CHAPTER 12

In Chapters 7 through 10 we discussed the semantics, or meaning, of programs from an informal, or descriptive, point of view. Historically, this has been the usual approach, both for the programmer and the language designer, and it is typified by the language reference manual, which explains language features based on an underlying model of execution that is more implied than explicit.

However, many computer scientists have emphasized the need for a more mathematical description of the behavior of programs and programming languages. The advantages of such a description are to make the definition of a programming language so precise that programs can be **proven** correct in a mathematical way and that translators can be **validated** to produce exactly the behavior described in the language definition. In addition, the work of producing such a precise specification aids the language designer in discovering inconsistencies and ambiguities.

Attempts to develop a standard mathematical system for providing precise semantic descriptions of languages have not met with complete acceptance, so there is no single method for formally defining semantics. Instead, there are a number of methods that differ in the formalisms used and the kinds of intended applications. No one method can be considered universal and most languages are still specified in a somewhat informal way. Formal semantic descriptions are more often supplied after the fact, and only for a part of the language. Also, the use of formal definitions to prove correct program behavior has been confined primarily to academic settings, although formal methods have begun to be used as part of the specification of complex software projects, including programming language translators. The purpose of this chapter is to survey the different methods that have been developed and to give a little flavor of their potential application.

Researchers have developed three principal methods to describe semantics formally:

1. *Operational semantics*. This method defines a language by describing its actions in terms of the operations of an actual or hypothetical machine. Of course, this requires that the operations of the machine used in the description also be precisely defined, and for this reason a very simple hypothetical machine is often used that bears little resemblance to an actual computer. Indeed, the machine we use for operational semantics in Section 12.2 is more of a mathematical model, namely, a "reduction machine," which is a collection of permissible steps in reducing programs by applying their operations to values. It is similar in spirit to the notion of a Turing machine, in which actions are precisely described in a mathematical way.
2. *Denotational semantics*. This approach uses mathematical functions on programs and program components to specify semantics. Programs are translated into functions about which properties can be proved using the standard mathematical theory of functions.

3. *Axiomatic semantics*. This method applies mathematical logic to language definition. Assertions, or predicates, are used to describe desired outcomes and initial assumptions for programs. Language constructs are associated with **predicate transformers** that create new assertions out of old ones, reflecting the actions of the construct. These transformers can be used to prove that the desired outcome follows from the initial conditions. Thus, this method of formal semantics is aimed specifically at correctness proofs.

All these methods are syntax-directed—that is, the semantic definitions are based on a context-free grammar or Backus-Naur Form (BNF) rules as discussed in Chapter 6. Formal semantics must then define all properties of a language that are not specified by the BNF.

These include static properties such as static types and declaration before use, which a translator can determine prior to execution. Although some authors consider such static properties to be part of the syntax of a language rather than its semantics, formal methods can describe both static and dynamic properties, and we will continue to view the semantics of a language as everything not specified by the BNF.

As with any formal specification method, two properties of a specification are essential. First, it must be **complete**; that is, every correct, terminating program must have associated semantics given by the rules. Second, it must be **consistent**; that is, the same program cannot be given two different, conflicting semantics. Finally, though of much less concern, it is advantageous for the given semantics to be minimal, or **independent**, in the sense that no rule is derivable from the other rules.

Formal specifications written in the operational or denotational style have a nice additional property, in that they can be translated relatively easily into working programs in a language suitable for prototyping, such as Prolog, ML, or Haskell.

In the sections that follow, we give an overview of each of these approaches to formal semantics. To make the differences in approach clearer, we use a sample small language as a standard example. In the operational semantics section, we also show briefly how the semantics of this example might be translated into a Prolog program. (Prolog is closest in spirit to operational semantics.) In the denotational semantics section, we also show briefly how the semantics of the example might be translated into a Haskell program. (Haskell is closest in spirit to denotational semantics.)

First, however, we give a description of the syntax and informal semantics of this language.

12.1 A Sample Small Language

The basic sample language that we will use throughout the chapter is a version of the integer expression language used in Chapter 6 and elsewhere. BNF rules for this language are given in Figure 12.1.

expr → *expr* '+' *term* | *expr* '−' *term* | *term*
term → *term* '*' *factor* | *factor*
factor → '(' *expr* ')' | *number*
number → *number digit* | *digit*
digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Figure 12.1 Basic sample language

The semantics of such arithmetic expressions are particularly simple: The value of an expression is a complete representation of its meaning. Thus, $2 + 3 * 4$ means the value 14, and $(2 + 3) * 4$ means 20. Since this language is a little too simple to demonstrate adequately all the aspects of the formal methods, we add complexity to it in two stages, as follows.

In the first stage, we add variables, statements, and assignments, as given by the grammar in Figure 12.2.

A program in the extended language consists of a list of statements separated by semicolons, and a statement is an assignment of an expression to an identifier. The grammar of Figure 12.1 remains as before, except that identifiers are added to factors. The revised grammar is shown in Figure 12.2.

factor → '(' *expr* ') ' | *number* | *identifier*
program → *stmt-list*
stmt-list → *stmt* '; ' *stmt-list* | *stmt*
stmt → *identifier* ': =' *expr*
identifier → *identifier letter* | *letter*
letter → 'a' | 'b' | 'c' | . . . | 'z'

Figure 12.2 First extension of the sample language

The semantics of such programs are now represented not by a single value but by a set of values corresponding to identifiers whose values have been defined, or bound, by assignments. For example, the program

```
a := 2+3;
b := a*4;
a := b-5
```

results in the bindings $b = 20$ and $a = 15$ when it finishes, so the set of values representing the semantics of the program is $\{a = 15, b = 20\}$. Such a set is essentially a function from identifiers (strings of lower-case letters according to the foregoing grammar) to integer values, with all identifiers that have not been assigned a value undefined. For the purposes of this chapter we will call such a function an **environment**, and we will write:

$$Env\text{: Identifier} \to \text{Integer} \cup \{\text{undef}\}$$

to denote a particular environment *Env*. For example, the *Env* function given by the program example can be defined as follows:

$$Env(I) = \begin{cases} 15 \text{ if } I = a \\ 20 \text{ if } I = b \\ \text{undef otherwise} \end{cases}$$

The operation of looking up the value of an identifier *I* in an environment *Env* is then simply described by function evaluation *Env*(*I*). The operation of adding a new value binding to *Env* can also be defined in functional terms. We will use the notation *Env* & $\{I = n\}$ to denote the adding of the new value *n* for *I* to *Env*. In terms of functions,

$$(Env\& \{I = n\})(J) = \begin{cases} n \text{ if } J = I \\ Env(J) \text{ otherwise} \end{cases}$$

Finally, we also need the notion of the **empty environment**, which we will denote by Env_0:

$$Env_0(I) = \text{undef for all } I$$

This notion of environment is particularly simple and differs from what we called the environment in Chapters 7 and 10. Indeed, an environment as defined here incorporates both the symbol table and state functions from Chapter 7. We note that such environments do not allow pointer values, do not include scope information, and do not permit aliases. More complex environments require much greater complexity and will not be studied here.

This view of the semantics of a program as represented by a resulting, final environment has the effect that the consistency and completeness properties stated in the introduction have the following straightforward interpretation: Consistency means that we cannot derive two different final environments for the same program, and completeness means that we must be able to derive a final environment for every correct, terminating program.

The second extension to our sample language will be the addition of "if" and "while" control statements to the first extension. Statements can now be of three kinds, and we extend their definition accordingly; see Figure 12.3.

stmt → *assign-stmt* | *if-stmt* | *while-stmt*
assign-stmt → *identifier* ':=' *expr*
if-stmt → 'if' *expr* 'then' *stmt-list* 'else' *stmt-list* 'fi'
while-stmt → 'while' *expr* 'do' *stmt-list* 'od'

Figure 12.3 Second extension of the sample language

The syntax of the if statement and while statement borrow the Algol68 convention of writing reserved words backward—thus od and fi—to close statement blocks rather than using the begin and end of Pascal and Algol60.

The meaning of an *if-stmt* is that *expr* should be evaluated in the current environment. If it evaluates to an integer greater than 0, then the *stmt-list* after 'then' is executed. If not, the *stmt-list* after the 'else' is executed. The meaning of a *while-stmt* is similar: As long as *expr* evaluates to a quantity greater than 0, *stmt-list* is repeatedly executed, and *expr* is reevaluated. Note that these semantics are nonstandard!

Here is an example of a program in this language:

```
n := 0 - 5;
if n then i := n else i := 0 - n fi;
fact := 1;
while i do
  fact := fact * i;
  i := i - 1
od
```

The semantics of this program are given by the (final) environment $\{n = -5, i = 0, \text{fact} = 120\}$.

Loops are the most difficult of the foregoing constructs to give formal semantics for, and in the following we will not always give a complete solution. These can be found in the references at the end of the chapter.

Formal semantic methods frequently use a simplified version of syntax from that given. Since the parsing step can be assumed to have already taken place, and since semantics are to be defined only for syntactically correct constructs, an ambiguous grammar can be used to define semantics. Further, the nonterminal symbols can be replaced by single letters, which may be thought to represent either strings of tokens or nodes in a parse tree. Such a syntactic specification is sometimes called **abstract syntax**. An abstract syntax for our sample language (with extensions) is the following:

$$
\begin{array}{l}
P \rightarrow L \\
L \rightarrow L_1 \text{ `;' } L_2 \mid S \\
S \rightarrow I \text{ `:=' } E \mid \text{ `if' } E \text{ `then' } L_1, \text{ `else' } L_2 \text{ `fi'} \\
\quad\quad \mid \text{ `while' } E \text{ `do' } L \text{ `od'} \\
E \rightarrow E_1 \text{ `+' } E_2 \mid E_1 \text{ `}-\text{' } E_2 \mid E_1 \text{ `*' } E_2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \text{ `(' } E_1 \text{ `)'} \mid N \\
N \rightarrow N_1\, D \mid D \\
D \rightarrow \text{`0'} \mid \text{`1'} \mid \ldots \mid \text{`9'} \\
I \rightarrow I_1\, A \mid A \\
A \rightarrow \text{`}a\text{'} \mid \text{`}b\text{'} \mid \ldots \mid \text{`}z\text{'}
\end{array}
$$

Here the letters stand for syntactic entities as follows:

P :	Program
L :	Statement-list
S :	Statement
E :	Expression
N :	Number
D :	Digit
I :	Identifier
A :	Letter

To define the semantics of each one of these symbols, we define the semantics of each right-hand side of the abstract syntax rules in terms of the semantics of their parts. Thus, syntax-directed semantic definitions are recursive in nature. This also explains why we need to number the letters on the right-hand sides when they represent the same kind of construct: Each choice needs to be distinguished.

We note finally that the tokens in the grammar have been enclosed in quotation marks. This becomes an important point when we must distinguish between the symbol '1' and the operation of addition on the integers, or 1, which it represents. Similarly, the symbol '3' needs to be distinguished from its value, or the number 3.

We now survey the different formal semantic methods.

12.2 Operational Semantics

Operational semantics define the semantics of a programming language by specifying how an arbitrary program is to be executed on a machine whose operation is completely known.

We have noted in the introduction that there are many possibilities for the choice of a defining machine. The machine can be an actual computer, and the operational semantics can be specified by an actual translator for the language written in the machine code of the chosen machine. Such **definitional interpreters** or **compilers** have in the past been de facto language definitions (FORTRAN and C were originally examples). However, there are drawbacks to this method: The defining translator may not be available to a user, the operation of the underlying computer may not be completely specified, and the defining implementation may contain errors or other unexpected behavior.

