

Type Systems and Semantics

3

“Ishmael: Surely all this is not without meaning.”

Herman Melville, Moby Dick

CHAPTER OUTLINE

3.1	TYPE SYSTEMS	51
3.2	SEMANTIC DOMAINS AND STATE TRANSFORMATION	56
3.3	OPERATIONAL SEMANTICS	58
3.4	AXIOMATIC SEMANTICS	60
3.5	DENOTATIONAL SEMANTICS	71
3.6	EXAMPLE: SEMANTICS OF JAY ASSIGNMENTS AND EXPRESSIONS	73

Type systems have become enormously important in language design because they can be used to formalize the definition of a language’s data types and their proper usage in programs. Type systems are often associated with syntax, especially for languages whose programs are type checked at compile time. For these languages, a type system is a definitional extension that imposes specific syntactic constraints (such as the requirement that all variables referenced in a program be declared) that cannot be expressed in BNF or EBNF. For languages whose programs are type checked at run time, a type system can be viewed as part of the language’s semantics. Thus, a language’s type system stands at the bridge between syntax and semantics, and can be properly viewed in either realm.

The definition of a programming language is complete only when its semantics, as well as its syntax and type system, is fully defined. The *semantics* of a programming language is a definition of the *meaning* of any program that is syntactically valid from both the concrete syntax and the static type checking points of view.¹

Program meaning can be defined in several different ways. A straightforward intuitive idea of program meaning is “whatever happens in a (real or model) computer when the program is executed.” A precise characterization of this idea is called *operational semantics*.² Another way to view program meaning is to start with a formal specification of what the program is supposed to do, and then rigorously prove that the program does that by using a systematic series of logical steps. This approach evokes the idea of *axiomatic semantics*. A third way to view the semantics of a programming language is to define the meaning of each type of statement that occurs in the (abstract) syntax as a state-transforming mathematical function. Thus, the meaning of a program can be expressed as a collection of functions operating on the program state. This approach is called *denotational semantics*.

All three semantic definition methods have advantages and disadvantages. Operational semantics has the advantage of representing program meaning directly in the code of a real (or simulated) machine. But this is also a potential weakness, since defining the semantics of a programming language on the basis of any particular architecture, whether it be real or abstract, confines the utility of that definition for compiler-writers and programmers working with different architectures. Moreover, the virtual machine on which instructions execute also needs a semantic description, which adds complexity and can lead to circular definitions.

Axiomatic semantics is particularly useful in the exploration of formal properties of programs. Programmers who must write provably correct programs from a precise set of specifications are particularly well-served by this semantic style. Denotational semantics is valuable because its functional style brings the semantic definition of a language to a high level of mathematical precision. Through it, language designers obtain a functional definition of the meaning of each language construct that is independent of any particular machine architecture.

In this chapter, we introduce a formal approach to the definition of a language’s type system. We also introduce the three models of semantic definition, paying special attention to the denotational model. Denotational semantics is used in later chapters for discussing various concepts in language design and for clarifying various semantic issues. This model is particularly valuable because it also allows us to actively explore these language design concepts in a laboratory setting.

1. Not too long ago, it was possible to write a syntactically correct program in a particular language that would behave differently when run on different platforms (with the same input). This situation arose because the definition of the language’s semantics was not precise enough to require that all of its compilers translate a program to logically equivalent machine language versions. Language designers have realized in recent years that a formal treatment of semantics is as important as the formal treatment of syntax in ensuring that a particular program “means” the same thing regardless of the platform on which it runs. Modern languages are much better in this regard than older languages.

2. Technically, there are two kinds of operational semantics, called “traditional” and “structured” operational semantics (sometimes called “natural semantics”). In this chapter, we discuss the latter.

3.1 TYPE SYSTEMS

A *type* is a well-defined set of values and operations on those values. For example, the familiar type `int` has values $\{\dots, -2, -1, 0, 1, 2, \dots\}$ and operations $\{+, -, *, /, \dots\}$ on those values. The type `boolean` has values $\{\text{true}, \text{false}\}$ and operations $\{\&\&, ||, \text{and } !\}$ on those values.

A *type system* is a well-defined system of associating types with variables and other objects defined in a program. Some languages, like C and Java, associate a single type with a variable throughout the life of that variable at run time, and the types of all variables can be determined at compile time. Other languages, like Lisp and Scheme, allow the type of a variable, as well as its value, to change as the program runs. Languages that are in the former class are often called *statically typed* languages, while languages in the latter are *dynamically typed*.

A statically typed language allows its type rules to be fully defined on the basis of its abstract syntax alone. This definition is often called *static semantics* and it provides a means of adding context-sensitive information to a parser that the BNF grammar cannot provide. This process is identified as *semantic analysis* in Chapter 2, and we will treat it more carefully below.

A *type error* is a run-time error that occurs when an operation is attempted on a value for which it is not well-defined. For example, consider the C expression `x+u.p`, where `u` is defined as the union `{int a; double p;}` and initialized by the assignment:

```
u.a = 1;
```

This expression can raise a type error, since the alternative `int` and `double` values of `u` share the same memory address, and `u` is initialized with an `int` value. If `x` is a `double`, the expression `x+u.p` raises an error that cannot be checked at either compile time or run time.

A programming language is *strongly typed* if its type system allows all type errors in programs to be detected, either at compile time or at run time, before the statement in which they can occur is actually executed. (Whether a language is statically or dynamically typed does not prevent it from being strongly typed.) For example, Java is a strongly typed language, while C is not. That is, in C we can write and execute the expression `x+1` regardless of whether the value of `x` is a number or not. Strong typing generally promotes more reliable programs and is viewed as a virtue in programming language design.

A program is *type safe* if it is known to be free of type errors. All the programs in a strongly typed language are, by definition, type safe. Moreover, a language is type safe if all its programs are. Dynamically typed languages like Lisp must necessarily be type safe. Since all their type checking occurs at run time, variables' types can change dynamically. This places a certain overhead on the runtime performance of a program, since executable type-checking code must be intermingled with the code written by the programmer.

How can we define a type system for a language so that type errors can be detected? This question is addressed in the next section.

3.1.1 Formalizing the Type System

One way to define a language's type system is to write a set of function specifications that define what it means for a program to be type safe.³ These rules can be written as boolean-valued functions, and can express ideas like “all declared variables have unique names” or “all variables used in the program must be declared.”

Since the rules for writing type-safe programs can be expressed functionally, they can be implemented in Java as a set of boolean-valued methods. The basis for this functional definition is a *type map*, which is a set of pairs representing the declared variables and their types. We can write the type map, *tm*, of a program as follows:

$$tm = \{\langle v_1, t_1 \rangle, \langle v_2, t_2 \rangle, \dots, \langle v_n, t_n \rangle\}$$

where each v_i denotes a *Variable* and each t_i denotes its declared *Type*. For the language Jay, the *Variables* in a type map are mutually unique and every type is taken from a set of available types that are fixed at compile time. For Jay, that set is fixed permanently (it is `{int, boolean}`). For most languages, like C and Java, that set can be expanded by the programmer by defining new types (typedef's in C, and classes in Java).⁴ In any case, here is an example type map for a program that has three variables declared; *i* and *j* with type `int` and *p* with type `boolean`:

$$tm = \{\langle i, \text{int} \rangle, \langle j, \text{int} \rangle, \langle p, \text{boolean} \rangle\}$$

A formal treatment of static type checking for a program relies on the existence of a type map that has been extracted out of the *Declarations* that appear at the top of the program. We define the function *typing* for Jay as follows:

$$\text{typing: } \text{Declarations} \rightarrow \text{TypeMap}$$

$$\text{typing}(\text{Declarations } d) = \bigcup_{i \in \{1, \dots, n\}} \langle d_i . v, d_i . t \rangle$$

This function is easily implemented in Java as follows:

```
TypeMap typing (Declarations d) {
    TypeMap map = new TypeMap();
    for (int i=0; i<d.size(); i++) {
        map.put (((Declaration)(d.elementAt(i))).v,
                ((Declaration)(d.elementAt(i))).t);
    }
    return map;
}
```

3. It is important to note that the BNF-defined concrete syntax of a language is not adequate for defining its type checking requirements. That is, BNF is not a powerful enough device to express ideas like “all declared variables must have unique names.” This important limitation of BNF grammars is usually exposed in a study of formal (Chomsky-type) languages, and would occur in a course on the theory of computation or compilers.

4. Moreover, the rules for naming variables in C and Java are more flexible and complex, taking into account the syntactic *environment* in which the variable is declared. We revisit this topic in Chapter 5.

Thus, given a Vector of *Declarations* d_i , the method `typing` returns a Java `TypeMap` whose keys are the declared variables $d_i.v$ and whose values are their respective types $d_i.t$; in actuality, a `TypeMap` is an extension of a Java `Hashtable`. Here, we are assuming that the abstract syntax for *Declaration* is defined in Java as follows (see Appendix B):

```
class Declaration {
    Variable v;
    Type t;
}
```

Static type checking for a language can be expressed in functional notation, in which each rule that helps define the type system is a boolean-valued function V (meaning “valid”). V returns **true** or **false** depending on whether or not a particular member of an abstract syntactic class is valid, in relation to these rules. That is,

$V: \text{Class} \rightarrow \mathbf{B}$

For example, suppose we want to define the idea that “a list of declarations is valid if all its variables have mutually unique identifiers.” This idea can be expressed precisely as follows:

$V: \text{Declarations} \rightarrow \mathbf{B}$

$V(\text{Declarations } d) = \forall i, j \in \{1, \dots, n\}: (i \neq j \supset d_i.v \neq d_j.v)$

That is, every distinct pair of variables in a declaration list has mutually different identifiers. Recall that the i th variable in a declaration list has two parts, a *Variable* v and a *Type* t . This particular validity rule addresses only the mutual uniqueness requirement for variables. In a real language, it is important that this function definition also specify that the type of each variable be taken from the set of available types (such as `{int, boolean}` for Jay). This refinement is left as an exercise.

