of theorems are shown to contain errors years after their truth were accepted.

EXERCISES

9.1 Describe the axiomatic semantics of the following Pascal statements:

- **repeat** . . . **until**.
- **for**.
- **case**.

9.2 Describe the axiomatic semantics of Dijkstra's guarded **if** and **do** statements.

9.3 Give an example of an assignment statement with a side-effect that invalidates the rule given in Section 9.3 to evaluate the weakest precondition.

9.4 Consider the Pascal-like program of the example given in Section 9.4. Using axiomatic semantics, prove it correct with respect to the following input and output assertions:

$$\text{IN:}\quad n>0$$
$$\text{OUT: } fact=n*(n-1)*(n-2)*\ldots*1$$

9.5 Write an iterative Pascal program that evaluates the product of two positive integers m and n by repeated additions. Prove the program correct with respect to the following assertions:

$$\text{IN:}\quad m>0, n>0$$
$$\text{OUT: } prod=m*n.$$

Does the program work properly, with exactly the same postcondition, if either $m \geq 0$ or $n \geq 0$? If the answer is yes in one such case, why can we say that IN is not the weakest precondition?

9.6 What is the weakest precondition for some postcondition P and a program that contains a loop that never terminates? For example

```
i:= 0;
while i >= 0 do
      i:= i+1
od
```

9.7 Give a denotational description of the Pascal **case** statement.

9.8 Give a formal definition of $dsem_{BOOL}$, which was introduced but not defined in Section 9.4.

9.9 Give a denotational description of the Pascal **repeat** statement.

9.10 Suppose that two concurrent processes $P1$ and $P2$ are executing the statement lists $L1$ and $L2$, respectively. Suppose that these processes also access some shared variables in mutual exclusion. Does $dsem_{SL}$ defined in Section 9.4 describe the semantics of $L1$ and $L2$ correctly? Why? Why not?

10 Language Design

". . . consolidation, not innovation . . ." (*Hoare 1973*)

The leitmotiv of this book is that programming languages are tools for software production. The previous chapters have expounded this viewpoint in depth by discussing programming language concepts, and comparing and evaluating the many solutions adopted by existing programming languages. We have suggested a number of criteria for evaluating programming languages, centered around the concepts of data types, control structures, program correctness, and programming in the large. Viewed slightly differently, however, these criteria also suggest a number of guidelines for programming language design. As we examine language design issues in this chapter, therefore, we will encounter many considerations that we have seen in previous chapters. This chapter, then, is mainly recapitulatory; we will try to put together the many facets of the problem and, in several cases, show how different language features, each desirable in itself, can interfere with one another when combined.

According to C. A. R. Hoare (Hoare 1973),

> The language designer should be familiar with many alternative features designed by others, and should have excellent judgment in choosing the best and rejecting any that are mutually inconsistent. He must be capable of reconciling, by good engineering design, any remaining minor inconsistencies or overlaps between separately designed features. He must have a clear idea of the scope and purpose and range of application of his new language, and how far it should go in size and complexity. . . . One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation.

Having designed a language, it is also necessary to design an implementation for it. According to Wirth (1975a), "In practice, a programming language is as good as its compiler(s)." In fact, much of the popularity of older languages such as FORTRAN probably stems from the availability of effi-