By contrast, operational semantics can define the behavior of programs in terms of an **abstract machine** that does not need to exist anywhere in hardware, but that is simple enough to be completely understood and to be simulated by any user to answer questions about program behavior.

In principle, an abstract machine can be viewed as consisting of a program, a control, and a store or memory, as shown in Figure 12-4.

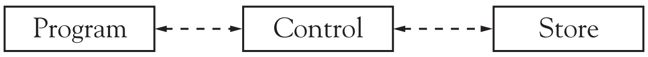

Figure 12-4 Three parts of an abstract machine

An operational semantic specification of a programming language specifies how the control of this abstract machine reacts to an arbitrary program in the language to be defined, and in particular, how storage is changed during the execution of a program.

The particular form of abstract machine that we will present is that of a **reduction machine**, whose control operates directly on a program to reduce it to its semantic "value." For example, given the expression (3 + 4) * 5, the control of the reduction machine will reduce it to its numeric value (which is its semantic content) using the following sequence of steps:

```
(3 + 4) * 5 => (7) * 5     — 3 and 4 are added to get 7
            => 7 * 5       — the parentheses around 7 are dropped
            => 35          — 7 and 5 are multiplied to get 35
```

In general, of course, the semantic content of a program will be more than just a numeric value, but as you will see, the semantic content of a program can be represented by a data value of some structured type, which operational semantic reductions will produce.

To specify the operational semantics of our sample language, we give **reduction rules** that specify how the control reduces the constructs of the language to a value. These reduction rules are given in a mathematical notation similar to logical inference rules, so we discuss such logical inference rules first.

12.2.1 Logical Inference Rules

Inference rules in logic are written in the following form:

$$\frac{\text{premise}}{\text{conclusion}}$$

That is, the premise, or condition, is written first; then a line is drawn, and the conclusion, or result, is written. This indicates that, whenever the premise is true, the conclusion is also true. As an example, we can express the commutative property of addition as the following inference rule:

$$\frac{a + b = c}{b + a = c}$$

In logic, such inference rules are used to express the basic rules of propositional and predicate calculus. As an example of an inference rule in logic, the transitive property of implication is given by the following rule:

$$\frac{a \to b,\ b \to c}{a \to c}$$

This says that if a implies b and b implies c, then a implies c.

Axioms are inference rules with no premise—they are always true. An example of an axiom is $a + 0 = a$ for integer addition. This can be written as an inference rule with an empty premise:

$$\frac{}{a + 0 = a}$$

More often, this is written without the horizontal line:

$$a + 0 = a$$

12.2.2 Reduction Rules for Integer Arithmetic Expressions

We use the notation of inference rules to describe the way the control operates to reduce an expression to its value. There are several styles in current use as to how these rules are written. The particular form we use for our reduction rules is called **structural operational semantics**; an alternative form, called **natural semantics**, is actually closer to the denotational semantics studied later; see the Notes and References.

We base the semantic rules on the abstract syntax for expressions in our sample language:

$$\begin{aligned}
&E \to E_1 \text{ `+' } E_2 \mid E_1 \text{ `−' } E_2 \mid E_1 \text{ `*' } E_2 \mid \text{ `(' } E_1 \text{ `)'}\\
&N \to N_1\, D \mid D\\
&D \to \text{`0'} \mid \text{`1'} \mid \ldots \mid \text{`9'}
\end{aligned}$$

For the time being we can ignore the storage, since this grammar does not include identifiers. We use the following notation: E, E_1, and so on to denote expressions that have not yet been reduced to values; V, V_1, and so on will stand for integer values; $E \Rightarrow E_1$ states that expression E reduces to expression E_1 by some reduction rule. The reduction rules for expressions are the following, each of which we will discuss in turn.

(1) We collect all the rules for reducing digits to values in this one rule, all of which are axioms:

$$\begin{gathered}
\text{'0'} \Rightarrow 0\\
\text{'1'} \Rightarrow 1\\
\text{'2'} \Rightarrow 2\\
\text{'3'} \Rightarrow 3\\
\text{'4'} \Rightarrow 4\\
\text{'5'} \Rightarrow 5\\
\text{'6'} \Rightarrow 6\\
\text{'7'} \Rightarrow 7\\
\text{'8'} \Rightarrow 8\\
\text{'9'} \Rightarrow 9
\end{gathered}$$

(2) We collect the rules for reducing numbers to values in this one rule, which are also axioms:

$$\begin{gathered}
V\,\text{'0'} \Rightarrow 10 * V\\
V\,\text{'1'} \Rightarrow 10 * V + 1\\
V\,\text{'2'} \Rightarrow 10 * V + 2\\
V\,\text{'3'} \Rightarrow 10 * V + 3\\
V\,\text{'4'} \Rightarrow 10 * V + 4\\
V\,\text{'5'} \Rightarrow 10 * V + 5\\
V\,\text{'6'} \Rightarrow 10 * V + 6\\
V\,\text{'7'} \Rightarrow 10 * V + 7\\
V\,\text{'8'} \Rightarrow 10 * V + 8\\
V\,\text{'9'} \Rightarrow 10 * V + 9
\end{gathered}$$

(3) $$V_1\ \text{'+'}\ V_2 \Rightarrow V_1 + V_2$$

(4) $$V_1\ \text{'−'}\ V_2 \Rightarrow V_1 - V_2$$

(5) $$V_1\ \text{'*'}\ V_2 \Rightarrow V_1 * V_2$$

(6) $$\text{'('}\ V\ \text{')'} \Rightarrow V$$

(7) $$\frac{E \Rightarrow E_1}{E\ \text{'+'}\ E_2 \Rightarrow E_1\ \text{'+'}\ E_2}$$

(8) $$\frac{E \Rightarrow E_1}{E\ \text{'−'}\ E_2 \Rightarrow E_1\ \text{'−'}\ E_2}$$

(9) $$\frac{E \Rightarrow E_1}{E\ \text{'*'}\ E_2 \Rightarrow E_1\ \text{'*'}\ E_2}$$

(10) $$\frac{E \Rightarrow E_1}{V\ \text{'+'}\ E \Rightarrow V\ \text{'+'}\ E_1}$$

(11) $$\frac{E \text{ => } E_1}{E \text{ ‘}-\text{’ } E \text{ => } V \text{ ‘}-\text{’ } E_1}$$

(12) $$\frac{E \text{ => } E_1}{V \text{ ‘*’ } E \text{ => } V \text{ ‘*’ } E_1}$$

(13) $$\frac{E \text{ => } E_1}{\text{‘(’ } E \text{ ‘)’ => ‘(’ } E_1 \text{ ‘)’}}$$

(14) $$\frac{E \text{ =>} E_1,\ E_1 \text{ => } E_2}{E \text{ => } E_2}$$

Rules 1 through 6 are all axioms. Rules 1 and 2 express the reduction of digits and numbers to values: ‘0’ => 0 states that the **character** ‘0’ (a syntactic entity) reduces to the **value** 0 (a semantic entity). Rules 3, 4, and 5 say that whenever we have an expression that consists of two values and an operator symbol, we can reduce that expression to a value by applying the appropriate operation whose symbol appears in the expression. Rule 6 says that if an expression consists of a pair of parentheses surrounding a value, then the parentheses can be dropped.

The remainder of the reduction rules are inferences that allow the reduction machine to combine separate reductions together to achieve further reductions. Rules 7, 8, and 9 express the fact that, in an expression that consists of an operation applied to other expressions, the left subexpression may be reduced by itself and that reduction substituted into the larger expression. Rules 10 through 12 express the fact that, once a value is obtained for the left subexpression, the right subexpression may be reduced. Rule 13 says that we can first reduce the inside of an expression consisting of parentheses surrounding another expression. Finally, rule 14 expresses the general fact that reductions can be performed stepwise (sometimes called the **transitivity rule** for reductions).

Let us see how these reduction rules can be applied to a complicated expression to derive its value. Take, for example, the expression 2 * (3 + 4) − 5. To show each reduction step clearly, we surround each character with quotation marks within the reduction steps.

We first reduce the expression 3 + 4 as follows:

```
‘3’ ‘+’ ‘4’ => 3 ‘+’ ‘4’        (Rules 1 and 7)
            => 3 ‘+’ 4          (Rules 1 and 10)
            => 3 + 4 = 7        (Rule 3)
```

Hence by rule 14, we have ‘3’ ‘+’ ‘4’ => 7. Continuing,

```
‘(’ ‘3’ ‘+’ ‘4’ ‘)’ => ‘(’ 7 ‘)’        (Rule 13)
                    => 7                (Rule 6)
```

Now we can reduce the expression 2 * (3 + 4) as follows:

```
‘2’ ‘*’ ‘(’ ‘3’ ‘+’ ‘4’ ‘)’ => 2 ‘*’ ‘(’ ‘3’ ‘+’ ‘4’ ‘)’        (Rules 1 and 9)
                            => 2 ‘*’ 7                          (Rule 12)
                            => 2 * 7 = 14                       (Rule 5)
```

And, finally,

$$
\begin{aligned}
\text{'2' '*' '(' '3' '+' '4' ')' '}-\text{' '5'} &\Rightarrow 14\ \text{'}-\text{' '5'} && \text{(Rules 1 and 8)}\\
&\Rightarrow 14\ \text{'}-\text{'}\ 5 && \text{(Rule 11)}\\
&\Rightarrow 14 - 5 = 9 && \text{(Rule 4)}
\end{aligned}
$$

We have shown that the reduction machine can reduce the expression 2 * (3 + 4) − 5 to 9, which is the value of the expression.

12.2.3 Environments and Assignment

We want to extend the operational semantics of expressions to include environments and assignments, according to the following abstract syntax:

$$
\begin{aligned}
&P \to L\\
&L \to L_1 \text{ ';' } L_2 \mid S\\
&S \to I \text{ ':=' } E\\
&E \to E_1 \text{ '+' } E_2 \mid E_1 \text{ '}-\text{' } E_2 \mid E_1 \text{ '*' } E_2\\
&\qquad \mid \text{ '(' } E_1 \text{ ')'} \mid N\\
&N \to N_1\, D \mid D\\
&D \to \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'}\\
&I \to I_1\, A \mid A\\
&A \to \text{'}a\text{'} \mid \text{'}b\text{'} \mid \ldots \mid \text{'}z\text{'}
\end{aligned}
$$

To do this, we must include the effect of assignments on the storage of the abstract machine. Our view of the storage will be the same as in other sections; that is, we view it as an environment that is a function from identifiers to integer values (including the undefined value):

$$Env\text{: Identifier} \to \text{Integer} \cup \{\text{undef}\}$$

To add environments to the reduction rules, we need a notation to show the dependence of the value of an expression on an environment. We use the notation $<E \mid Env>$ to indicate that expression E is evaluated in the presence of environment *Env*. Now our reduction rules change to include environments. For example, rule 7 with environments becomes:

(7) $$\frac{<E \mid Env> \Rightarrow <E_1 \mid Env>}{<E \text{ '+' } E_2 \mid Env> \Rightarrow <E_1 \text{ '+' } E_2 \mid Env>}$$

This states that if E reduces to E_1 in the presence of environment *Env*, then E '+' E_2 reduces to E_1 '+' E_2 in the same environment. Other rules are modified similarly. The one case of evaluation that explicitly involves the environment is when an expression is an identifier I:

(15) $$\frac{Env(I) = V}{<I \mid Env> \Rightarrow <V \mid Env>}$$

This states that if the value of identifier I is V in environment *Env*, then I reduces to V in the presence of *Env*.