Given this function, the implementation of type checking in Java is not a difficult exercise. As a basis, we can use the abstract syntax which is defined as a set of classes. Assuming that `Declarations` is implemented as a Java `Vector` (see Appendix B), the method V for `Declarations` can be viewed as one of a collection of methods V which together define the entire static type checking requirements of a language. Here is the Java method V for declarations:

```
public boolean V (Declarations d) {
    for (int i=0; i<d.size() - 1; i++)
        for (int j=i+1; j<d.size(); j++)
            if (((Declaration)(d.elementAt(i))).v).equals
                (((Declaration)(d.elementAt(j))).v))
                return false;
    return true;
}
```

In the interest of efficiency, this method does not exactly mirror the definition given in function V for uniqueness among declared variable names. That is, the inner loop

controlled by j does not cover the entire range $\{1, \dots, n\}$ as suggested by the formal definition of this function. Readers should recognize, however, that this implementation is an effective one. Note also that, if we were developing a type checker for a compiler, we would include a device for reporting useful error messages.

Another common feature of a language's typing rules is that every variable referenced from within a statement in a program must be declared. Moreover, every variable used in an arithmetic expression must have a proper arithmetic type in its declaration. These requirements can be defined using the classes *Statement* and *Expression*, contexts in which the variables actually occur. These requirements also use the information in the program's type map as a basis for checking the existence and type conformance for each variable referenced.

For a complete language, a set of functions V will be defined over all of its abstract syntactic classes, including the most general class *Program* itself. A complete program from the abstract syntax point of view has two parts; a series of *Declarations* and a body which is a *Block*. That is,

```
class Program {
    Declarations decpart;
    Block body;
}
```

A complete type check, therefore, can be defined functionally at this most general level by the function V , defined as follows:

$$V: \text{Program} \rightarrow \mathbf{B}$$

$$V(\text{Program } p) = V(p.\text{decpart}) \wedge V(p.\text{body}, \text{typing}(p.\text{decpart}))$$

That is, if the *Declarations* have mutually unique variable names and every statement in the body of the program is valid with respect to the type map generated from the declarations, then the entire program is valid from a type checking point of view.

Our definition of the abstract syntax as a collection of Java classes makes this particular specification simple to implement.

```
public boolean V (Program p) {
    return V(p.decpart) && V(p.body, typing (p.decpart));
}
```

That is, a program has two parts, a *decpart*, which is a *Declarations*, and a *body*, which is a *Block*. The type validity of the program's *Block* can be checked only in relation to the particular type map that represents its declarations.

Following is a summary of the all the type-checking requirements of Jay. The details of type validity functions for the individual expression and statement classes in a Jay program are covered in Chapter 4, where we discuss the design of imperative languages in greater generality.

3.1.2 Type Checking in Jay

Jay is a statically and strongly typed language. Thus, all type errors can be checked at compile time, before the program is executed. Here is an informal summary of the type checking requirements of Jay.

- Each declared *Variable* must have a unique *Identifier*.
- Each *Variable*'s type must be either `int` or `boolean`.
- Each *Variable* referenced within any *Expression* in the program must have been declared.
- Every *Expression*'s result type is determined as follows:
 - If the *Expression* is a simple *Variable* or *Value*, then its result type is the type of that *Variable* or *Value*.
 - If the *Expression*'s *Operator* is arithmetic (`+`, `-`, `*`, or `/`) then its terms must all be type `int` and its result type is correspondingly `int`. For example, the *Expression* `x+1` requires `x` to be `int` (since `1` is `int`), and its result type is `int`.
 - If the *Operator* is relational (`<`, `<=`, `>`, `>=`, `==`, `!=`) then its terms must be type `int` and its result type is `boolean`.
 - If the *Operator* is boolean (`&&`, `||`, or `!`) its term(s) must be type `boolean` and its result type is `boolean`.
- For every *Assignment* statement, the type (`int` or `boolean`) of its target variable must agree with the type of its source expression.
- For every *Conditional* and *Loop*, the type of its expression must be `boolean`.

To illustrate these requirements, consider the simple Jay program given below:

```
// compute result = the factorial of integer n
void main () {
    int n, i, result;
    n = 8;
    i = 1;
    result = 1;
    while (i < n) {
        i = i + 1;
        result = result * i;
    }
}
```

In this program, the variables `n`, `i`, and `result` used in the program must be declared. Moreover, the *Expressions* `result*i` and `i+1` have both of their operands of type `int`. Further, the *Expression* `i<=n` has type `boolean`, since the operator `<=` takes operands of type `int` and returns a `boolean` result. Finally, each of the *Assignments* in this program has an `int` type *Variable* as its target and an `int` type *Expression* as its source. Overall, these features make the simple Jay program above valid with respect to the type rules given above. These ideas will be more formally presented and carefully discussed in Chapter 4.

3.2 SEMANTIC DOMAINS AND STATE TRANSFORMATION

The natural numbers, integers, real numbers, and booleans and their mathematical properties provide a foundational context for programming language design. For instance, the language Jay relies on our mathematical intuition about the existence and behavior of \mathbf{N} (the natural numbers), \mathbf{I} (the integers), \mathbf{B} (the set $\{true, false\}$ of boolean values), and \mathbf{S} (the set of character strings).

These sets are instances of *semantic domains* for programming languages. A semantic domain is any set whose properties and operations are independently well-understood and upon which the functions that define the semantics of a language are ultimately based.

Three useful semantic domains for programming languages are the environment, the memory, and the locations. The *environment* γ is a set of pairs that unite specific variables with memory locations. The *memory* μ is a set of pairs that unite specific locations with values. The *locations* in each case are the natural numbers \mathbf{N} .

For example, suppose we have variables i and j with values 13 and -1 at some time during the execution of a program. Suppose that the memory locations are serially numbered beginning at 0, and the variables i and j are associated with the memory locations 154 and 155 at that time. Then the environment and memory represented by this configuration can be expressed as follows:⁵

$$\begin{aligned}\gamma &= \{\langle i, 154 \rangle, \langle j, 155 \rangle\} \\ \mu &= \{\langle 0, \text{undef} \rangle, \dots, \langle 154, 13 \rangle, \langle 155, -1 \rangle, \dots\}\end{aligned}$$

The *state* σ of a program is the product of its environment and its memory. However, for our introductory discussions in this chapter, it is convenient to represent the state of a program in a more simplified form. This representation takes the memory locations out of play and simply defines the *state* σ of a program as a set of pairs $\langle v, val \rangle$ that represents all the active variables and their currently assigned values at some stage during the program's execution.⁶ This is expressed as follows:

$$\sigma = \{\langle v_1, val_1 \rangle, \langle v_2, val_2 \rangle, \dots, \langle v_m, val_m \rangle\}$$

Here, each v_i denotes a variable and each val_i denotes its currently assigned value. Before the program begins execution, $\sigma = \{\langle v_1, \text{undef} \rangle, \langle v_2, \text{undef} \rangle, \dots, \langle v_m, \text{undef} \rangle\}$. We also use the expression $\sigma(v)$ to denote the function that retrieves the value of the variable v from the current state.

5. The special value *undef* is used to denote the value of a memory location (variable) that is currently undefined (not yet assigned).

6. We shall expand this definition of state in Chapter 5, when it will be important to formally represent the environment in a more dynamic fashion. There, a dynamic environment allows us to precisely characterize the ideas of run-time stack and heap, and thus deal effectively with procedure call, object creation, parameter passing, recursion, and so on. For the elementary language Jay, the environment is static, so the state can be simply represented as a set of variable name-value pairs.

For example, the expression below describes the state of a program that has computed and assigned values to three variables, x , y , and z , after several statements have been executed:

$$\sigma = \{\langle x, 1 \rangle, \langle y, 2 \rangle, \langle z, 3 \rangle\}$$

For this particular state, we can retrieve the value of a variable, say y , by writing the expression $\sigma(y)$, which gives the value 2. The next step in the program might be an assignment statement, such as

$$y = 2 * z + 3;$$

whose effect would be to change the state as follows:

$$\sigma = \{\langle x, 1 \rangle, \langle y, 9 \rangle, \langle z, 3 \rangle\}$$

The next step in the computation might assign a value to a fourth variable, as in the assignment

$$w = 4;$$

resulting in the following state transformation:

$$\sigma = \{\langle x, 1 \rangle, \langle y, 9 \rangle, \langle z, 3 \rangle, \langle w, 4 \rangle\}$$

State transformations that represent these types of assignments can be represented mathematically by a special function called the *overriding union*, represented by the symbol $\bar{\cup}$. This function is similar to the ordinary set union, but differs exactly in the way that can represent transformations like the ones illustrated above.

Specifically, $\bar{\cup}$ is defined for two sets of pairs X and Y as follows:

$X \bar{\cup} Y =$ replace in X all pairs $\langle x, v \rangle$ whose first member matches a pair $\langle x, w \rangle$ from Y by $\langle x, w \rangle$ and then add to X any remaining pairs in Y

For example, suppose $\sigma_1 = \{\langle x, 1 \rangle, \langle y, 2 \rangle, \langle z, 3 \rangle\}$ and $\sigma_2 = \{\langle y, 9 \rangle, \langle w, 4 \rangle\}$. Then $\sigma_1 \bar{\cup} \sigma_2 = \{\langle x, 1 \rangle, \langle y, 9 \rangle, \langle z, 3 \rangle, \langle w, 4 \rangle\}$

Another way to visualize the overriding union is through the natural join of these two sets. The *natural join* $\sigma_1 \otimes \sigma_2$ is the set of all pairs in σ_1 and σ_2 that have the same first member. For example,

$$\{\langle x, 1 \rangle, \langle y, 2 \rangle, \langle z, 3 \rangle\} \otimes \{\langle y, 9 \rangle, \langle w, 4 \rangle\} = \{\langle y, 2 \rangle, \langle y, 9 \rangle\}$$

Furthermore, the set difference between σ_1 and $\sigma_1 \otimes \sigma_2$, expressed as $\sigma_1 - (\sigma_1 \otimes \sigma_2)$, effectively removes every pair from σ_1 whose first member is identical with the first member of some pair in σ_2 . Thus, the expression

$$(\sigma_1 - (\sigma_1 \otimes \sigma_2)) \cup \sigma_2$$

is equivalent to $\sigma_1 \bar{\cup} \sigma_2$.

Readers may have observed that the operator $\bar{\cup}$ is a formal and generalized model of the familiar assignment operation in imperative programming. It thus plays a pivotal role in the formal semantics of imperative programming languages like C/C++, Ada, Java, and many more.

3.3 OPERATIONAL SEMANTICS

The *operational semantics* of a program provides a definition of program meaning by simulating the program's behavior on a machine model that has a very simple (though not necessarily realistic) instruction set and memory organization. An early operational semantics model was the "SECD machine" [Landin 1966], which provided a basis for formally defining the semantics of Lisp.

Structured operational semantics models use a rule-based approach and a few simple assumptions about the arithmetic and logical capabilities of the underlying machine. Below we develop this kind of operational semantics for Jay, using its abstract syntax as a starting point.

Our operational semantics model uses the notation $\sigma(e) \Rightarrow v$ to represent the computation of a value v from the expression e in state σ . If e is a constant, $\sigma(e)$ is just the value of that constant in the underlying semantic domain. If e is a variable, $\sigma(e)$ is the value of that variable in the current state σ . The two semantic domains for Jay are **I** and **B**, so the function $\sigma(e)$ will deliver either an integer or a boolean value for any Jay expression e .⁷

The second notational convention for operational semantics is that of an *execution rule*. An execution rule has the form $\frac{\textit{premise}}{\textit{conclusion}}$, which is to be read, "if the *premise* is true, then the *conclusion* is true." Consider the execution rule for addition in Jay, given below:

$$\frac{\sigma(e_1) \Rightarrow v_1 \quad \sigma(e_2) \Rightarrow v_2}{\sigma(e_1 + e_2) \Rightarrow v_1 + v_2}$$

This defines addition on the basis of the sums of the current values of expressions e_1 and e_2 , using the properties of the domain of integers in which the values v_1 and v_2 reside. The operational semantics of the other Jay arithmetic, relational, and boolean operators is defined in a similar way.

For statements, the operational semantics is defined by a separate execution rule for each statement type in the abstract syntax of Jay. Here is the operational semantics for an assignment statement s of the form $s.target = s.source$:

$$\frac{\sigma(s.source) \Rightarrow v}{\sigma(s.target = s.source;) \Rightarrow \sigma \bar{\cup} \{\langle s.target, v \rangle\}}$$

For example, suppose we have the assignment $x = x + 1$; and a current state in which the value of x is 5. That is, $\sigma = \{ \dots, \langle x, 5 \rangle, \dots \}$. The operational semantics of constants, variables, expressions, and assignment, therefore, computes a new state in the following series of execution rule applications (reading from top to bottom):

$$\frac{\frac{\sigma(x) \Rightarrow 5 \quad \sigma(1) \Rightarrow 1}{\sigma(x + 1) \Rightarrow 6}}{\sigma(x = x + 1;) \Rightarrow \{ \dots, \langle x, 5 \rangle, \dots \} \bar{\cup} \{\langle x, 6 \rangle\}}$$

This leaves the state transformed to $\sigma = \{ \dots, \langle x, 6 \rangle, \dots \}$.

7. Note the distinction between the two uses of σ here: when used alone, it denotes a *state*, or a set of name-value pairs; when written as $\sigma(e)$, it denotes a *function* that maps expressions to values.

The execution rule for statement sequences s_1s_2 is also intuitively defined:

$$\frac{\sigma(s_1) \Rightarrow \sigma_1 \quad \sigma_1(s_2) \Rightarrow \sigma_2}{\sigma(s_1s_2) \Rightarrow \sigma_2}$$

That is to say, if execution of s_1 in state σ yields state σ_1 , and execution of s_2 in state σ_1 yields state σ_2 , then the compound effect of executing the sequence s_1s_2 in state σ is state σ_2 .

For Jay conditionals, which are of the form $s = \text{if } (s.\text{test}) \ s.\text{thenpart} \ \text{else} \ s.\text{elsepart}$, the execution rule has two versions, one to be applied when the evaluation of $s.\text{test}$ is *true* and the other to be applied when $s.\text{test}$ is *false*.

$$\frac{\sigma(s.\text{test}) \Rightarrow \text{true} \quad \sigma(s.\text{thenpart}) \Rightarrow \sigma_1}{\sigma(\text{if } (s.\text{test})s.\text{thenpart} \ \text{else} \ s.\text{elsepart}) \Rightarrow \sigma_1}$$

$$\frac{\sigma(s.\text{test}) \Rightarrow \text{false} \quad \sigma(s.\text{elsepart}) \Rightarrow \sigma_1}{\sigma(\text{if } (s.\text{test})s.\text{thenpart} \ \text{else} \ s.\text{elsepart}) \Rightarrow \sigma_1}$$

The interpretation is straightforward. If we have state $\sigma = \{\dots, \langle x, 6 \rangle, \dots\}$ and the following Jay statement:

```
if (x<0)
  x = x - 1;
else
  x = x + 1;
```

then the second rule is the only one whose premise $\sigma(x < 0) \Rightarrow \text{false}$ is satisfied by the current value of x . This rule concludes that the result of executing the conditional is the same as the result of executing the assignment $x = x + 1$; in state $\sigma = \{\dots, \langle x, 6 \rangle, \dots\}$, leaving as a final result the state $\sigma_1 = \{\dots, \langle x, 7 \rangle, \dots\}$.

Loops are statements with the general form $s = \text{while } (s.\text{test}) \ s.\text{body}$. Their execution rules have two different forms, one of which is chosen when the condition $s.\text{test}$ is *true* and the other when $s.\text{test}$ is *false*.

$$\frac{\sigma(s.\text{test}) \Rightarrow \text{true} \quad \sigma(s.\text{body}) \Rightarrow \sigma_1 \quad \sigma_1(\text{while } (s.\text{test}) \ s.\text{body}) \Rightarrow \sigma_2}{\sigma(\text{while } (s.\text{test}) \ s.\text{body}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(s.\text{test}) \Rightarrow \text{false}}{\sigma(\text{while } (s.\text{test}) \ s.\text{body}) \Rightarrow \sigma}$$

The first rule says, in effect, that when $s.\text{test}$ is *true* we execute $s.\text{body}$ one time and achieve the state σ_1 , and then recursively execute the loop starting in this new state. The second rule provides an exit from the loop; if $s.\text{test}$ is *false*, we complete the loop without changing the state σ .

For example, suppose we have the following loop in Jay, which doubles the value of x until $x < 100$ is no longer *true*:

```
while (x<100)
  x = 2*x;
```

Beginning with initial state $\sigma = \{ \dots, \langle x, 7 \rangle \}$, the first rule recursively applies, giving the following conclusion:

$$\frac{\sigma(x < 100) \Rightarrow true \quad \sigma(x = 2*x;) \Rightarrow \sigma_1 = \{ \dots, \langle x, 14 \rangle \} \quad \sigma_1(\text{while } (x < 100) \text{ } x = 2*x;) \Rightarrow \sigma_2}{\sigma(\text{while } (x < 100) \text{ } x = 2*x;) \Rightarrow \sigma_2}$$

Now, to compute the final state σ_2 , we need to reapply this rule starting from the new state $\sigma_1 = \{ \dots, \langle x, 14 \rangle \}$. A series of applications for this rule eventually leaves the state $\sigma_1 = \{ \dots, \langle x, 112 \rangle \}$, in which case the premise $\sigma(x < 100) \Rightarrow false$ is satisfied and the final state for executing the loop becomes $\sigma_2 = \{ \dots, \langle x, 112 \rangle \}$.

3.4 AXIOMATIC SEMANTICS

While it is important to programmers and compiler-writers to understand what a program does in all circumstances, it is also important for programmers to be able to confirm, or *prove*, that it does what it is supposed to do under all circumstances. That is, if someone presents the programmer with a specification for what a program is supposed to do, the programmer may need to be able to prove, beyond a reasonable doubt, that the program and this specification are absolutely in agreement with each other. That is, the program is “correct” in some convincing way. *Axiomatic semantics* provides a vehicle for developing such proofs.

For instance, suppose we want to prove mathematically that the C/C++ function *Max* in Figure 3.1 actually computes as its result the maximum of its two parameters: *a* and *b*.

Figure 3.1 A C/C++ Max Function

```
int Max (int a, int b) {
    int m;
    if (a >= b)
        m = a;
    else
        m = b;
    return m;
}
```

Calling this function one time will obtain an answer for a particular *a* and *b*, such as 8 and 13. But the parameters *a* and *b* define a wide range of integers, so calling it several times with all the different values to prove its correctness would be an infeasible task.

Axiomatic semantics provides a vehicle for reasoning about programs and their computations. This allows programmers to predict a program’s behavior in a more circumspect and convincing way than running the program several times using as test cases different random choices of input values.