It remains to add assignment statements and statement sequences to the reduction rules. First, statements must reduce to environments instead of integer values, since they create and change environments. Thus, we have:

(16) $$<I \text{ ':=' } V \mid Env> \Rightarrow Env \ \& \ \{I = V\}$$

which states that the assignment of the value *V* to *I* in environment *Env* reduces to a new environment where *I* is equal to *V*.

The reduction of expressions within assignments proceeds via the following rule:

(17) $$\frac{<E \mid Env> \Rightarrow <E_1 \mid Env>}{<I \text{ ':=' } E \mid Env> \Rightarrow <I \text{ ':=' } E_1 \mid Env>}$$

A statement sequence reduces to an environment formed by accumulating the effect of each assignment:

(18) $$\frac{<S \mid Env> \Rightarrow Env_1}{<S \text{ ';' } L \mid \text{Env}> \Rightarrow <L \mid Env_1>}$$

Finally, a program is a statement sequence that has no prior environment; it reduces to the effect it has on the empty starting environment:

(19) $$L \Rightarrow <L \mid Env_0>$$

(recall that $Env_0(I)$ = undef for all identifiers *I*).

We leave the rules for reducing identifier expressions to the reader; they are completely analogous to the rules for reducing numbers (see Exercise 12.4).

Let us use these rules to reduce the following sample program to an environment:

```
a := 2+3;
b := a*4;
a := b-5
```

To simplify the reduction, we will suppress the use of quotes to differentiate between syntactic and semantic entities. First, by rule 19, we have:

$a := 2 + 3; b := a * 4; a := b - 5 \Rightarrow$
$<a := 2 + 3; b := a * 4; a := b - 5 \mid Env_0>$

Also, by rules 3, 17, and 16,

$<a := 2 + 3 \mid Env_0> \Rightarrow$
$<a := 5 \mid Env_0> \Rightarrow$
$Env_0 \ \& \ \{a = 5\} = \{a = 5\}$

Then, by rule 18,

$<a := 2 + 3; b := a * 4; a := b - 5 \mid Env_0> \Rightarrow$
$<b := a * 4; a := b - 5 \mid \{a = 5\}>$

Similarly, by rules 15, 9, 5, 17, and 16,

$<b := a * 4 \mid \{a = 5\}> \;=>\; <b := 5 * 4 \mid \{a = 5\}> \;=>$
$<b := 20 \mid \{a = 5\}> \;=>\; \{a = 5\}\ \&\ \{b = 20\} = \{a = 5, b = 20\}$

Thus, by rule 18,

$<b := a * 4; a := b - 5 \mid \{a = 5\}> \;=>$
$<a := b - 5 \mid \{a = 5, b = 20\}>$

Finally, by a similar application of the rules, we get:

$<a := b - 5 \mid \{a = 5, b = 20\}> \;=>$
$<a := 20 - 5 \mid \{a = 5, b = 20\}> \;=>$
$<a := 15 \mid \{a = 5, b = 20\}> \;=>$
$\{a = 5, b = 20\}\ \&\ \{a = 15, b = 20\}$

and the program reduces to the environment $\{a = 15, b = 20\}$.

12.2.4 Control

It remains to add the if- and while statements to our sample language, with the following abstract syntax:

$$S \rightarrow \text{‘if’}\ E\ \text{‘then’}\ L_1\ \text{‘else’}\ L_2\ \text{‘fi’}$$
$$\mid\ \text{‘while’}\ E\ \text{‘do’}\ L\ \text{‘od’}$$

Reduction rules for if statements are the following:

(20) $$\frac{<E \mid Env> \;=>\; <E_1 \mid Env>}{<\text{‘if’}\ E\ \text{‘then’}\ L_1\ \text{‘else’}\ L_2\ \text{‘fi’} \mid Env> \;=>\; <\text{‘if’}\ E_1\ \text{‘then’}\ L_1\ \text{‘else’}\ L_2\ \text{‘fi’} \mid Env>}$$

(21) $$\frac{V > 0}{<\text{‘if’}\ V\ \text{‘then’}\ L_1\ \text{‘else’}\ L_2\ \text{‘fi’} \mid Env> \;=>\; <L_1 \mid Env>}$$

(22) $$\frac{V \le 0}{<\text{‘if’}\ V\ \text{‘then’}\ L_1\ \text{‘else’}\ L_2\ \text{‘fi’} \mid Env> \;=>\; <L_2 \mid Env>}$$

Reduction rules for while statements are as follows:

(23) $$\frac{<E \mid Env> \;=>\; <V \mid Env>,\ V \le 0}{<\text{‘while’}\ E\ \text{‘do’}\ L\ \text{‘od’} \mid Env> \;=>\; Env}$$

(24) $$\frac{<E \mid Env> \;=>\; <V \mid Env>,\ V > 0}{<\text{‘while’}\ E\ \text{‘do’}\ L\ \text{‘od’} \mid Env \;=>\; < L;\ \text{‘while’}\ E\ \text{‘do’}\ L\ \text{‘od’} \mid Env>}$$

Note that the last rule is recursive. It states that if, given environment *Env*, the expression *E* evaluates to a positive value, then execution of the while-loop under *Env* reduces to an execution of *L*, followed by the execution of the same while-loop all over again.

As an example, let us reduce the while statement of the program

```
n := 0 - 3;
if n then i := n else i := 0 - n fi;
fact := 1;
while i do
  fact := fact * i;
  i := i - 1
od
```

to its environment value. The environment at the start of the while-loop is $\{n = -3, i = 3, \text{fact} = 1\}$. Since $<i \mid \{n = -3, i = 3, \text{fact} = 1\}> \Rightarrow <3 \mid \{n = -3, i = 3, \text{fact} = 1\}>$ and $3 > 0$, rule 24 applies, so by rule 18 we must compute the environment resulting from the application of the body of the loop to the environment $\{n = -3, i = 3, \text{fact} = 1\}$:

$<\text{fact} := \text{fact} * i \mid \{n = -3, i = 3, \text{fact} = 1\}> \Rightarrow$
$<\text{fact} := 1 * i \mid \{n = -3, i = 3, \text{fact} = 1\}> \Rightarrow$
$<\text{fact} := 1 * 3 \mid \{n = -3, i = 3, \text{fact} = 1\}> \Rightarrow$
$<\text{fact} := 3 \mid \{n = -3, i = 3, \text{fact} = 1\}> \Rightarrow$
$\{n = -3, i = 3, \text{fact} = 3\}$

and

$<i := i - 1 \mid \{n = -3, i = 3, \text{fact} = 1\}> \Rightarrow$
$<i := 3 - 1 \mid \{n = -3, i = 3, \text{fact} = 3\}> \Rightarrow$
$<i := 2 \mid \{n = -3, i = 3, \text{fact} = 3\}> \Rightarrow$
$\{n = -3, i = 2, \text{fact} = 3\}$

so

$<\text{while } i \text{ do} \ldots \text{od} \mid \{n = -3, i = 3, \text{fact} = 1\}> \Rightarrow$
$< \text{fact} := \text{fact} * i\,;\, i := i - 1; \text{while } i \text{ do} \ldots \text{od} \mid \{n = -3, i = 3, \text{fact} = 1\}> \Rightarrow$
$< i := i - 1; \text{while } i \text{ do} \ldots \text{od} \mid \{n = -3, i = 3, \text{fact} = 3\}> \Rightarrow$
$< \text{while } i \text{ do} \ldots \text{od} \mid \{n = -3, i = 2, \text{fact} = 3\}>$

Continuing in this way, we get:

$<\text{while } i \text{ do} \ldots \text{od} \mid \{n = -3, i = 2, \text{fact} = 3\}> \Rightarrow$
$<\text{while } i \text{ do} \ldots \text{od} \mid \{n = -3, i = 1, \text{fact} = 6\}> \Rightarrow$
$<\text{while } i \text{ do} \ldots \text{od} \mid \{n = -3, i = 0, \text{fact} = 6\}> \Rightarrow$
$\{n = -3, i = 0, \text{fact} = 6\}$

so the final environment is $\{n = -3, i = 0, \text{fact} = 6\}$.

12.2.5 Implementing Operational Semantics in a Programming Language

It is possible to implement operational semantic rules directly as a program to get a so-called **executable specification**. This is useful for two reasons. First, it allows us to construct a language interpreter directly from a formal specification. Second, it allows us to check the correctness of the specification by testing the resulting interpreter. Since operational semantic rules as we have defined them are similar to logical inference rules, it is not surprising that Prolog is a natural choice as an implementation language. In this section, we briefly sketch a possible Prolog implementation for the reduction rules of our sample language.

First, consider how we might represent a sample language program in abstract syntax in Prolog. This is easy to do using terms. For example, 3 * (4 + 5) can be represented in Prolog as:

```
times(3,plus(4,5))
```

and the program

```
a := 2+3;
b := a*4;
a := b-5
```

as

```
seq(assign(a,plus(2,3)),
        seq(assign(b,times(a,4)),assign(a,sub(b,5))))
```

Note that this form of abstract syntax is actually a tree representation and that no parentheses are necessary to express grouping. Thus, all rules involving parentheses become unnecessary. Note that we could also take the shortcut of letting Prolog compute the semantics of integers, since we could also write, for example, `plus(23,345)`, and Prolog will automatically compute the values of 23 and 345. Thus, with these shortcuts, rules 1, 2, 6, and 13 can be eliminated.

Consider now how we might write reduction rules. Ignoring environments for the moment, we can write a general reduction rule for expressions as:

```
reduce(X,Y) :- ...
```

where `X` is any arithmetic expression (in abstract syntax) and `Y` is the result of a single reduction step applied to `X`. For example, rule 3 can be written as:

```
reduce(plus(V1,V2),R) :-
        integer(V1), integer(V2), !, R is V1 + V2.
```

Here the predicate `integer` tests its argument to make sure that it is an (integer) value, and then `R` is set to the result of the addition.

In a similar fashion, rule 7 can be written as:

```
reduce(plus(E,E2),plus(E1,E2)) :- reduce(E,E1).
```

and rule 10 can be written as:

```
reduce(plus(V,E),plus(V,E1)) :-
        integer(V), !, reduce(E,E1).
```

Rule 14 presents a problem. If we write it as given:

```
reduce(E,E2) :- reduce(E,E1), reduce(E1,E2).
```

then infinite recursive loops will result. Instead, we make a distinction between a single reduction step, which we call `reduce` as before, and multiple reductions, which we will call `reduce_all`. We then write rule 14 as two rules (the first is the stopping condition of the recursion):

```
reduce_all(V,V) :- integer(V), !.
reduce_all(E,E2) :- reduce(E,E1), reduce_all(E1,E2).
```

Now, if we want the final semantic value `V` of an expression `E`, we must write `reduce_all(E,V)`.

Finally, consider how this program might be extended to environments and control. First, a pair <*E* | *Env*> or <*L* | *Env*> can be thought of as a *configuration* and written in Prolog as `config(E,Env)` or `config(L,Env)`. Rule 15 can then be written as:

```
reduce(config(I,Env),config(V,Env)) :-
                    atom(I), !, lookup(Env, I, V).
```

(`atom(I)` tests for a variable, and the `lookup` operation finds values in an environment). Rule 16 can be similarly written as:

```
reduce(config(assign(I,V),Env),Env1) :-
     integer(V), !, update(Env, value(I,V), Env1).
```

where `update` inserts the new value `V` for `I` into `Env`, yielding new environment `Env1`.