3.4.1 Fundamental Concepts

Axiomatic semantics is based on the notion of an *assertion*, which is a predicate that describes the *state* of a program at any point during its execution. An assertion can define the meaning of a computation, like “*the maximum of a and b*,” without concern for how that computation is accomplished. For instance, the code in Figure 3.1 is just one way of algorithmically expressing the maximum computation; even for a function this

simple, there are many minor variations. In any case, the following assertion Q describes the function Max declaratively, without regard for the underlying computational process:

$$Q \equiv m = \max(a, b)$$

This predicate defines the meaning of the function $Max(a, b)$ for any integer values of a and b . To prove that the program in Figure 3.1 actually computes $Max(a, b)$, we need to show that the logical expression Q is somehow equivalent in meaning to that program. Q is called a *postcondition* for the program Max .

Axiomatic semantics allows us to logically derive a series of predicates by reasoning about the behavior of each individual statement in the program, beginning with the postcondition Q and the last statement and working backwards. The last predicate, say P , that is derived in this series of steps is called the program's *precondition*. The precondition thus expresses what must be *true* before program execution begins in order for the postcondition to be satisfied.

In the case of Max , the postcondition Q is well-defined for all integer values of a and b . This suggests the following precondition:

$$P \equiv true$$

That is, for the program to be considered for correctness at all, no assumption or precondition is needed.

One final consideration must be taken into account before we look at the details of correctness proofs themselves. That is, for *some* initial values of the variables that satisfy the program's precondition P , executing the program may *never* reach its last statement. This may occur either because the program enters an infinite loop or because it may try to do a calculation that exceeds the capabilities of the machine on which it is running. For example, if we try to compute $n!$ for a large enough value of n ,⁸ the program will raise an arithmetic overflow condition and halt, thus never reaching its final goal. In general, we must be content to prove correctness of programs only partially—that is, only for those selections of initial values of variables that allow the execution of all its statements to be completed. This notion is called *partial correctness*.

These concerns notwithstanding, we can prove the (partial) correctness of a program by placing its precondition in front of its first statement and its postcondition after its last statement, and then systematically deriving a series of valid predicates as we simulate the execution of each instruction in its turn. For any statement or series of statements s , the following expression

$$\{P\}s\{Q\}$$

represents the predicate that the statement s is partially correct with respect to the precondition P and the postcondition Q . The expression $\{P\}s\{Q\}$ is called a *Hoare triple*⁹ and reads “execution of statements s , beginning in a state that satisfies P , results in a state that satisfies Q , provided that s halts.”

8. $30!$ exceeds the size of a 32-bit integer.

9. These forms are called *Hoare triples* since they were first characterized by C.A.R. Hoare in the original proposal for axiomatizing the semantics of programming languages [Hoare 1969].

For our example program, we first write the following Hoare triple:

```
{true}
  if (a >= b)
    m = a;
  else
    m = b;
{m = max(a, b)}
```

To prove the validity of this Hoare triple, we derive intermediate Hoare triples $\{P\}s\{Q\}$ that are valid for individual statements s , beginning with the postcondition. This process continues until we have successfully shown that the above Hoare triple is *true* for all initial values of a and b for which the program halts. That is, if we can derive Hoare triples that logically connect the individual lines in a program with each other, we will effectively connect the program's postcondition with its precondition.

How are these intermediate triples derived? That is done by formalizing what we know about the behavior of each type of statement in the language. Programs in Jay-like languages have four basic types of statements: assignments, conditionals, loops, and blocks (sequences). Each statement type has a *proof rule* which defines the meaning of that statement type in terms of the pre- and postconditions that it satisfies. The proof rules for Jay-like languages are shown in Table 3.1.

Table 3.1

Proof Rules for
Different Types of
Jay Statements

Statement Type	Proof Rule
1. <i>Assignment</i> $s = s.target = s.source;$	$\frac{true}{\{Q[s.target \setminus s.source]\}s\{Q\}}$
2. <i>Sequence (Block)</i> $s = s_1 s_2$	$\frac{\{P\}s_1\{R\} \quad \{R\}s_2\{Q\}}{\{P\}s_1 s_2\{Q\}}$
3. <i>Conditional</i> $s = \text{if } (s.test) \text{ } s.thenpart$ $\quad \text{else } s.elsepart$	$\frac{\{s.test \wedge P\}s.thenpart\{Q\} \quad \{\neg s.test \wedge P\}s.elsepart\{Q\}}{\{P\}s\{Q\}}$
4. <i>Loop</i> $s = \text{while } (s.test) \text{ } s.body$	$\frac{\{s.test \wedge P\}s.body\{P\}}{\{P\}s\{\neg s.test \wedge P\}}$
5. <i>Rule of consequence</i>	$\frac{P \supset P' \quad \{P'\}s\{Q'\} \quad Q' \supset Q}{\{P\}s\{Q\}}$

These proof rules are all of the form $\frac{premise}{conclusion}$, which is similar to the execution rules used in operational semantics. However, proof rules are to be read, “if the *premise* is valid then the *conclusion* is valid.”

The *Assignment* proof rule has *true* as its premise, guaranteeing that we can always derive the conclusion; such a rule is called an *axiom*. The notation $Q[a/b]$ means “the state that results from replacing the value of b in Q by a .” For instance, if

$\{Q\} \equiv \{x = 1 \wedge y = 4\}$ then $\{Q[1/x]\} \equiv \{1 = 1 \wedge y = 4\}$. Applied to an assignment, rule 1 suggests that the following Hoare triple is valid:

$$\begin{array}{l} \{a = \max(a, b)\} \\ \quad m = a; \\ \{m = \max(a, b)\} \end{array}$$

That is, the replacement of m in the postcondition by a results in precondition of $\{a = \max(a, b)\}$. So the assignment rule allows us to reason backwards through a program, deriving preconditions from postconditions in individual statements.

Proof rule 5 allows us to perform arithmetic and logical simplification in a predicate during the proof process. In the above example, for instance, the assertion $\{a \geq b \wedge a = \max(a, b)\}$ is implied by $\{a \geq b\}$, so that we can substitute it and form an equivalent Hoare triple using rule 5 as follows:

$$\frac{a \geq b \supset a = \max(a, b) \quad \{a \geq b \wedge a = \max(a, b)\} m = a; \{m = \max(a, b)\}}{\{a \geq b\} m = a; \{m = \max(a, b)\}}$$

This example also suggests that there may be several alternative preconditions that can be derived from a given statement and postcondition, using the proof rules. That precondition which is the least restrictive on the variables in play is called the *weakest precondition*. For instance, the precondition $\{a \geq b\}$ is the weakest precondition for the statement $m = a$; and its postcondition $\{m = \max(a, b)\}$. Finding weakest preconditions is important because it enables simplification of the proof at various stages.

A strategy for proving the partial correctness of the rest of the program in Figure 3.1 works systematically from the postcondition backwards through the if, and then through the assignment statements in the then and else parts toward the given precondition.

Next, using the rule 1 for assignments and our postcondition on the else part of the if statement, we obtain:

$$\begin{array}{l} \{b = \max(a, b)\} \\ \quad m = b; \\ \{m = \max(a, b)\} \end{array}$$

As before, we use $a < b$ and rule 5 to show:

$$\frac{a < b \supset b = \max(a, b) \quad \{a < b \wedge b = \max(a, b)\} m = b; \{m = \max(a, b)\}}{\{a < b\} m = b; \{m = \max(a, b)\}}$$

Having proved both premises of rule 3 for a conditional, we can conclude:

$$\begin{array}{l} \{true\} \\ \quad \text{if } (a \geq b) \\ \quad \quad m = a; \\ \quad \text{else} \\ \quad \quad m = b; \\ \{m = \max(a, b)\} \end{array}$$

In Subsection 3.4.2, we prove the correctness of a program involving a loop.

3.4.2 Correctness of Factorial

Suppose we want to prove mathematically that the C/C++ function *Factorial* in Figure 3.2 actually computes as its result $n!$, for any integer n where $n \geq 1$. By $n!$ we mean the product $1 \times \cdots \times n$.

Figure 3.2 A C/C++ Factorial Function

```
int Factorial (int n) {
    int f = 1;
    int i = 1;
    while (i < n) {
        i = i + 1;
        f = f * i;
    }
    return f;
}
```

So the precondition or input assertion Q for *Factorial* is $1 \leq n$, while the postcondition or output assertion is $f = n!$. In general in a program involving a loop, rule 4 of Table 3.1 is used to break the code into three parts, as follows:

```
{Q}
initialization
{P}
while (test) {
    loopBody
}
{¬test ∧ P}
finalization
{R}
```

where Q is our input assertion, R is our final or output assertion, and P is the loop *invariant*. An invariant is an assertion that remains *true* for every iteration of the loop. In general, there is no algorithmic way to derive a loop invariant from the input-output assertions for the program.¹⁰ Thus, it is important for proving program correctness that the loop invariant be supplied for each loop.

For the *Factorial* function given in Figure 3.2, the loop invariant P is $\{1 \leq i \wedge i \leq n \wedge (f = i!)\}$. Thus, the *Factorial* program with its input-output assertions and loop invariant can be rewritten as:

```
{1 ≤ n}
    f = 1;
    i = 1;
{1 ≤ i ∧ i ≤ n ∧ (f = i!)}
```

10. Finding a loop invariant is often tricky, as is the whole correctness proof process itself. Interested readers are encouraged to find additional sources (e.g., [Gries 1981]) that cover this very interesting topic in more detail.

```

while (i < n) {
  i = i + 1;
  f = f * i;
}
{i ≥ n ∧ 1 ≤ i ∧ i ≤ n ∧ (f = i!)}
{f = n!}
return f;

```

This effectively reduces the original problem of proving the correctness of the original program to the three problems: (1) proving the initialization part; (2) proving the premise of rule 4; and (3) proving the finalization part. These subproblems may be proved in any convenient order.

The third part seems easiest, since it involves the empty statement (or the skip statement in Jay). The proof can be achieved by repeated applications of rule 5:

$$i \geq n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset (i = n) \wedge (f = i!) \supset (f = n!)$$

Since $i \geq n$ and $i \leq n$, it follows that $i = n$. The second step involves a substitution of one variable for another.

A strategy for proving the initialization part is to use rule 2 to break a *Block*, or sequence of statements, into its individual components:

```

{1 ≤ n}
  f = 1;
{R'}
  i = 1;
{1 ≤ i ∧ i ≤ n ∧ (f = i!)}

```

The intermediate assertion R' can be found from the application of rule 1 by back substitution: $\{1 \leq 1 \wedge 1 \leq n \wedge (f = 1!)\}$. Again, applying rule 1 to R' on the program fragment:

```

{1 ≤ n}
  f = 1;
{1 ≤ 1 ∧ 1 ≤ n ∧ (f = 1!)}

```

we obtain: $\{1 \leq 1 \wedge 1 \leq n \wedge (1 = 1!)\}$, which simplifies to $1 \leq n$. Thus, we have shown:

$$\frac{\{1 \leq n\}f = 1; \{1 \leq 1 \wedge 1 \leq n \wedge f = 1!\} \quad \{1 \leq 1 \wedge 1 \leq n \wedge f = 1!\}i = 1; \{1 \leq i \wedge i \leq n \wedge f = i!\}}{\{1 \leq n\} f = 1; i = 1; \{1 \leq i \wedge i \leq n \wedge f = i!\}}$$

Thus, it only remains to show that P is a loop invariant; this means that we must show the premise of rule 4, namely, that the loop body preserves the truth of the loop invariant:

$$\frac{\{s.test \wedge P\}s.body\{P\}}{\{P\}s\{\neg s.test \wedge P\}}, \text{ where } s \text{ is a loop statement.}$$

Specifically, we must show it for the given loop test, invariant P , and loop body:

$$\begin{aligned} &\{i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!)\} \\ &\quad i = i + 1; \\ &\quad f = f * i; \\ &\{1 \leq i \wedge i \leq n \wedge (f = i!)\} \end{aligned}$$

Again we employ the strategy of using rule 2 for sequences to obtain:

$$\begin{aligned} &\{i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!)\} \\ &\quad i = i + 1; \\ &\{R'\} \\ &\quad f = f * i; \\ &\{1 \leq i \wedge i \leq n \wedge (f = i!)\} \end{aligned}$$

and then applying rule 1 on the assignment to f to find R' by back substitution: $1 \leq i \wedge i \leq n \wedge (f \times i = i!)$. We repeat this strategy using R' and rule 1 on the assignment to i to find: $1 \leq i \wedge i \leq n \wedge (f \times (i + 1) = (i + 1)!)$. If we can show that

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset 1 \leq i + 1 \wedge i + 1 \leq n \wedge (f \times (i + 1) = (i + 1)!),$$

then our proof is complete by rule 5. Using the following rule from logic:

$$\frac{p \supset q \quad p \supset r}{p \supset q \wedge r}$$

and rule 5, we shall prove each term of the consequent separately. First, we must show:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset 1 \leq i + 1,$$

This follows since $1 \leq i$ is a term in the antecedent. Next we must show:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset i + 1 \leq n,$$

Again, since $i < n$ is in the antecedent and since we are dealing with integers, it follows that $i + 1 \leq n$. Finally, we must show:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset (f \times (i + 1) = (i + 1)!),$$

Since $1 \leq i$, we can safely divide both sides of $f \times (i + 1) = (i + 1)!$ by $i + 1$, resulting in:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge (f = i!) \supset (f = i!),$$

which follows since the consequent appears as a term in the antecedent.

This concludes our proof of the (partial) correctness of the *Factorial* function given in Figure 3.2.

3.4.3 Correctness of Fibonacci

Suppose we want to prove mathematically that the C/C++ function *Fib* in Figure 3.3 actually computes as its result the n th Fibonacci number in the series 0, 1, 1, 2, 3, 5, 8, 13, 21, . . . , for any particular nonnegative value of n .

A strategy for proving the partial correctness of the rest of the program in Figure 3.3 works systematically from the postcondition backwards through the *if*, and then through the assignment statements in the *then* and *else* parts toward the given precondition.

| **Figure 3.3** A C/C++ Fibonacci Function

```

int Fib (int n) {
    int fib0 = 0;
    int fib1 = 1;
    int k = n;
    while (k > 0) {
        int temp = fib0;
        fib0 = fib1;
        fib1 = fib0 + temp;
        k = k - 1;
    }
    return fib0;
}

```

Rule 2 allows us to break a *Block*, or sequence of statements, into its individual constituents, and rules 3 and 4 allow us to reason through *Conditional* and *Loop* statements to derive their preconditions from their postconditions. As noted above, rule 5 allows us to cast off extraneous information along the way. Rule 5 is also useful when we want to introduce additional information in a predicate to anticipate what may be needed later in the process.

Now we can apply rule 1 two more times to derive a Hoare triple for each of the first three statements in the program:

$$\begin{aligned}
 &\{n \geq 0\} \\
 &\quad \text{fib0} = 0; \\
 &\{n \geq 0 \wedge \text{fib0} = 0\} \\
 &\{n \geq 0 \wedge \text{fib0} = 0\} \\
 &\quad \text{fib1} = 1; \\
 &\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\} \\
 &\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\} \\
 &\quad k = n; \\
 &\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1 \wedge 0 \leq k = n\}
 \end{aligned}$$

Now rule 2 gives us the ability to simplify this. For instance, if s_1 and s_2 are the first two statements in the above fragment, then:

$$\frac{\{n \geq 0\}s_1\{n \geq 0 \wedge \text{fib0} = 0\} \quad \{n \geq 0 \wedge \text{fib0} = 0\}s_2\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\}}{\{n \geq 0\}s_1 s_2\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\}}$$

allows us to rewrite the first two of these three Hoare triples as follows:

$$\begin{aligned}
 &\{n \geq 0\} \\
 &\quad \text{fib0} = 0; \\
 &\quad \text{fib1} = 1; \\
 &\{n \geq 0 \wedge \text{fib0} = 0 \wedge \text{fib1} = 1\}
 \end{aligned}$$

Using rule 2 again allows us to establish the validity of the following Hoare triple:

$$\begin{aligned}
 &\{n \geq 0\} \\
 &\quad \text{fib0} = 0;
 \end{aligned}$$

```

fib1 = 1;
k = n;
{n ≥ 0 ∧ fib0 = 0 ∧ fib1 = 1 ∧ 0 ≤ k = n}

```

Next we consider the while loop. To deal with it, we need to discover an assertion that is true just before every repetition of the loop, including the first. That is, we need to find the loop's *invariant*. Finding the loop's invariant requires anticipation of what should be true when the loop finally terminates. Here is an invariant for our while loop:

$$INV = \left\{ \begin{array}{l} 0 \leq k \leq n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \dots, n - k + 1\}: Fib(j) = Fib(j - 1) + Fib(j - 2) \wedge \\ fib0 = Fib(n - k) \wedge fib1 = Fib(n - k + 1) \end{array} \right\}$$

This says that for every value of k in which the loop has already run, we will have computed the Fibonacci numbers $Fib(0), Fib(1), \dots, Fib(n - k + 1)$, and the variables `fib0` and `fib1` are identical to the Fibonacci numbers $Fib(n - k)$ and $Fib(n - k + 1)$ respectively. In particular, just before the loop begins we have $k = n$ (i.e., the loop will have computed no numbers), the value of `fib0` is $Fib(0)$, or 0, and the value of `fib1` is $Fib(1)$, or 1.

Note that this invariant is implied by the postcondition we carefully derived for the first three statements in the program. Thus, proof rule 5 allows the invariant to be valid before the first iteration of the loop begins. Looking ahead, we anticipate that when $k = n - 1$ the loop will have just computed $Fib(2)$ and the value of `fib0` will become $Fib(1) = 1$. When $k = n - 2$, the loop will have computed $Fib(2)$ and $Fib(3)$, leaving `fib0` = $Fib(2)$, and so forth.

This process continues until $k > 0$ is no longer true (i.e., $k = 0$), and the invariant asserts that the loop will have computed $Fib(2), Fib(3), \dots$, and $Fib(n + 1)$, leaving the value of `fib0` = $Fib(n)$. Using the proof rule $\frac{\{s.test \wedge P\}s.body\{P\}}{\{P\}s\{\neg s.test \wedge P\}}$ in Table 3.1 for loop s , we relate the loop invariant with the precondition $s.test \wedge P$. Thus, when the loop terminates the test condition $k > 0$ is no longer *true*, which forces the following Hoare triple to be valid:

$$\left\{ \begin{array}{l} Fib(0) = 0 \wedge Fib(1) = 1 \wedge k = 0 \wedge \\ \forall j \in \{2, \dots, n - 0 + 1\}: Fib(j) = Fib(j - 1) + Fib(j - 2) \wedge \\ fib0 = Fib(n - 0) \wedge fib1 = Fib(n - 0 + 1) \end{array} \right\}$$

This logically implies the program's postcondition, using proof rule 5 (the rule of consequence) along with some fairly obvious simplifications:

$$\{Fib(0) = 0 \wedge Fib(1) = 1 \wedge \forall j \in \{2, \dots, n\}: Fib(j) = Fib(j - 1) + Fib(j - 2) \wedge fib0 = Fib(n)\}$$

A final task is to show that the invariant INV does remain valid for each repetition of the statements inside the body of the loop. That is, we need to show that the following (ugly!) Hoare triple is valid for every value of $k > 0$:

$$INV = \left\{ \begin{array}{l} 0 \leq k - 1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \dots, n - (k - 1)\}: Fib(j) = Fib(j - 1) + Fib(j - 2) \wedge \\ fib0 = Fib(n - k) \wedge fib1 = Fib(n - (k - 1)) \end{array} \right\}$$

```
int temp = fib0;
fib0 = fib1;
fib1 = fib0 + temp;
k = k - 1;
```

$$INV' = \left\{ \begin{array}{l} 0 \leq k < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \dots, n - k\}: Fib(j) = Fib(j - 1) + Fib(j - 2) \wedge \\ fib0 = Fib(n - k) \wedge fib1 = Fib(n - (k - 1)) \end{array} \right\}$$

To accomplish this, we use proof rule 5 to simplify intermediate expressions, and substitute equivalent expressions using our knowledge of algebra. For instance, we use the facts that the expression $n - k + 1$ in the above invariant INV is equivalent to $n - (k - 1)$, and $k > 0$ in the invariant is equivalent to $k - 1 \geq 0$.