Any dictionary structure for which `lookup` and `update` can be defined can be used to represent an environment in this code. For example, the environment $\{n = -3, i = 3, \text{fact} = 1\}$ can be represented as the list `[value(n,23),value(i,3),value(fact,1)]`. The remaining details are left to the reader.

12.3 Denotational Semantics

Denotational semantics use functions to describe the semantics of a programming language. A function describes semantics by associating semantic values to syntactically correct constructs. A simple example of such a function is a function that maps an integer arithmetic expression to its value, which we could call the *Val* function:

$$Val : \text{Expression} \rightarrow \text{Integer}$$

For example, $Val(2 + 3 * 4) = 14$ and $Val((2 + 3) * 4) = 20$. The domain of a semantic function such as *Val* is a **syntactic domain**. In the case of *Val*, it is the set of all syntactically correct integer arithmetic expressions. The range of a semantic function is a **semantic domain**, which is a mathematical structure. In the case of *Val*, the set of integers is the semantic domain. Since *Val* maps the syntactic construct 2 + 3 * 4 to the semantic value 14, 2 + 3 * 4 is said to **denote** the value 14. This is the origin of the name denotational semantics.

A second example may be useful before we give denotational semantics for our sample language from Section 12.1. In many programming languages a program can be viewed as something that receives input and produces output. Thus, the semantics of a program can be represented by a function from input to output, and a semantic function for programs would look like this:

$$P : \text{Program} \to (\text{Input} \to \text{Output})$$

The semantic domain to which *P* maps programs is a set of functions, namely, the functions from Input to Output, which we represent by Input → Output, and the semantic value of a program is a function. For example, if *p* represents the C program

```
main()
{ int x;
  scanf("%d",&x);
  printf("%d\n",x);
  return 0;
}
```

that inputs an integer and outputs the same integer, then *p* denotes the identity function *f* from integers to integers: $P(p) = f$, where f: Integer → Integer is given by $f(x) = x$.

Very often semantic domains in denotational descriptions will be function domains, and values of semantic functions will be functions themselves. To simplify the notation of these domains, we will often assume that the function symbol "→" is right associative and leave off the parentheses from domain descriptions. Thus,

$$P : \text{Program} \to (\text{Input} \to \text{Output})$$

becomes:

$$P : \text{Program} \to \text{Input} \to \text{Output}$$

In the following, we will give a brief overview of a denotational definition of the semantics of a programming language and then proceed with a denotational definition of our sample language from Section 12.1. A denotational definition of a programming language consists of three parts:

1. A definition of the **syntactic domains**, such as the sets Program and Expression, on which the semantic functions act
2. A definition of the **semantic domains** consisting of the values of the semantic functions, such as the sets Integer and Integer → Integer
3. A definition of the semantic functions themselves (sometimes called **valuation functions**)

We will consider each of these parts of the denotational definition in turn.

12.3.1 Syntactic Domains

Syntactic domains are defined in a denotational definition using notation that is almost identical to the abstract syntax described in Section 12.1. The sets being defined are listed first with capital letters denoting elements from the sets. Then the grammar rules are listed that recursively define the elements of the set. For example, the syntactic domains Number and Digit are specified as follows:

D: Digit

N: Number

$N \rightarrow N\,D \mid D$

$D \rightarrow$ '0' | '1' | . . . | '9'

A denotational definition views the syntactic domains as sets of syntax trees whose structure is given by the grammar rules. Semantic functions will be defined recursively on these sets, based on the structure of a syntax tree node.

12.3.2 Semantic Domains

Semantic domains are the sets in which semantic functions take their values. These are sets like syntactic domains, but they also may have additional mathematical structure, depending on their use. For example, the integers have the arithmetic operations "+," "−," and "*." Such domains are **algebras**, which need to be specified by listing their functions and properties. A denotational definition of the semantic domains lists the sets and the operations but usually omits the properties of the operations. These can be specified by the algebraic techniques studied in Chapter 11, or they can simply be assumed to be well known, as in the case of the arithmetic operations on the integers. A specification of the semantic domains also lists only the basic domains without specifically including domains that are constructed of functions on the basic domains.

Domains sometimes need special mathematical structures that are the subject of **domain theory** in programming language semantics. In particular, the term "domain" is sometimes reserved for an algebra with the structure of a complete partial order. Such a structure is needed to define the semantics of recursive functions and loops. See the references at the end of the chapter for further detail on domain theory.

An example of a specification of a semantic domain is the following specification of the integers:

Domain v: Integer = $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$

Operations

$+$: Integer $\times$ Integer $\rightarrow$ Integer

$-$: Integer $\times$ Integer $\rightarrow$ Integer

$*$: Integer $\times$ Integer $\rightarrow$ Integer

In this example, we restrict ourselves to the three operations "+," "−," and "*," which are the only operations represented in our sample language. In the foregoing notation, the symbols "v:" in the first line indicate that the name v will be used for a general element from the domain, that is, an arbitrary integer.

12.3.3 Semantic Functions

A semantic function is specified for each syntactic domain. Each semantic function is given a different name based on its associated syntactic domain. A common convention is to use the boldface letter corresponding to the elements in the syntactic domain. Thus, the value function from the syntactic domain Digit to the integers is written as follows:

$$\boldsymbol{D} : \text{Digit} \to \text{Integer}$$

The value of a semantic function is specified recursively on the trees of the syntactic domains using the structure of the grammar rules. This is done by giving a **semantic equation** corresponding to each grammar rule.

For example, the grammar rules for digits

$$D \to \text{‘0’} \mid \text{‘1’} \mid \ldots \mid \text{‘9’}$$

give rise to the syntax tree nodes

$$\begin{array}{ccccc} D & & D & \ldots & D \\ | & & | & & | \\ \text{‘0’} & & \text{‘1’} & & \text{‘9’} \end{array}$$

and the semantic function **D** is defined by the following semantic equations:

$$\boldsymbol{D}\Bigl(\begin{array}{c} D \\ | \\ \text{‘0’} \end{array}\Bigr) = 0, \quad \boldsymbol{D}\Bigl(\begin{array}{c} D \\ | \\ \text{‘1’} \end{array}\Bigr) = 1, \ldots, \quad \boldsymbol{D}\Bigl(\begin{array}{c} D \\ | \\ \text{‘9’} \end{array}\Bigr) = 9$$

representing the value of each leaf.

This cumbersome notation is shortened to the following:

$$\boldsymbol{D}[[\text{‘0’}]] = 0, \boldsymbol{D}[[\text{‘1’}]] = 1, \ldots, \boldsymbol{D}[[\text{‘9’}]] = 9$$

The double brackets [[. . .]] indicate that the argument is a syntactic entity consisting of a syntax tree node with the listed arguments as children.

As another example, the semantic function

$$N : \text{Number} \to \text{Integer}$$

from numbers to integers is based on the syntax

$$N \to N\,D \mid D$$

and is given by the following equations:

$$\begin{aligned} &\boldsymbol{N}[[ND]] = 10 * \boldsymbol{N}[[N]]] + \boldsymbol{N}[[D]] \\ &\boldsymbol{N}[[D]] = \boldsymbol{D}[[D]] \end{aligned}$$

Here [[ND]] refers to the tree node $\begin{array}{c} N \\ / \backslash \\ N\ D \end{array}$ and [[D]] to the node $\begin{array}{c} N \\ | \\ D \end{array}$. We are now ready to give a complete denotational definition for the expression language of Section 12.1.

12.3.4 Denotational Semantics of Integer Arithmetic Expressions

Here is the denotational definition according to the conventions just described.

Syntactic Domains

E: Expression
N: Number
D: Digit

$E \rightarrow E_1$ '+' E_2 | E_1 '−' E_2 | E_1 '*' E_2
 | '(' E ')' | N
$N \rightarrow N\,D$ | D
$D \rightarrow$ '0' | 1' | . . . | '9'

Semantic Domains

Domain v: Integer = {. . ., −2, −1, 0, 1, 2, . . .}
Operations
 $+$: Integer × Integer → Integer
 $-$: Integer × Integer → Integer
 $*$: Integer × Integer → Integer

Semantic Functions

E : Expression → Integer

E[[E_1 '+' E_2]] = ***E***[[E_1]] + ***E***[[E_2]]
E[[E_1 '−' E_2]] = ***E***[[E_1]] − ***E***[[E_2]]
E[[E_1 '*' E_2]] = ***E***[[E_1]] * ***E***[[E_2]]
E[['(' E ')']] = ***E***[[E]]
E[[N]] = ***N***[[N]]

N: Number → Integer

N[[ND]] = 10 * ***N***[[N]]] + ***N***[[D]]
N[[D]] = ***D***[[D]]

D : Digit → Integer

D[['0']] = 0, ***D***[['1']] = 1,. . ., ***D***[['9']] = 9

In this denotational description, we have retained the use of quotation marks to distinguish syntactic from semantic entities. In denotational semantics, this is not as necessary as in other semantic descriptions, since arguments to semantic functions are always syntactic entities. Thus, we could drop the quotation marks and write ***D***[[0]] = 0, and so on. For clarity, we will generally continue to use the quotation marks, however.

To see how these equations can be used to obtain the semantic value of an expression, we compute $\boldsymbol{E}[[(2 + 3)*4]]$ or, more precisely, $\boldsymbol{E}[[\text{'('}\ \text{'2'}\ \text{'+'}\ \text{'3'}\ \text{')'}\ \text{'*'}\ \text{'4'}]]$:

$$\begin{aligned}
&\boldsymbol{E}[[\text{'('}\ \text{'2'}\ \text{'+'}\ \text{'3'}\ \text{')'}\ \text{'*'}\ \text{'4'}]]\\
&\quad = \boldsymbol{E}[[\text{'('}\ \text{'2'}\ \text{'+'}\ \text{'3'}\ \text{')'}]] * \boldsymbol{E}[[\text{'4'}]]\\
&\quad = \boldsymbol{E}[[\text{'2'}\ \text{'+'}\ \text{'3'}]] * \boldsymbol{N}[[\text{'4'}]]\\
&\quad = (\boldsymbol{E}[[\text{'2'}]] + \boldsymbol{E}[[\text{'3'}]]) * \boldsymbol{D}[[\text{'4'}]]\\
&\quad = (\boldsymbol{N}[[\text{'2'}]] + \boldsymbol{N}[[\text{'3'}]]) * 4\\
&\quad = \boldsymbol{D}[[\text{'2'}]] + \boldsymbol{D}[[\text{'3'}]]) * 4\\
&\quad = (2 + 3) * 4 = 5 * 4 = 20
\end{aligned}$$

12.3.5 Environments and Assignment

The first extension to our basic sample language adds identifiers, assignment statements, and environments to the semantics. Environments are functions from identifiers to integers (or undefined), and the set of environments becomes a new semantic domain:

Domain *Env*: Environment = Identifier → Integer ∪ {undef}

In denotational semantics the value undef is given a special name, *bottom*, taken from the theory of partial orders, and is denoted by a special symbol, $\perp$. Semantic domains with this special value added are called **lifted domains** and are subscripted with the symbol $\perp$. Thus, Integer ∪ {$\perp$} is written as $\text{Integer}_{\perp}$. The initial environment Env_0 defined in Section 12.1, in which all identifiers have undefined values, can now be defined as $Env_0(I) = \perp$ for all identifiers I.