Looking at the intermediate statements inside a single iteration of the loop, we see that they maintain the validity of the invariant and make progress toward the final computation of $fib0 = Fib(n)$.

$$INV = \left\{ \begin{array}{l} 0 \leq k - 1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \dots, n - (k - 1)\}: Fib(j) = Fib(j - 1) + Fib(j - 2) \wedge \\ fib0 = Fib(n - k) \wedge fib1 = Fib(n - (k - 1)) \end{array} \right\}$$

```
int temp = fib0;
```

$$\left\{ \begin{array}{l} 0 \leq k - 1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \dots, n - (k - 1)\}: Fib(j) = Fib(j - 1) + Fib(j - 2) \wedge \\ \underline{fib0 = Fib(n - k) \wedge fib1 = Fib(n - (k - 1)) \wedge temp = Fib(n - k)} \end{array} \right\}$$

```
fib0 = fib1;
```

$$\left\{ \begin{array}{l} 0 \leq k - 1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \dots, n - (k - 1)\}: Fib(j) = Fib(j - 1) + Fib(j - 2) \wedge \\ \underline{fib0 = Fib(n - (k - 1)) \wedge fib1 = Fib(n - (k - 1)) \wedge temp = Fib(n - k)} \end{array} \right\}$$

```
fib1 = fib0 + temp;
```

$$\left\{ \begin{array}{l} 0 \leq k - 1 < n \wedge Fib(0) = 0 \wedge Fib(1) = 1 \wedge \\ \forall j \in \{2, \dots, n - (k - 1)\}: Fib(j) = Fib(j - 1) + Fib(j - 2) \wedge \\ \underline{fib0 = Fib(n - (k - 1)) \wedge fib1 = Fib(n - (k - 1)) + Fib(n - k) \wedge temp = Fib(n - k)} \end{array} \right\}$$

```
k = k - 1;
```

$$\left\{ \begin{array}{l} 0 \leq k < n \wedge \text{Fib}(0) = 0 \wedge \text{Fib}(1) = 1 \wedge \\ \forall j \in \{2, \dots, n - k\}: \text{Fib}(j) = \text{Fib}(j - 1) + \text{Fib}(j - 2) \wedge \\ \underline{\text{fib0} = \text{Fib}(n - k) \wedge \text{fib1} = \text{Fib}(n - (k - 1)) \wedge \text{temp} = \text{Fib}(n - k)} \end{array} \right\}$$

To clarify the above, we have underlined that part of the invariant that changes after each of these four statements is taken into account. The last line implies the transformation of the invariant to the following form; since `temp` is a variable to be reassigned at the beginning of the next loop iteration, it can be dropped from the final assertion. Thus, the invariant INV' at the end of a single iteration is transformed as follows:

$$INV' = \left\{ \begin{array}{l} 0 \leq k < n \wedge \text{Fib}(0) = 0 \wedge \text{Fib}(1) = 1 \wedge \\ \forall j \in \{2, \dots, n - k\}: \text{Fib}(j) = \text{Fib}(j - 1) + \text{Fib}(j - 2) \wedge \\ \text{fib0} = \text{Fib}(n - k) \wedge \text{fib1} = \text{Fib}(n - (k - 1)) \end{array} \right\}$$

The key here is to notice that the values of `fib0` and `fib1` are transformed so that they represent the next adjacent pair of Fibonacci numbers following the pair that they represented at the beginning of this sequence.

The indexing is a bit tricky, but readers should see that the three Fibonacci numbers that influence a single iteration are $\text{Fib}(n - k)$, $\text{Fib}(n - (k - 1))$, and $\text{Fib}(n - (k - 2))$, in ascending order. For instance, when $k = n$, this group of statements begins with $\text{fib0} = \text{Fib}(0)$ and $\text{fib1} = \text{Fib}(1)$. This group of statements ends with $k = n - 1$, $\text{fib0} = \text{Fib}(n - (n - 1)) = \text{Fib}(1)$, and $\text{fib1} = \text{Fib}(n - ((n - 1) - 1)) = \text{Fib}(2)$.

3.4.4 Perspective

Axiomatic semantics and the corresponding techniques for proving the correctness of imperative programs were developed in the late 1960s and early 1970s. At that time it was the expectation that by now most programs would routinely be proven correct. Clearly that has not happened.

Actually, the importance of correctness proofs in software design has been a subject of heated debate, especially in the early 1990s. Many software engineers reject the notion of formal proof [DeMillo 1979], arguing that it is too complex and time-consuming a process for most programmers to master. Instead they use elaborate testing methods to convince themselves that the software runs correctly most of the time.

The counter argument was made by Dijkstra [1972] who stated that testing could only prove the presence of bugs, never their absence. Consider a simple program that inputs two 32-bit integers, computes some function, and outputs a 32-bit integer. There are 2^{64} possible inputs (approximately 10^{20}), so that even if one could test and verify (!) 100 million test cases per second, complete testing would take approximately 10^5 years. And this is for one of the simplest programs one can imagine!

Is there a middle ground between complex and time-consuming proofs and totally inadequate testing? We believe so.

First, properties of programs other than correctness can be routinely proved. These include safety of programs where safety is a critical issue. Absence of deadlock in concurrent programs is also often formally proved.

Second, the methods in object-oriented programs (as we shall see in Chapter 7) are often quite small in size. Informal proofs of such methods are routinely possible, although not often practiced. One reason for this is that many programmers, largely ignorant of formal program correctness, are not able to precisely state the input-output assertions for the methods they write.

Third, programmers trained in program correctness can and do state input-output assertions for the methods they write using formal English (or other natural language); this leads to vastly improved documentation.

As an example where such formalism could (and should) have been used, consider Sun's javadoc documentation for the various `String` methods in JDK 1.1. The comment for the method:

```
public String substring(int beginIndex, int endIndex);
```

states: “Returns a new string that is a substring of this string.” How imprecise! How would an implementor carry out an informal proof given such a vague specification? What are the range of valid values for `beginIndex` and `endIndex`? Is the minimum valid value for `beginIndex` 0 or 1? A programmer interested in producing an informal proof of an implementation of `substring` would at least require a more formal description of this method; such a description is left as an exercise.

3.5 DENOTATIONAL SEMANTICS

The *denotational semantics* of a language defines the meanings of abstract language elements as a collection of environment- and state-transforming functions. The *environment* of a program is the set of objects and types that are active at each step during its execution. The *state* of a program is the set of all active objects and their current values. These state-transforming functions depend on the assumption of some primitive types and transformations.

While axiomatic semantics is valuable for clarifying the meaning of a program as an abstract text, operational (or denotational) semantics addresses the meaning of a program as an active object within a computational (or functional) environment. The use of denotational semantics for defining meaning has both advantages and disadvantages. An advantage is that we can use the functional denotations of program meaning as the basis for specifying an interpreter for the language. However, this advantage raises an additional issue. That is, strictly speaking, the use of Java to implement the functional definition of, say, a *Loop* requires that we define the semantics of Java *itself* before using it to implement the semantics of Jay. That is, there is some circularity in using the denotational model as a basis for defining the meaning of a language.

Nevertheless, denotational semantics is widely used, and it allows us to define the meaning of an abstract Jay program as a series of state transformations resulting from the application of a series of functions M . These functions individually define the meaning of every class of element that can occur in the program's abstract syntax tree—*Program*, *Block*, *Conditional*, *Loop*, *Assignment*, and so forth.

Let Σ represent the set of all program states σ . Then a meaning function M is a mapping from a particular member of a given abstract class and current state in Σ to a new

state in Σ . That is:¹¹

$$M: \text{Class} \times \Sigma \rightarrow \Sigma$$

For instance, the meaning of an abstract *Statement* can be expressed as a state-transforming function of the form.

$$M: \text{Statement} \times \Sigma \rightarrow \Sigma$$

These functions are necessarily *partial* functions, which means that they are not well-defined for all members of their domain $\text{Class} \times \Sigma$. That is, the abstract representations of certain program constructs in certain states do not have finite meaning representations, even though those constructs are syntactically valid. For instance, consider the following insidious, yet syntactically valid, C/C++ statements (assuming *i* is an `int` variable).

```
for (i=1; i>-1; i++)
    i--;
```

In this case, there is no reasonable definition for a final state, since the value of *i* alternates between 1 and 0 as the loop repeats itself endlessly.

Since the abstract syntax of a Jay program is a tree structure whose root is the abstract element *Program*, the meaning of a Jay program can be defined by a series of functions, the first of which defines the meaning of *Program*.

$$M: \text{Program} \rightarrow \Sigma$$

$$M(\text{Program } p) = M(p.\text{body}, \{\langle v_1, \text{undef} \rangle, \langle v_2, \text{undef} \rangle, \dots, \langle v_m, \text{undef} \rangle\})$$

The first line of this definition gives a prototype function *M* for the program element being defined (in this case, *Program*), while the second defines the meaning of a program, using the abstract syntax definitions of the direct constituents of *Program*. Recall (see Appendix B) that the abstract definition of *Program* has two parts; a *decpart* (a series of abstract *Declarations*) and a *body*, which is a *Block*. So this functional definition says that the meaning of a program is the meaning of the program's body with the initial state $\{\langle v_1, \text{undef} \rangle, \langle v_2, \text{undef} \rangle, \dots, \langle v_m, \text{undef} \rangle\}$. We can predict that, as these meaning functions unfold and are applied to the elements of a specific program, this initial state will change as variables are assigned and reassigned values.

This functional style for defining the meaning of a program is particularly straightforward to implement in Java. The following Java method implements the above definition of function *M*, assuming the abstract syntax for language Jay that is defined in Appendix B.

```
State M (Program p) {
    return M (p.body, initialState(p.decpart));
}
```

11. Some treatments of formal semantics define the meaning of a program as a series of functions in which the name (e.g., $M_{\text{Statement}}$) distinguishes it from the rest. In those treatments, $M_{\text{Statement}}: \Sigma \rightarrow \Sigma$ would be equivalent to our $M: \text{Statement} \times \Sigma \rightarrow \Sigma$. This variation is used in the Scheme implementation of Jay semantics in Chapter 8.

```

State initialState (Declarations d) {
    State sigma = new State();
    Value undef = new Value();
    for (int i = 0; i < d.size(); i++)
        sigma.put(((Declaration)(d.elementAt(i))).v, undef);
    return sigma;
}

```

Chapter 4 introduces and discusses the semantics of the remaining abstract program elements. In Section 3.6, we preview that discussion by illustrating the formal semantics of Jay assignment statements and expressions.

3.6 EXAMPLE: SEMANTICS OF JAY ASSIGNMENTS AND EXPRESSIONS

Expressions and assignments abound in imperative programs. Their semantics can be expressed functionally by using the notions of state and state-changing functions that were introduced in Section 3.5. To illustrate, suppose we have the following concrete Jay assignment statement.

$$z = x + 2*y;$$

This statement translates to an abstract *Assignment* whose representation is shown in Figure 3.4, using the methods described in Chapter 2 and the Jay abstract syntax in Appendix B. An abstract *Assignment*, therefore, has two fundamental parts: a source *Expression* and a target *Variable*.

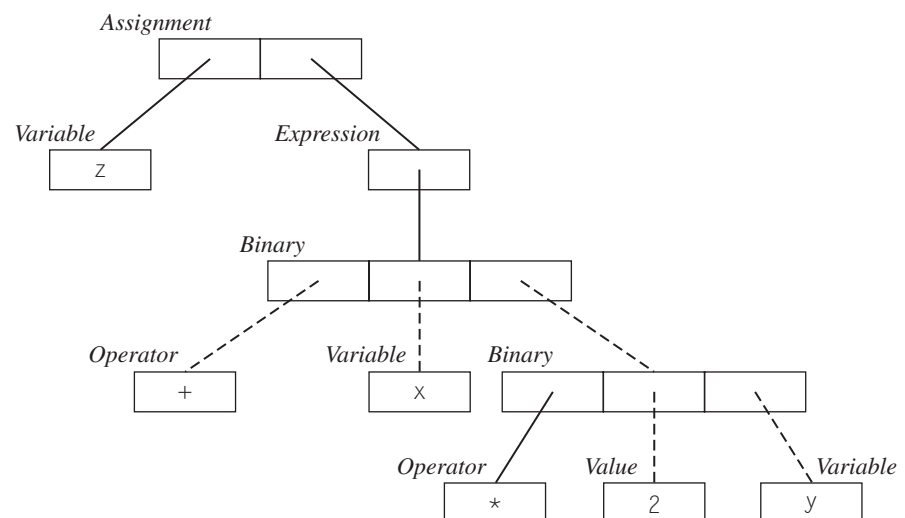


Figure 3.4 Abstract Syntax Sketch for the Jay Assignment $z = x + 2*y;$

3.6.1 Meaning of Assignments

The meaning of an *Assignment* can be defined by a state-transforming function of the following kind:

$$M: \text{Assignment} \times \Sigma \rightarrow \Sigma$$

$$M(\text{Assignment } a, \text{State } \sigma) = \sigma \bar{\cup} \{\langle a.\text{target}, M(a.\text{source}, \sigma) \rangle\}$$

What does this say? It is simply a formal way of describing the state that results from transforming the current state σ into a new state that differs from σ only by the pair whose first member is the *Variable* on the left of the *Assignment*. The new pair is constructed by combining that *Variable* with the meaning of the *Expression* on the right of the *Assignment*. The meaning of that *Expression* is (you guessed it!) defined by another function M , which will be explained in detail in Subsection 3.6.2.

All we need to know here is that the meaning of an expression is defined by a function M that returns a *Value*, since the second member of each pair in the state must be a *Value*. Let's return to our example in Figure 3.4, and let's assume that the current state σ is as follows:

$$\sigma = \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}$$

Intuitively, we expect that the meaning of the source expression in this state, or

$$M(a.\text{source}, \sigma) = M(x + 2*y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\})$$

will deliver the *Value* -4 , since we are assuming that M is fully defined for *Expressions*. Thus,

$$\begin{aligned} M(z = x + 2*y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}) &= \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\} \bar{\cup} \{\langle z, -4 \rangle\} \\ &= \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, -4 \rangle\} \end{aligned}$$

which completes the state transformation for this assignment.

3.6.2 Meaning of Arithmetic Expressions

In real programming languages, the meaning of an *arithmetic expression* must be carefully and elaborately defined, since expressions have several features that are not readily apparent when they are first read. For example, the innocuous expression $x + y/2$ may, in the context where x and y are declared as integers, describe a simple integer division followed by an addition. However, if either x or y is declared with type `float`, this expression conveys somewhat different meaning possibilities, since division may yield a `float` product, rather than an integer. Moreover, in some languages, x and y may denote vectors or matrices, which would attach an entirely different meaning to this expression. Chapter 4 considers these kinds of issues more carefully, and identifies the elements that must be added to the meaning function for expressions so that situations like these are well-defined.

In this section, we only consider the meaning of a Jay abstract *Expression*, so that we can make some fairly simple assumptions about the types of its arguments and result. That is, we assume that an *Expression* has only `int` arguments and only the four arithmetic operators ($+$, $-$, $*$, and $/$) that are defined in the concrete syntax for *Expression* discussed in Chapter 2. Thus, all individual arithmetic operations and results are of type

int. This discussion uses the following notational conventions:

- $v \in \sigma$ tests whether there is a pair whose *Identifier* is v in the current state σ .
- $\sigma(v)$ is a function that extracts from σ the value in the pair whose *Identifier* is v .

To facilitate the definition of meaning for *Expression*, the auxiliary function *ApplyBinary* is first defined. This function just takes two integer values and calculates an integer result. Here is its definition for the arithmetic operators; note the definition of the operator $/$ is a complicated, mathematical way of specifying the integer quotient of $v1$ divided by $v2$:

$$\begin{aligned}
 & \text{ApplyBinary} : \text{Operator} \times \text{Value} \times \text{Value} \rightarrow \text{Value} \\
 & \text{ApplyBinary}(\text{Operator } op, \text{Value } v1, \text{Value } v2) \\
 & \quad = v1 + v2 \quad \text{if } op = + \\
 & \quad = v1 - v2 \quad \text{if } op = - \\
 & \quad = v1 \times v2 \quad \text{if } op = * \\
 & \quad = \text{floor}\left(\left\lfloor \frac{v1}{v2} \right\rfloor\right) \times \text{sign}(v1 \times v2) \quad \text{if } op = /
 \end{aligned}$$

Using the abstract syntax for *Expression* given in Appendix B, but only insofar as it applies to expressions that have *Integer* operands, we can define the meaning of an *Expression* as follows:

$$\begin{aligned}
 & M : \text{Expression} \times \text{State} \rightarrow \text{Value} \\
 & M(\text{Expression } e, \text{State } \sigma) \\
 & \quad = e \quad \text{if } e \text{ is a } \textit{Value} \\
 & \quad = \sigma(e) \quad \text{if } e \text{ is a } \textit{Variable} \\
 & \quad = \text{ApplyBinary}(e.op, M(e.term1, \sigma), M(e.term2, \sigma)) \quad \text{if } e \text{ is a } \textit{Binary}
 \end{aligned}$$

This function defines the meaning of an expression in a particular state by cases. The first case extracts the value if the expression itself is a *Value*. The second case extracts the value of a variable from the state if the expression is a *Variable*. The third case computes a value by applying the appropriate binary operator to the values of the two terms that accompany the operator, in the case that the expression itself is a binary.

To illustrate, let's return to our familiar example, the expression $x+2*y$, and assume that it is being evaluated in state $\sigma = \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}$. That is, we want to use these functional definitions to show that $M(x+2*y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}) = -4$. Recall the abstract representation of the expression $x+2*y$ in Figure 3.4. Since this expression is a *Binary*, we have:

$$\begin{aligned}
 M(x+2*y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}) &= \text{ApplyBinary}(+, A, B) \\
 &\quad \text{where } A = M(x, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}) \\
 &\quad \text{and } B = M(2*y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\})
 \end{aligned}$$

Now the meaning of A is the value of x in state $\sigma = \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}$, or 2, since x is a *Variable*. The meaning of the *Binary* B , however, comes from another application of the function M , which is:

$$\begin{aligned}
 M(2*y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}) &= \text{ApplyBinary}(*, C, D) \\
 &\quad \text{where } C = M(2, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}) \\
 &\quad \text{and } D = M(y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\})
 \end{aligned}$$

The meaning of C is 2, since 2 is a *Value*. The meaning of D is -3 , since y is a *Variable* in state $\sigma = \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}$. With this information, the definition of *ApplyBinary* gives us the meaning of $2*y$:

$$\begin{aligned} M(2*y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}) &= \text{ApplyBinary}(*, 2, -3) \\ &= -6 \end{aligned}$$

Thus, the meaning of our original expression unravels as follows:

$$\begin{aligned} M(x+2*y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}) &= \text{ApplyBinary}(+, 2, -6) \\ &= -4 \end{aligned}$$

Note that this example used only the definitions of the functions *ApplyBinary* and *M* defined above, along with the mathematical properties of integer arithmetic to derive this result. It should be clear to readers that the meanings of more complex abstract *Expressions* with binary operations and integer operands are fully defined by these functions.