The evaluation of expressions in the presence of an environment must include an environment as a parameter, so that identifiers may be associated with integer values. Thus, the semantic value of an expression becomes a function from environments to integers:

$$\boldsymbol{E} : \text{Expression} \to \text{Environment} \to \text{Integer } \perp$$

In particular, the value of an identifier is its value in the environment provided as a parameter:

$$\boldsymbol{E}[[I]](Env) = Env(I)$$

In the case of a number, the environment is immaterial:

$$\boldsymbol{E}[[N]](Env) = \boldsymbol{N}[[N]]$$

In other expression cases the environment is simply passed on to subexpressions.

To extend the semantics to statements and statement lists, we note that the semantic values of these constructs are functions from environments to environments. An assignment statement changes the environment to add the new value assigned to the identifier; in this case, we will use the same "&" notation for adding values to functions that we have used in previous sections. Now a statement-list is simply the composition of the functions of its individual statements (recall that the composition $f \circ g$ of two functions f and g is defined by $(f \circ g)(x) = f(g(x))$. A complete denotational definition of the extended language is given in Figure 12.5.

Syntactic Domains

P: Program
L: Statement-list
S: Statement
E: Expression
N: Number
D: Digit
I: Identifier
A: Letter

$P \rightarrow$ L
$L \rightarrow L_1$ ‘;’ $L_2 \mid S$
$S \rightarrow I$ ‘:=’ E
$E \rightarrow E_1$ ‘+’ $E_2 \mid E_1$ ‘−’ $E_2 \mid E_1$ ‘∗’ E_2
$\quad\mid$ ‘(’ E ‘)’ $\mid I \mid N$
$N \rightarrow N\,D \mid D$
$D \rightarrow$ ‘0’ | ‘1’ | . . .| ‘9’
$I \rightarrow I\,A \mid A$
$A \rightarrow$ ‘a’ | ‘b’ | . . .| ‘z’

Semantic Domains

Domain v: Integer = {. . ., −2, −1, 0, 1, 2,. . .}
Operations

$+$: Integer $\times$ Integer $\rightarrow$ Integer
$-$: Integer $\times$ Integer $\rightarrow$ Integer
$*$: Integer $\times$ Integer $\rightarrow$ Integer

Domain Env: Environment = Identifier $\rightarrow$ $\text{Integer}_\perp$

Semantic Functions

$\boldsymbol{P}$: Program $\rightarrow$ Environment

$$\boldsymbol{P}[[L]] = \boldsymbol{L}[[L]](Env_0)$$

$\boldsymbol{L}$: Statement-list $\rightarrow$ Environment $\rightarrow$ Environment

$$\boldsymbol{L}[[L_1 \text{ ‘;’ } L_2]] = \boldsymbol{L}[[L_2]] \circ \boldsymbol{L}[[L_1]]$$
$$\boldsymbol{L}[[S]] = \boldsymbol{S}[[\text{S}]]$$

Figure 12.5 A denotational definition for the sample language extended with assignment statements and environments *(continues)*

(continued)

$\boldsymbol{S}$: Statement $\rightarrow$ Environment $\rightarrow$ Environment

$$\boldsymbol{S}[[\ I\ \text{`:='}\ E\ ]](Env) = Env\ \&\ \{I = \boldsymbol{E}[[E]](Env)\}$$

$\boldsymbol{E}$: Expression $\rightarrow$ Environment $\rightarrow$ $\text{Integer}_{\perp}$

$$E[[E_1\ \text{`+'}\ E_2]](Env) = \boldsymbol{E}[[E_1]](Env) + \boldsymbol{E}[[E_2]](Env)$$
$$E[[E_1\ \text{`}-\text{'}\ E_2]](Env) = \boldsymbol{E}[[E_1]](Env) - \boldsymbol{E}[[E_2]](Env)$$
$$E[[\mathrm{E}_1\ \text{`*'}\ E_2]](Env) = \boldsymbol{E}[[E_1]](Env) * \boldsymbol{E}[[E_2]](Env)$$
$$E[[\text{`('}\ E\ \text{`)'}]](Env) = \boldsymbol{E}[[E]](Env)$$
$$E[[I]](Env) = Env(\ I\ )$$
$$E[[N]](Env) = \boldsymbol{N}[[N]]$$

$\boldsymbol{N}$: Number $\rightarrow$ Integer

$$\boldsymbol{N}[[ND]] = 10*\boldsymbol{N}[[N]]] + \boldsymbol{N}[[D]]$$
$$\boldsymbol{N}[[D]] = D[[D]]$$

$\boldsymbol{D}$: Digit $\rightarrow$ Integer

$$\boldsymbol{D}[[\text{`0'}]] = 0, \boldsymbol{D}[[\text{`1'}]] = 1, \ldots, \boldsymbol{D}[[\text{`9'}]] = 9$$

Figure 12.5 A denotational definition for the sample language extended with assignment statements and environments

12.3.6 Denotational Semantics of Control Statements

To complete our discussion of denotational semantics, we need to extend the denotational definition of Figure 12.5 to if- and while statements, with the following abstract syntax:

$$\begin{aligned}
&S\text{: Statement}\\
&S \rightarrow I\ \text{`:='}\ E\\
&\quad\mid\ \text{`if'}\ E\ \text{`then'}\ L_1\ \text{`else'}\ L_2\ \text{`fi'}\\
&\quad\mid\ \text{`while'}\ E\ \text{`do'}\ L\ \text{`od'}
\end{aligned}$$

As before, the denotational semantics of these statements must be given by a function from environments to environments:

$$\boldsymbol{S} : \text{Statement} \rightarrow \text{Environment} \rightarrow \text{Environment}$$

We define the semantic function of the if statement as follows:

$$\begin{aligned}
&\boldsymbol{S}[[\text{`if'}\ E\ \text{`then'}\ L_1\ \text{`else'}\ L_2\ \text{`fi'}]](Env) =\\
&\qquad \text{if } \boldsymbol{E}[[E]](Env) > 0 \text{ then } \boldsymbol{L}[[L_1]](Env) \text{ else } \boldsymbol{L}[[L_2]](Env)
\end{aligned}$$

Note that we are using the if-then-else construct on the right-hand side to express the construction of a function. Indeed, given *F*: Environment → Integer, *G*: Environment → Environment, and *H*: Environment → Environment, then the function "if *F* then *G* else *H*:" Environment → Environment is given as follows:

$$(\text{if } F \text{ then } G \text{ else } H)\ (Env) = \begin{cases} G(Env), \text{ if } F(Env) > 0 \\ H(Env), \text{ if } F(Env) \leq 0 \end{cases}$$

The semantic function for the while statement is more difficult. In fact, if we let $F = \boldsymbol{S}$[['while' *E* 'do' *L* 'od']], so that *F* is a function from environments to environments, then *F* satisfies the following equation:

$$F(Env) = \text{if } \boldsymbol{E}[[E]](Env) \leq 0 \text{ then } Env \text{ else } F(\boldsymbol{L}[[L]](Env))$$

This is a recursive equation for *F*. To use this equation as a specification for the semantics of *F*, we need to know that this equation has a unique solution in some sense among the functions Environment → Environment. We saw a very similar situation in Section 11.6, where the definition of the factorial function also led to a recursive equation for the function. In that case, we were able to construct the function as a set by successively extending it to a so-called **least-fixed-point solution**, that is, the "smallest" solution satisfying the equation. A similar approach will indeed work here too, and the solution is referred to as the least-fixed-point semantics of the while-loop. The situation here is more complicated, however, in that *F* is a function on the semantic domain of environments rather than the integers. The study of such equations and their solutions is a major topic of domain theory. For more information, see the references at the end of the chapter.

Note that there is an additional problem associated with loops: nontermination. For example, in our sample language, the loop

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

does not terminate. Such a loop does not define any function at all from environments to environments, but we still need to be able to associate a semantic interpretation with it. One does so by assigning it the "undefined" value ⊥ similar to the value of an undefined identifier in an environment. In this case, the domain of environments becomes a lifted domain,

$$\text{Environment}_\perp = (\text{Identifier} \to \text{Integer}_\perp)_\perp$$

and the semantic function for statements must be defined as follows:

$$\boldsymbol{S} : \text{Statement} \to \text{Environment}_\perp \to \text{Environment}_\perp$$

We shall not discuss such complications further.

12.3.7 Implementing Denotational Semantics in a Programming Language

As we have noted previously, it is possible to implement denotational semantic rules directly as a program. Since denotational semantics are based on functions, particularly higher-order functions, it is not surprising that a functional language is a natural choice as an implementation language. In this section, we briefly sketch a possible Haskell implementation for the denotational functions of our sample language. (We choose Haskell because of its minimal extra syntax; ML or Scheme would also be a good choice.)

First, consider how one might define the abstract syntax of expressions in the sample language, using a data declaration:

```
data Expr = Val Int | Ident String | Plus Expr Expr
                    | Minus Expr Expr | Times Expr Expr
```

Here, as with the operational semantics, we are ignoring the semantics of numbers and simply letting values be integers (`Val Int`).

Suppose now that we have defined an Environment type, with a lookup and update operation (a reasonable first choice for it, as in operational semantics, would be a list of string-integer pairs). Then the "***E***" evaluation function can be defined almost exactly as in the denotational definitions:

```
exprE :: Expr -> Environment -> Int
exprE (Plus e1 e2) env = (exprE e1 env) + (exprE e2 env)
exprE (Minus e1 e2) env = (exprE e1 env) - (exprE e2 env)
exprE (Times e1 e2) env = (exprE e1 env) * (exprE e2 env)
exprE (Val n) env = n
exprE (Ident a) env = lookup env a
```

Note that there is no rule for parentheses. Again in this case, the abstract syntax is in tree form, and parentheses simply do not appear.

Similar definitions can be written for statements, statement-lists, and programs. We leave the details as an exercise for the reader.

12.4 Axiomatic Semantics

Axiomatic semantics define the semantics of a program, statement, or language construct by describing the effect its execution has on assertions about the data manipulated by the program. The term "axiomatic" is used because elements of mathematical logic are used to specify the semantics of programming languages, including logical axioms. We discussed logic and assertions in the introduction to logic programming in Chapter 4. For our purposes, however, it suffices to consider logical assertions to be statements about the behavior of a program that are true or false at any moment during execution.

Assertions associated with language constructs are of two kinds: assertions about things that are true just before execution of the construct and assertions about things that are true just after the execution of the construct. Assertions about the situation just before execution are called **preconditions**, and assertions about the situation just after execution are called **postconditions**. For example, given the assignment statement

```
x := x + 1
```

we would expect that, whatever value `x` has just before execution of the statement, its value just after the execution of the assignment is one more than its previous value. This can be stated as the precondition that `x` $= A$ before execution and the postcondition that `x` $= A + 1$ after execution. Standard notation for this is to write the precondition inside curly brackets just before the construct and to write the postcondition similarly just after the construct:

$$\{x = A\}\ \texttt{x := x + 1}\ \{x = A + 1\}$$

or:

$$\{x = A\}$$
$$\texttt{x := x + 1}$$
$$\{x = A + 1\}$$

As a second example of the use of precondition and postcondition to describe the action of a language construct, consider the following assignment:

```
x := 1 / y
```

Clearly, a precondition for the successful execution of the statement is that $y \neq 0$, and then `x` becomes equal to `1/y`. Thus we have

$$\{y \neq 0\}$$
$$\texttt{x := 1 / y}$$
$$\{x = 1/y\}$$

Note that in this example the precondition establishes a restriction that is a requirement for successful execution, while in the first example, the precondition $x = A$ merely establishes a name for the value of `x` prior to execution, without making any restriction whatever on that value.