Several more examples of meaning definition for other Jay abstract program elements will be discussed and illustrated in Chapter 4.

3.6.3 Implementing the Semantic Functions

Implementing the denotational semantics of a programming language provides a ready test bed for experimentation with different semantic models. Implementing the semantics of Jay in Java requires defining a Java class `Semantics` which contains a method for each of the various functions *M* and auxiliary functions (like *ApplyBinary*) that together define the meaning of a program. The methods in this class refer to objects that are defined by the abstract syntax. The abstract syntax thus serves as a bridge between the concrete syntax of a program and its meaning. A complete definition of the Java class `Semantics` is given at the book's website given in the Preface.

A complete collection of semantic methods (that is, one that defines the semantics of *all* the constructs in the abstract syntax) defines, in effect, an interpreter for the language. Such an interpreter can be exercised to test the validity of the semantic definitions, as well as the trade-offs that occur among alternative semantic definitions. In defining a Java interpreter, we recall the limitations of the denotational approach mentioned at the beginning of this chapter. That is, a Java interpreter for Jay, to be complete, relies on the assumption that Java itself has been formally defined. In fact, that is nearly the case—a formal definition of Java has been carefully considered in recent research papers. Interested readers are referred to [Alves-Foss 1999] for more information.

With this strong assumption, we overview the definition of the `Semantics` class by considering the implementations of semantic functions *M* for *Assignment* and *Expression* that were introduced in Subsections 3.6.1–3.6.2. Chapter 4 will continue this development so that a complete interpreter for the small language Jay can be fully constructed.

Since it is a set of unique key-value pairs, the state of a computation is naturally implemented as a Java `Hashtable`, as shown in Figure 3.5. This means that the variable `sigma`, representing σ , can be defined and initialized as follows:

```
State sigma = new State();
```

```

class State extends Hashtable {
// State is implemented as a Java Hashtable

public State( ) { }

public State(Variable key, Value val) { put(key, val); }

public State union (State t) {
    for (Enumeration e = t.keys(); e.hasMoreElements(); ) {
        Variable key = (Variable)e.nextElement();
        put(key, t.get(key));
    }
    return this;
}
}

```

Figure 3.5 Java Implementation of the State Class

The functional expression $\sigma(v)$, which extracts from σ the value of variable v , can be implemented in Java as:

```
sigma.get(v)
```

Finally, the overriding union operation, $\sigma \bar{\cup} \{(v, val)\}$, which either replaces or adds a new variable v and its value val to the state, is implemented as the method `union`, which contains code to insert every pair in the new state t into the `Hashtable`, either replacing a pair with a matching key or adding a pair with a new key.

This function provides a straightforward way of implementing the meaning M of an *Assignment* statement from its definition:

$$M(\text{Assignment } a, \text{State } \sigma) = \sigma \bar{\cup} \{ \langle a.\text{target}, M(a.\text{source}, \sigma) \rangle \}$$

That is, the Java method in the `Semantics` class is implemented as follows:

```

State M (Assignment a, State sigma) {
    return sigma.union(new State(a.target, M (a.source, sigma)));
}

```

The meaning M of an arithmetic *Expression* with operators $+$, $-$, $*$, and $/$ follows directly from its definition as well. Since no state transformations occur here, all we are asking is that the Java method return a *Value* (rather than a *State*). Recalling that the meaning of an arithmetic expression is defined as:

$$\begin{aligned}
 M(\text{Expression } e, \text{State } \sigma) &= e.\text{val} && \text{if } e \text{ is a Value} \\
 &= \sigma(e) && \text{if } e \text{ is a Variable} \\
 &= \text{ApplyBinary}(e.\text{op}, M(e.\text{term1}, \sigma), M(e.\text{term2}, \sigma)) && \text{if } e \text{ is a Binary}
 \end{aligned}$$

the following Java implementation is suggested:

```

Value M (Expression e, State sigma) {
    if (e instanceof Value)
        return (Value)e;
    if (e instanceof Variable)
        return (Value)(sigma.get((Variable)e));
    if (e instanceof Binary)
        return applyBinary (((Binary)e).op,
                             M(((Binary)e).term1, sigma),
                             M(((Binary)e).term2, sigma));
    return null;
}

```

Considering the function *ApplyBinary*, which is defined for *Expression* as:

$$\begin{aligned}
 \text{ApplyBinary}(\text{Operator } op, \text{Value } v1, \text{Value } v2) & \\
 = v1 + v2 & \quad \text{if } op = + \\
 = v1 - v2 & \quad \text{if } op = - \\
 = v1 \times v2 & \quad \text{if } op = * \\
 = \text{floor}\left(\frac{v1}{v2}\right) \times \text{sign}(v1 \times v2) & \quad \text{if } op = /
 \end{aligned}$$

we have the following direct Java encoding (note that the Java code for integer division is simpler than the mathematical definition).

```

Value applyBinary (Operator op, Value v1, Value v2) {
    if (v1.type.isUndefined() || v2.type.isUndefined())
        return new Value();
    if (op.ArithmeticOp( )) {
        if (op.val.equals(Operator.PLUS))
            return new Value(v1.intValue + v2.intValue);
        if (op.val.equals(Operator.MINUS))
            return new Value(v1.intValue - v2.intValue);
        if (op.val.equals(Operator.TIMES))
            return new Value(v1.intValue * v2.intValue);
        if (op.val.equals(Operator.DIV))
            return new Value(v1.intValue / v2.intValue);
    }
    return null;
}

```

The message in these illustrations is clear. Implementation of these meaning functions *M* in Java is relatively painless and straightforward, once we have settled on a representation scheme for the notion of *State*. The Java `Hashtable` provides that representation directly.

EXERCISES

- 3.1** Expand the static type checking function V for *Declarations* so that it defines the requirement that the type of each variable be taken from a small set of available types, say $\{\text{int}, \text{boolean}\}$. Use the same functional style and abstract syntax for *Declarations* that are discussed in this chapter.
- 3.2** Expand the Java method that implements the function V for *Declarations* so that it implements the additional requirement stated in Question 3.1.
- 3.3** Argue that the Java method that implements the function V for *Declarations* is correct, in the sense that it covers all the cases that the function itself covers.
- 3.4** Suppose $\sigma_1 = \{\langle x, 1 \rangle, \langle y, 2 \rangle, \langle z, 3 \rangle\}$, $\sigma_2 = \{\langle y, 5 \rangle\}$, and $\sigma_3 = \{\langle w, 1 \rangle\}$. What are the results of the following operations?
- (a) $\sigma_1 \bar{\cup} \sigma_2$
 - (b) $\sigma_1 \bar{\cup} \sigma_3$
 - (c) $\sigma_2 \bar{\cup} \sigma_3$
 - (d) $\emptyset \bar{\cup} \sigma_2$
 - (e) $\sigma_1 \otimes \sigma_3$
 - (f) $\sigma_2 \otimes \sigma_3$
 - (g) $(\sigma_1 - (\sigma_1 \otimes \sigma_3)) \cup \sigma_3$
- 3.5** Complete the operational semantics for the arithmetic, boolean, and logical operators of Jay by writing an execution rule for each operator.
- 3.6** Derive the complete operational semantics for the following program segment, showing all execution rule applications and deriving the final state that includes the result $\langle \text{fib0}, 3 \rangle$. Assume that the initial state $\sigma = \emptyset$.

```

int fib0 = 0;
int fib1 = 1;
int k = 4;
while (k > 0) {
    int temp = fib0;
    fib0 = fib1;
    fib1 = fib0 + temp;
    k = k - 1;
}

```

- 3.7** Below is a Hoare triple that includes a Jay program segment to compute the product z of two integers x and y .

```

{y ≥ 0}
z = 0;
n = y;
while (n > 0) {
    z = z + x;
    n = n - 1;
}
{z = xy}

```


- 3.15** (a) How does Java define the numerical idea of infinity? (You should look at the *Java Language Specification* [Gosling 1996] for the details.)
- (b) Looking at the specifications in the *Java Language Definition*, can you explain in plain English the meaning of the statement $i = 3 / j$; for all possible values of j , including 0?
- (c) (Optional) Can you write a functional definition for the meaning of division in Java using these ideas? Start with the prototype function $M : Division \times \Sigma \rightarrow \Sigma$.
- 3.16** Consider the expression $x + y/2$ in the language C. How many different interpretations does this expression have, depending on the types of x and y . Can you find a language in which this expression can denote vector or matrix arithmetic, when x and y themselves denote vectors or matrices?
- 3.17** Show how the meaning of each of the following expressions and given states are derived from the functions M and $ApplyBinary$ given in this chapter. (You developed the abstract syntax for each of these expression as an exercise in Chapter 2.)
- (a) $M(z+2)*y, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\}$
- (b) $M(2*x+3/y-4, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\})$
- (c) $M(1, \{\langle x, 2 \rangle, \langle y, -3 \rangle, \langle z, 75 \rangle\})$
- 3.18** Show all steps in the derivation of the meaning of the following assignment statement when executed in the given state, using this chapter's definitions of the functions M and $ApplyBinary$.
- $M(z=2*x+3/y-4, \{\langle x, 6 \rangle, \langle y, -12 \rangle, \langle z, 75 \rangle\})$