Precondition/postcondition pairs can be useful in specifying the expected behavior of programs—the programmer simply writes down the conditions he or she expects to be true at each step in a program. For example, a program that sorts the array `a[1]..a[n]` could be specified as follows:

$$\{n \geq 1 \text{ and for all } i,\ 1 \leq i \leq n,\ a[i] = A[i]\}$$
$$\text{sort-program}$$
$$\{\text{sorted}(a) \text{ and permutation}(a, A)\}$$

Here, the assertions sorted(`a`) and permutation(`a`, `A`) mean that the elements of `a` are sorted and that the elements of `a` are the same, except for order, as the original elements of the array `A`.

Such preconditions and postconditions are often capable of being tested for validity during execution of the program, as a kind of error checking, since the conditions are usually Boolean expressions that can be evaluated as expressions in the language itself. Indeed, a few languages such as Eiffel and Euclid have language constructs that allow assertions to be written directly into programs. The C language also has a rudimentary but useful macro library for checking simple assertions: `assert.h`. Using this library and the macro `assert` allows programs to be terminated with an error message on assertion failure, which can be a useful debugging feature:

```
#include <assert.h>
...
assert(y != 0);
x = 1/y;
...
```

If `y` is 0 when this code is executed, the program halts and an error message, such as

```
Assertion failed at test.c line 27: y != 0
Exiting due to signal SIGABRT
...
```

is printed. One can get this same kind of behavior by using exceptions in a language with exception handling:

```
if (y != 0) throw Assertion_Failure();
```

An **axiomatic specification** of the semantics of the language construct *C* is of the form

$$\{P\}\ C\ \{Q\}$$

where *P* and *Q* are assertions; the meaning of such a specification is that, if *P* is true just before the execution of *C*, then *Q* is true just after the execution of *C*.

Unfortunately, such a representation of the action of *C* is not unique and may not completely specify all the actions of *C*. In the second example, for instance, we did not include in the postcondition the fact that `y` $\neq 0$ continues to be true after the execution of the assignment. To specify completely the semantics of the assignment to `x`, we must somehow indicate that `x` is the only variable that changes under the assignment (unless it has aliases). Also, `y` $\neq 0$ is not the only condition that will guarantee the correct evaluation of the expression; `1/y`: `y` > 0 or `y` < 0 will do as well. Thus, writing an expected precondition and an expected postcondition will not always precisely determine the semantics of a language construct.

What is needed is a way of associating to the construct *C* a general relation between precondition *P* and postcondition *Q*. The way to do this is to use the property that programming is a **goal-oriented activity**: We usually know what we want to be true after the execution of a statement or program, and the question is whether the known conditions before the execution will guarantee that this becomes true. Thus, postcondition *Q* is assumed to be given, and a specification of the semantics of *C* becomes a statement of which preconditions *P* of *C* have the property that $\{P\}\ C\ \{Q\}$. To the uninitiated this may seem backward, but it is a consequence of working backward from the goal (the postcondition) to the initial requirements (the precondition).

In general, given an assertion *Q*, there are many assertions *P* with the property that $\{P\}\ C\ \{Q\}$. One example has been given: For `1/y` to be evaluated, we may require that `y` $\neq 0$ or `y` > 0 or `y` < 0. There is one precondition *P*, however, that is the **most general** or **weakest** assertion with the property that $\{P\}\ C\ \{Q\}$. This is called the **weakest precondition** of postcondition *Q* and construct *C* and is written $wp(C,Q)$.

In the example, `y` $\neq 0$ is clearly the weakest precondition such that `1/y` can be evaluated. Both `y` > 0 and `y` < 0 are stronger than `y` $\neq 0$, since they both imply `y` $\neq 0$. Indeed, *P* is by definition weaker than *R* if *R* implies *P* (written in logical form as $R \to P$). Using these definitions we have the following restatement of the property $\{P\}\ C\ \{Q\}$:

$$\{P\}\ C\ \{Q\} \text{ if and only if } P \to wp(C,Q)$$

Finally, we define the axiomatic semantics of the language construct *C* as the function $wp(C,\_)$ from assertions to assertions. This function is a **predicate transformer** in that it takes a predicate as argument and returns a predicate result. It also appears to work backward, in that it computes the weakest precondition from any postcondition. This is a result of the goal-oriented behavior of programs as described earlier.

Our running example of the assignment can now be restated as follows:

$$wp(\texttt{x := 1/y, x = 1/y}) = \{\texttt{y} \neq 0\}$$

As another example, consider the assignment `x := x + 1` and the postcondition $\texttt{x} > 0$:

$$wp(\texttt{x := x + 1, x} > 0) = \{\texttt{x} > \text{-}1\}$$

In other words, for `x` to be greater than 0 after the execution of `x := x + 1`, `x` must be greater than -1 just prior to execution. On the other hand, if we have no condition on `x` but simply want to state its value, we have:

$$wp(\texttt{x := x + 1, x = } A) = \{\texttt{x} = A - 1\}$$

Again, this may seem backward, but a little reflection should convince you of its correctness. Of course, to determine completely the semantics of an assignment such as `x :=` E, where `x` is a variable and E is an expression, we need to compute $wp(\texttt{x :=} E, Q)$ for any postcondition Q. This is done in Section 12.4.2, where the general rule for assignment is stated in terms of substitution. First, we will study *wp* a little further.

12.4.1 General Properties of *wp*

The predicate transformer $wp(C,Q)$ has certain properties that are true for almost all language constructs C, and we discuss these first, before giving axiomatic semantics for the sample language. The first of these is the following:

Law of the Excluded Miracle

$$wp(C,\text{false}) = \text{false}$$

This states that nothing a programming construct C can do will make false into true—if it did it would be a miracle!

The second property concerns the behavior of *wp* with regard to the "and" operator of logic (also called conjunction):

Distributivity of Conjunction

$$wp(C,P \text{ and } Q) = wp(C,P) \text{ and } wp(C,Q)$$

Two more properties regard the implication operator "→" and the "or" operator (also called disjunction):

Law of Monotonicity

$$\text{if } Q \to R \text{ then } wp(C,Q) \to wp(C,R)$$

Distributivity of Disjunction

$$wp(C,P) \text{ or } wp(C,Q) \to wp(C,P \text{ or } Q)$$

with equality if C is deterministic.

The question of determinism adds a complicating technicality to the last law. Recall that some language constructs can be nondeterministic, such as the guarded commands discussed in Chapter 9. An example of the need for a weaker property in the presence of nondeterminism is discussed in Exercise 12.35. However, the existence of this exception serves to emphasize that, when one is talking about *any* language construct C, one must be extremely careful. Indeed, it is possible to invent complex language mechanisms in which all of the foregoing properties become questionable without further conditions. (Fortunately, such situations are rare.)

12.4.2 Axiomatic Semantics of the Sample Language

We are now ready to give an axiomatic specification for our sample language. We note first that the specification of the semantics of expressions alone is not something that is commonly included in an axiomatic specification. In fact, the assertions involved in an axiomatic specificator are primarily statements about the side effects of language constructs; that is, they are statements involving identifiers and environments. For example, the assertion $Q = \{\texttt{x} > 0\}$ is an assertion about the value of x in an environment. Logically, we could think of Q as being represented by the set of all environments for which Q is true. Then logical operations can be represented by set theoretic operations. For example, $P \rightarrow Q$ is the same as saying that every environment for which P is true is in the set of environments for which Q is true—in other words, that P is contained in Q as sets.

We will not pursue this translation of logic into set theory. We will also skip over the specification of expression evaluation in terms of weakest preconditions and proceed directly to statements, environments, and control.

The abstract syntax for which we will define the *wp* operator is the following:

$$\begin{aligned}
&P \rightarrow L\\
&L \rightarrow L_1 \text{ ‘;’ } L_2 \mid S\\
&S \rightarrow I \text{ ‘:=’ } E\\
&\qquad\mid \text{ ‘if’ } E \text{ ‘then’ } L_1 \text{ ‘else’ } L_2 \text{ ‘fi’}\\
&\qquad\mid \text{ ‘while’ } E \text{ ‘do’ } L \text{ ‘od’}
\end{aligned}$$

Syntax rules such as $P \rightarrow L$ and $L \rightarrow S$ do not need separate specifications,[1] since these grammar rules simply state that the *wp* operator for a program *P* is the same as for its associated statement-list *L*, and similarly, if a statement-list *L* is a single statement *S*, then *L* has the same axiomatic semantics as *S*. The remaining four cases are treated in order. To simplify the description we will suppress the use of quotes; code will be distinguished from assertions by the use of a different typeface.

Statement-lists. For lists of statements separated by a semicolon, we have

$$wp(L_1; L_2, Q) = wp(L_1, wp(L_2, Q))$$

This states that the weakest precondition of a series of statements is essentially the composition of the weakest preconditions of its parts. Note that since *wp* works “backward” the positions of L_1 and L_2 are not interchanged, as they are in denotational semantics.

Assignment Statements. The definition of *wp* for the assignment statement is as follows:

$$wp(I := E, Q) = Q[E/I]$$

This rule involves a new notation: $Q[E/I]$. $Q[E/I]$ is defined to be the assertion Q, with E replacing all free occurrences of the identifier I in Q. The notion of “free occurrences” was discussed in Chapter 4; it also

[1]If we did have to write semantics for a program, we would have to change its designation from *P* to *Prog*, say, since we have been using *P* to refer to a precondition.

arose in Section 3.6 in connection with reducing lambda calculus expressions. An identifier I is **free** in a logical assertion Q if it is not **bound** by either the existential quantifier "there exists" or the universal quantifier "for all." Thus, in the following assertion, j is free, but i is bound (and thus not free):

$$Q = (\text{for all } i, a[i] \,.\, a[j])$$

In this case $Q[1/j] = (\text{for all } i, a[i] \,.\, a[1])$, but $Q[1/i] =$ Q. In commonly occurring assertions, this should not become a problem, and in the absence of quantifiers, one can simply read $Q[E/I]$ as replacing all occurrences of I by E.

The axiomatic semantics $wp(I := E,Q) = Q[E/I]$ simply says that, for Q to be true after the assignment $I := E$, whatever Q says about I must be true about E before the assignment is executed.

A couple of examples will help to explain the semantics for assignment.

First, consider the previous example $wp($`x := x + 1 , x` $> 0)$. Here Q = (`x` > 0) and Q[`x` + 1/`x`] = (`x` + 1 > 0). Thus,

$$wp(\texttt{x := x + 1, x} > 0) = (x + 1 > 0) = (x > -1)$$

which is what we obtained before. Similarly,

$$\begin{aligned} wp(\texttt{x ::= x + 1, x} = A) &= (\texttt{x} = A)[(\texttt{x} + 1)/\texttt{x}] \\ &= (\texttt{x} + 1 = A) \\ &= (\texttt{x} = A - 1) \end{aligned}$$

If statements. Recall that the semantics of the if statement in our sample language are somewhat unusual: `if` E `then` L_1 `else` L_2 `fi` means that L_1 is executed if the value of $E > 0$, and L_2 is executed if the value of $E \le 0$. The weakest precondition of this statement is defined as follows:

$$\begin{aligned} &wp(\texttt{if } E \texttt{ then } L_1 \texttt{ else } L_2 \texttt{ fi}, Q) = \\ &\quad (E > 0 \rightarrow wp(L_1,Q)) \text{ and } (E \le 0 \rightarrow wp(L_2,Q)) \end{aligned}$$

As an example, we compute

$$\begin{aligned} &wp(\texttt{if x then x := 1 else x := -1 fi, x} = 1) = \\ &(\texttt{x} > 0 \rightarrow wp(\texttt{x =: 1, x} = 1)) \text{ and } (\texttt{x} \le 0 \rightarrow wp(\texttt{x := -1, x} = 1)) \\ &= (\texttt{x} > 0 \rightarrow 1 = 1) \text{ and } (\texttt{x} \le 0 \rightarrow -1 = 1) \end{aligned}$$

Recalling that $(P \rightarrow Q)$ is the same as Q or not P (see Exercise 12.1), we get

$$(\texttt{x} > 0 \rightarrow 1 = 1) = ((1 = 1) \text{ or not}(\texttt{x} > 0)) = \text{true}$$

and

$$\begin{aligned} &(\texttt{x} \le 0 \rightarrow 1 = 1) = (-1 = 1) \text{ or not}(\texttt{x} \le 0) = \\ &\text{not}(\texttt{x} \le 0) = (\texttt{x} > 0) \end{aligned}$$

so

$$wp(\texttt{if x then x :=1 else x := -1 fi, x} = 1) = (\texttt{x} > 0)$$

as we expect.

While statements. The while statement `while` E `do` L `od`, as defined in Section 12.1, executes as long as $E > 0$. As in other formal semantic methods, the semantics of the while-loop present particular problems. We must give an inductive definition based on the number of times the loop executes. Let H_i (`while` E `do` L `od`, Q) be the statement that the loop executes i times and terminates in a state satisfying Q. Then clearly

$$H_0(\texttt{while } E \texttt{ do } L \texttt{ od}, Q) = E \le 0 \text{ and } Q$$

and

$$\begin{aligned} H_1(\texttt{while } E \texttt{ do } L \texttt{ od}, Q) &= E > 0 \text{ and } wp(L, Q \text{ and } E \le 0) \\ &= E > 0 \text{ and } wp(L, H_0(\texttt{while } E \texttt{ do } L \texttt{ od}, Q)) \end{aligned}$$

Continuing in this fashion we have in general that

$$\begin{aligned} &H_{i+1}(\texttt{while } E \texttt{ do } L \texttt{ od}, Q) = \\ &E > 0 \text{ and } wp(L, H_i(\texttt{while } E \texttt{ do } L \texttt{ od}, Q)) \end{aligned}$$

Now we define

$$\begin{aligned} &wp(\texttt{while } E \texttt{ do } L \texttt{ od}, Q) \\ &= \text{there exists an } i \text{ such that } H_i(\texttt{while } E \texttt{ do } L \texttt{ od}, Q) \end{aligned}$$

Note that this definition of the semantics of the while requires the while-loop to terminate. Thus, a nonterminating loop always has false as its weakest precondition; that is, it can never make a postcondition true. For example,

$$wp(\texttt{while 1 do } L \texttt{ od}, Q) = \text{false, for all } L \text{ and } Q$$

The semantics we have just given for loops has the drawback that it is very difficult to use in the main application area for axiomatic semantics, namely, the proof of correctness of programs. In the next section we will describe an approximation of the semantics of a loop that is more usable in practice.

12.5 Proofs of Program Correctness

The theory of axiomatic semantics was developed as a tool for proving the correctness of programs and program fragments, and this continues to be its major application. In this section we will use the axiomatic semantics of the last section to prove properties of programs written in our sample language.

We have already mentioned in the last section that a specification for a program C can be written as $\{P\}$ C $\{Q\}$, where P represents the set of conditions that are expected to hold at the beginning of a program and Q represents the set of conditions one wishes to have true after execution of the code C. As an example, we gave the following specification for a program that sorts the array `a[1]..a[n]`:

$$\begin{aligned} &\{\texttt{n} \ge 1 \text{ and for all } \texttt{i}, 1 \le \texttt{i} \le \texttt{n}, \texttt{a[i]} = \texttt{A[i]}\} \\ &\text{sort-program} \\ &\{\text{sorted}(\texttt{a}) \text{ and permutation}(\texttt{a}, \texttt{A})\} \end{aligned}$$

Two easier examples of specifications for programs that can be written in our sample language are the following:

1. A program that swaps the value of `x` and `y`:

$$\{\texttt{x} = X \text{ and } \texttt{y} = Y\}$$
$$\texttt{swapxy}$$
$$\{\texttt{x} = Y \text{ and } \texttt{y} = X\}$$

2. A program that computes the sum of integers less than or equal to a positive integer `n`:

$$\{\texttt{n} > 0\}$$
$$\texttt{sum\_to\_n}$$
$$\{\texttt{sum} = 1 + 2 + \ldots + \texttt{n}\}$$

We will give correctness proofs that the two programs we provide satisfy the specifications of (1) and (2).

Recall from the last section that C satisfies a specification $[P]\ C\ [Q]$ provided $P \to wp(C,Q)$. Thus, to prove that C satisfies a specification we need two steps: First, we must compute $wp(C,Q)$ from the axiomatic semantics and general properties of wp, and, second, we must show that $P \to wp(C,Q)$.

1. We claim that the following program is correct:

```
{x = X and y = Y}
t := x;
x := y;
y := t
{x = Y and y = X}
```

We first compute $wp(C,Q)$ as follows:

$$\begin{aligned}
&wp(\texttt{t := x ; x := y ; y := t}, \texttt{x} = Y \text{ and } \texttt{y} = X)\\
&\quad = wp(\texttt{t := x}, wp(\texttt{x := y; y := t}, \texttt{x} = Y \text{ and } \texttt{y} = X))\\
&\quad = wp(\texttt{t := x}, wp(\texttt{x := y}, wp(\texttt{y := t}, \texttt{x} = Y \text{ and } \texttt{y} = X)))\\
&\quad = wp(\texttt{t := x}, wp(\texttt{x := y}, wp(\texttt{y := t}, \texttt{x} = Y)\\
&\qquad\qquad \text{and } wp(\texttt{y := t}, \texttt{y} = X)))\\
&\quad = wp(\texttt{t := x}, wp(\texttt{x := y}, wp(\texttt{y := t}, \texttt{x} = Y))\\
&\qquad\qquad \text{and } wp(\texttt{x := y}, wp(\texttt{y := t}, \texttt{y} = X)))\\
&\quad = wp(\texttt{t := x}, wp(\texttt{x := y}, wp(\texttt{y := t}, \texttt{x} = Y)))\\
&\qquad\qquad \text{and } wp(\texttt{t := x}, wp(\texttt{x := y}, wp(\texttt{y := t}, \texttt{y} = X)))
\end{aligned}$$

by distributivity of conjunction and the axiomatic semantics of statement-lists. Now

$$wp(\texttt{t := x}, wp(\texttt{x := y}, wp(\texttt{y := t}, \texttt{x} = Y))) =$$
$$wp(\texttt{t := x}, wp(\texttt{x := y}, \texttt{x} = Y)) = wp(\texttt{t := x}, \texttt{y} = Y) = (\texttt{y} = Y)$$

and

$$wp(\texttt{t := x}, wp(\texttt{x :=y}, wp(\texttt{y := t}, \texttt{y} = X))) =$$
$$wp(\texttt{t := x}, wp(\texttt{x :=y}, \texttt{t} = X)) = wp(\texttt{t :=} X, \texttt{t} = \texttt{x}) = (\texttt{x} = X)$$

by the rule of substitution for assignments. Thus,

$wp(\texttt{t := x; x := y;y := t,}\ \texttt{x} = Y \text{ and } \texttt{y} = X) = (\texttt{y} = Y \text{ and } \texttt{x} = X)$

The second step is to show that $P \to wp(C,Q)$. But in this case P actually equals the weakest precondition, since $P = (\texttt{x} = X \text{ and } \texttt{y} = Y)$. Since $P = wp(C,Q)$, clearly also $P \to wp(C,Q)$. The proof of correctness is complete.

2. We claim that the following program is correct:

```
{n > 0}
i := n;
sum := 0;
while i do
sum := sum + i;
i := i - 1
od
{sum = 1+2+...+ n}
```

The problem now is that our semantics for while-statements are too difficult to use to prove correctness. To show that a while-statement is correct, we really do not need to derive completely its weakest precondition $wp(\texttt{while ...}, Q)$, but only an **approximation**, that is, some assertion W such that $W \to wp(\texttt{while ...}, Q)$. Then if we can show that $P \to W$, we have also shown the correctness of $\{P\}$ `while ...` $\{Q\}$, since $P \to W$ and $W \to wp(\texttt{while ...}, Q)$ imply that $P \to wp(\texttt{while ...}, Q)$.

We do this in the following way. Given the loop `while` E `do` L `od`, suppose we find an assertion W such that the following three conditions are true:

(a) $W \text{ and } (E > 0) \to wp(L,W)$
(b) $W \text{ and } (E \le 0) \to Q$
(c) $P \to W$

Then if we know that the loop `while` E `do` L `od` terminates, we must have $W \to wp(\texttt{while } E \texttt{ do } L \texttt{ od}, Q)$. This is because every time the loop executes, W continues to be true, by condition (a), and when the loop terminates, condition (b) says Q must be true. Finally, condition (c) implies that W is the required approximation for $wp(\texttt{while ...}, Q)$.

An assertion W satisfying condition (a) is said to be a **loop invariant** for the loop `while` E `do` L `od`, since a repetition of the loop leaves W true. In general, loops have many invariants W, and to prove the correctness of a loop, it sometimes takes a little skill to find an appropriate W, namely, one that also satisfies conditions (b) and (c).

In the case of our example program, however, a loop invariant is not too difficult to find:

$$W = (\texttt{sum} = (\texttt{i} + 1) + \ldots + \texttt{n} \text{ and } \texttt{i} \ge 0)$$

is an appropriate one. We show conditions (a) and (b) in turn:

(a) We must show that W and `i` $> 0 \rightarrow wp($`sum := sum + i ; i := i - 1`$, W)$. First, we have

$wp($`sum := sum + i; i := i - 1`$, W)$
$= wp($`sum := sum + i; i := i - 1`, sum = (i + 1) + . . . + n and i ≥ 0)
$= wp($`sum := sum + i`$, wp($`i := i - 1`, sum = (i + 1) + . . . + n and i ≥ 0))
$= wp($`sum := sum + i`, sum = ((i − 1) + 1) + . . . + n and i − 1 ≥ 0)
$= wp($`sum := sum + i`, sum = i + . . . + n and i − 1 ≥ 0)
= (sum + i = i + . . . + n and i − 1 ≥ 0)
= (sum = (i + 1) + . . . + n and i − 1 ≥ 0)

Now (W and i > 0) → (W and i − 1 ≥ 0), since
W and i > 0 = (sum = (i + 1) + . . . + n and i ≥ 0 and i > 0)
= (sum = (i + 1) + . . . + n and i > 0) →
(W and i − 1 ≥ 0)

Thus, W is a loop invariant.

(b) We must show that (W and (i ≤ 0)) → (sum = 1 + . . . + n).
But this is clear:

W and (i ≤ 0) = (sum = (i + 1) + . . . + n and i ≥ 0 and i ≤ 0)
= (sum = (i + 1) + . . . + n and i = 0)
= (sum = 1 + . . . + n and i = 0)

It remains to show that conditions just prior to the execution of the loop imply the truth of W.[2] We do this by showing that n > 0 → $wp($`i := n; sum := 0`$, W)$. We have

$wp($`i := n;sum := 0`$, W)$
$= wp($`i := n`$, wp($`sum := 0`, sum = (i + 1) + . . . + n and i ≥ 0))
$= wp($`i := n`, 0 = (i + 1) + . . . + n and i ≥ 0)
= (0 = (n + 1) + . . . + n and n ≥ 0)
= (0 = 0 and n ≥ 0)
= (n ≥ 0)

and of course n > 0 → n ≥ 0. In this computation we used the property that the sum (i + 1) + . . . + n with i ≥ n is 0. This is a general mathematical property: Empty sums are always assumed to be 0. We also note that this proof uncovered an additional property of our code: It works not only for n > 0, but for n ≥ 0 as well.

This concludes our discussion of proofs of programs. A few more examples will be discussed in the exercises.

[2]In fact, we should also prove termination of the loop, but we ignore this issue in this brief discussion. Proving correctness while assuming termination is called proving **partial correctness**, which is what we are really doing here.

Exercises

12.1 Our sample language used the bracketing keywords “fi” and “od” for if statements and while-statements, similar to Algol68. Was this necessary? Why?

12.2 Add unary minuses to the arithmetic expressions of the sample language, and add its semantics to **(a)** the operational semantics and **(b)** the denotational semantics.

12.3 Add division to the arithmetic expressions of the sample language, and add its semantics to **(a)** the operational semantics and **(b)** the denotational semantics. Try to include a specification of what happens when division by 0 occurs.

12.4 The operational semantics of identifiers was skipped in the discussion in the text. Add the semantics of identifiers to the operational semantics of the sample language.

12.5 The denotational semantics of identifiers was also (silently) skipped. What we did was to use the set Identifier as both a syntactic domain (the set of syntax trees of identifiers) and as a semantic domain (the set of strings with the concatenation operator). Call the latter set Name, and develop a denotational definition of the semantic function ***I***: Identifier $\rightarrow$ Name. Revise the denotational semantics of the sample language to include this correction.

12.6 A problem that exists with any formal description of a language is that the description itself must be written in some “language,” which we could call the **defining language**, to distinguish it from the defined language. For example, the defining language in each of the formal methods studied in this chapter are as follows:

operational semantics: reduction rules
denotational semantics: functions
axiomatic semantics: logic

For a formal semantic method to be successful, the defining language needs to have a precise description itself, and it must also be understandable. Discuss and compare the defining languages of the three semantic methods in terms of your perception of their precision and understandability.

12.7 One formal semantic method not discussed in this chapter but mentioned in Chapter 3 is the use of the defined language itself as the defining language. Such a method could be called **metacircular**, and metacircular interpreters are a common method of defining LISP-like languages. Discuss the advantages and disadvantages of this method in comparison with the methods discussed in this chapter.

12.8 The grammar of the sample language included a complete description of identifiers and numbers. However, a language translator usually recognizes such constructs in the scanner. Our view of semantics thus implies that the scanner is performing semantic functions. Wouldn’t it be better simply to make numbers and identifiers into tokens in the grammar, thus making it unnecessary to describe something done by the scanner as “semantics?” Why or why not?

12.9 The axiomatic semantics of Section 12.4 did not include a description of the semantics of expressions. Describe a way of including expressions in the axiomatic description.

12.10 Show how the operational semantics of the sample language describes the reduction of the expression 23 * 5 - 34 to its value.

12.11 Compute ***E***[[23 * 5 - 34]] using the denotational definition of the sample language.

12.12 Show how the operational semantics of the sample language describes the reduction of the program `a := 2; b := a + 1; a := b * b` to its environment.

12.13 Compute the value *Env*(*a*) for the environment *Env* at the end of the program `a := 2; b := a + 1; a := b * b` using the denotational definition of the sample language.

12.14 Repeat Exercise 12.12 for the program

```
a := 0 - 11;
if a then a := a else a := 0 - a fi
```

12.15 Repeat Exercise 12.13 for the program of Exercise 12.14.

12.16 Use operational semantics to reduce the following program to its environment:

```
n := 2;
while n do n := n - 1 od
```

12.17 The sample language did not include any input or output statements. We could add these to the grammar as follows:

$$stmt \rightarrow \ldots \mid \text{'input'}\ identifier \mid \text{'output'}\ expr$$

Add input and output statements to the **(a)** operational semantics and **(b)** denotational semantics. (*Hint for denotational semantics:* Consider a new semantic domain IntSequence to be the set of sequences of integers. Then statement sequences act on states that are environments plus an input sequence and an output sequence.)

12.18 An often useful statement in a language is an empty statement, which is sometimes denoted by the keyword `skip` in texts on semantics. Add a `skip` statement to the sample language of this chapter, and describe its **(a)** operational, **(b)** denotational, or **(c)** axiomatic semantics. Why is such a statement useful?

12.19 The sample language has unusual semantics for if- and while-statements, due to the absence of Boolean expressions. Add Boolean expressions such as `x = 0`, `true`, `y > 2` to the sample language, and describe their semantics in **(a)** operational and **(b)** denotational terms.

12.20 Revise the axiomatic description of the sample language to include the Boolean expressions of Exercise 12.19.

12.21 Find *wp*(`a := 2; b := a + 1; a := b * b`, `a` = 9).

12.22 Find *wp*(`if x then x := x else x := 0 - x fi`, `x` ≤ 0).

12.23 Show that the following program is correct with respect to the given specification:

```
{true}
if x then x := x else x := 0 - x fi
{x ≥ 0}
```

12.24 Which of the following are loop invariants of the loop `while i do sum := sum + i; i := i - 1 od`?

(a) `-sum = i + ... + n`
(b) `-sum = (i + 1) + ... + n` and `i` > 0
(c) `sum` ≥ 0 and `i` ≥ 0

12.25 Prove the correctness of the following program:

```
{n > 0}
i := n;
fact := 1;
while i do
fact := fact * i;
i := i - 1
od
{fact = 1 * 2 * ... * n}
```

12.26 Write a program in the sample language that computes the product of two numbers `n` and `m` by repeatedly adding `n m`- times. Prove that your program is correct.

12.27 In Section 12.4 and 12.5 we used the following example of a program specification using preconditions and postconditions:

> {`n` ≥ 1 and for all `i`, 1 ≤ `i` ≤ `n`, `a[i] = A[i]`}
> sort-program
> {sorted(`a`) and permutation(`a`, `A`)}

Write out the details of the assertions sorted(`a`) and permutation (`a`, `A`).

12.28 Show the correctness of the following program using axiomatic semantics:

```
{n > 0}
while n do n := n - 1 od
{n = 0}
```

12.29 Show using general properties of *wp* that *wp*(*C*, not *Q*) → not *wp*(*C*,*Q*) for any language construct *C* and any assertion *Q*.

12.30 Show that the law of monotonicity follows from the distributivity of conjunction. (*Hint:* Use the fact that *P* → *Q* is equivalent to (*P* and *Q*) = *P*.)

12.31 We did not describe how operations might be extended to lifted domains (domains with an undefined value) such as Integer$^{\perp}$. Give a definition for "+," "-," and "*" that includes the undefined value ⊥.

12.32 Sometimes operations can ignore undefined values and still return a defined value. Give an example where the "*" operation can do this.

12.33 The formal semantic descriptions of the sample language in this chapter did not include a description of the semantics in the presence of undefined values (such as the use of an identifier without an assigned value or a loop that never terminates). Try to extend the semantic descriptions of each of the three methods discussed in this chapter to include a description of the effect of undefined values.

12.34 We might be tempted to define an environment in denotational terms as a semantic function from identifiers to integers (ignoring undefined values):

Env: Identifier → Integer

Would this be wrong? Why or why not?

12.35 The text mentions that in the presence of nondeterminism it is not true that *wp*(*C*,*P* or *Q*) = *wp*(*C*,*P*) or *wp*(*C*,*Q*). Let *C* be the following guarded if, in which either statement might be executed:

```
if
    true => x := x + 1
    true => x := x - 1
fi
```

Show that *wp*(*C*, (`x` > 0) or (`x` < 0)) is not equal to *wp*(*C*, `x` > 0) or *wp*(*C*, `x` < 0).

12.36 The operational semantics in Section 12.2 specified a left-to-right evaluation for expressions. Rewrite the reduction rules so that no particular evaluation order is specified.

12.37 The rule for reducing parentheses in operational semantics was written as the axiom'(' *V* ')' => *V*, where *V* stands for a numeric value. Why is it wrong to write the more general rule '(' *E* ')' => *E*, where *E* can be any expression?

12.38 In Section 12.2 we wrote three reduction rules for if statements, but only two for while-statements.

(a) Rewrite the rules for if-statements as only two rules.

(b) Can one write three rules for the while-statement? Why?

12.39 Using the `skip` statement (Exercise 12.18), write a *single* reduction rule for the operational semantics of the while statement.

12.40 Is it possible to express the evaluation order of expressions in denotational semantics? Explain.

12.41 In operational semantics, an environment must be given before an abstract machine can perform any reductions. Thus, one (extremely abstract) view of operational semantics is as a function Φ: Program × Environment→ Environment. In this sense, denotational semantics can be viewed as a Curried version (see Section 12.4.3) of operational semantics. Explain what is meant by this statement.

12.42 Complete the Prolog program sketched in Section 12.2.5 implementing the operational semantics for expressions in the sample language.

12.43 Extend the Prolog program of the previous exercise to include the full sample language.

12.44 Complete the Haskell program sketched in Section 12.3.7 implementing the operational semantics for expressions in the sample language.

12.45 Extend the Haskell program of the previous exercise to include the full sample language.

Notes and References

Formal semantic methods are studied in greater depth in Winskel [1993], Reynolds [1998], Gunter [1992], and Mitchell [1996]. A different operational method is the Vienna definition language, which is surveyed in Wegner [1972]. An outgrowth of VDL is the Vienna development method, or VDM, which is denotational in approach. A description of this method appears in Harry [1997], which includes a description of a related method, called Z.

Denotational semantics as they are presented here began with the early work of Scott and Strachey (see Stoy [1977]). An in-depth coverage of denotational semantics, including domain theory and fixed-point semantics, is given in Schmidt [1986]. Axiomatic semantics began with a seminal paper by Hoare [1969]. Weakest preconditions were introduced by Dijkstra [1975, 1976]. An in-depth coverage is given in Dijkstra and Scholten [1990] and Gries [1981]. For a perspective on nondeterminism and the law of the excluded miracle, see Nelson [1989].

Formal semantic methods as they apply to object-oriented languages are studied in Gunter and Mitchell [1994]. Perhaps the only production language for which a complete formal description has been given is ML (Milner et al. [1997]); a partial formal semantics for Ada83 can be found in Björner and Oest [1981].

One topic not studied in this chapter is the formal semantics of data types and type checking, which are usually given as a set of rules similar to the reduction rules of operational semantics. Details can be found in Hindley [1997] or Schmidt [1994].

CHAPTER

Parallel Programming