
Chapter 11

AXIOMATIC SEMANTICS

The techniques for operational semantics, introduced in Chapters 5 through 8, and denotational semantics, discussed in Chapters 9 and 10, are based on the notion of the “state of a machine”. For example, in the denotational semantics of Wren, the semantic equation for the execution of a statement is a mapping from the current machine state, represented by the store, input stream and output stream, to a new machine state.

Based on methods of logical deduction from predicate logic, axiomatic semantics is more abstract than denotational semantics in that there is no concept corresponding to the state of the machine. Rather, the semantic meaning of a program is based on assertions about relationships that remain the same each time the program executes. The relation between an initial assertion and a final assertion following a piece of code captures the essence of the semantics of the code. Another piece of code that defines the algorithm slightly differently yet produces the same final assertion will be semantically equivalent provided any initial assertions are also the same. The proofs that the assertions are true do not rely on any particular architecture for the underlying machine; rather they depend on the relationships between the values of the variables. Although individual values of variables change as a program executes, certain relationships among them remain the same. These invariant relationships form the assertions that express the semantics of the program.

11.1 CONCEPTS AND EXAMPLES

Axiomatic semantics has two starting points: a paper by Robert Floyd and a somewhat different approach introduced by C. A. R. Hoare. We use the notation presented by Hoare. Axiomatic semantics is commonly associated with proving a program to be correct using a purely static analysis of the text of the program. This static approach is in clear contrast to the dynamic approach, which tests a program by focusing on how the values of variables change as a program executes. Another application of axiomatic semantics is to consider assertions as program specifications from which the program code itself can be derived. We look at this technique briefly in section 11.5.

Axiomatic semantics does have some limitations: Side effects are disallowed in expressions; the **goto** command is difficult to specify; aliasing is not allowed; and scope rules are difficult to describe unless we require all identifier names to be unique. Despite these limitations, axiomatic semantics is an attractive technique because of its potential effect on software development:

- The development of “bug free” algorithms that have been proved correct.
- The automatic generation of program code based on specifications.

Axiomatic Semantics of Programming Languages

In proving the correctness of a program, we use an applied predicate (first-order) logic with equality whose individual variables correspond to program variables and whose function symbols include all the operations that occur in program expressions. Therefore we view expressions such as “ $2 * n + 1$ ” and “ $x + y > 0$ ” as terms in the predicate logic (mathematical terms) whose values are determined by the current assignment to the individual variables in the logic language. Furthermore, we assume the standard mathematical and logical properties of operations modeled in the logic—for example, $2 * 3 + 1 = 7$ and $4 + 1 > 0 = \text{true}$.

An **assertion** is a logical formula constructed using the individual variables, individual constants, and function symbols in the applied predicate calculus. When each variable in an assertion is assigned a value (determined by the value of the corresponding program variable), the assertion becomes valid (true) or invalid (false) under a standard interpretation of the constants and functions in the logical language.

Typically, assertions consist of a conjunction of elementary statements describing the logical properties of program variables, such as stating that a variable takes values from a particular set, say $m < 5$, or defining a relation among variables, such as $k = n^2$. In many cases, assertions correspond directly to Boolean expressions in Wren, and the two notions are frequently confused in axiomatic semantics. We maintain a distinction between assertions and Boolean expressions by always presenting assertions in an italic font *like this*. In some instances, assertions use features of predicate logic that go beyond what is expressible in Boolean expressions—namely, when universal quantifiers, existential quantifiers, and implications occur in formulas.

For the purposes of axiomatic semantics, a program reduces to the meaning of a command, which in the abstract syntax includes a sequence of commands. We describe the semantics of a program by annotating it with assertions that are always valid when the control of the program reaches the points of the assertions. In particular, the meaning or correctness of a command (a

program) is described by placing an assertion, called a **precondition**, before a command and another assertion, called a **postcondition**, after the command:

$$\{PRE\} C \{POST\}.$$

Therefore the meaning of command C can be viewed as the ordered pair $\langle PRE, POST \rangle$, called a specification of C . We say that the command C is **correct with respect to the specification** given by the precondition and postcondition provided that if the command is executed with values that make the precondition true, the command halts and the resulting values make the postcondition true. Extending this notion to an entire program supplies a meaning of program correctness and a semantics to programs in a language.

Definition: A program is **partially correct** with respect to a precondition and a postcondition provided that if the program is started with values that make the precondition true, the resulting values make the postcondition true when the program halts (if ever). If it can also be shown that the program terminates when started with values satisfying the precondition, the program is called **(totally) correct**.

Partial Correctness = (Precondition and Termination \supset Postcondition)

Total Correctness = (Partial Correctness and Termination). ■

We focus on proofs of partial correctness for programs in Wren and Pelican in the next two sections and briefly look at proofs of termination in section 11.4. The goal of axiomatic semantics is to provide axioms and proof rules that capture the intended meaning of each command in a programming language. These rules are constructed so that a specification for a given command can be deduced, thereby proving the partial correctness of the command relative to the specification. Such a deduction consists of a finite sequence of assertions (formulas of the predicate logic) each of which is either the precondition, an axiom associated with a program command, or a rule of inference whose premises have already been established.

Before considering the axioms and proof rules for Wren, we need to discuss the problem of specifications briefly. Extensive literature has dealt with the difficult problem of accurate specifications of algorithms. Programmers frequently miss the subtlety inherent in precise specifications. As an example, consider the following specification of the problem of finding the smaller of two nonnegative integers:

$$PRE = \{ m \geq 0 \text{ and } n \geq 0 \}$$

$$POST = \{ \text{minimum} \leq m \text{ and } \text{minimum} \leq n \text{ and } \text{minimum} \geq 0 \}.$$

Unhappily, this specification is satisfied by the command “`minimum := 0`”, which does not satisfy the informal description. We do not have space in this

text to consider the problems of accurate specifications, but correctness proofs of programs only serve the programmer when the proof is carried out relative to correct specifications.

11.2 AXIOMATIC SEMANTICS FOR WREN

Again Wren serves as the initial programming language for semantic specification. In the next section we expand the presentation to Pelican with constants, procedures, blocks, and recursion. For each of these languages, axiomatic semantics focuses on assertions that describe the logical relationships between the values of program variables at points in a program.

An axiomatic analysis of Wren program behavior concentrates on the commands of the programming language. In the absence of side effects, expressions in Wren can be treated as mathematical expressions and be evaluated using mathematical rules. We assume that any program submitted for semantic analysis has already been verified as syntactically correct, including adherence to all context conditions. Therefore the declarations (of variables only) in Wren can be ignored in describing its axiomatic semantics. In the next section we investigate the impact of constant and procedure declarations on this approach to semantics.

Assignment Command

The first command we examine is assignment, beginning with three examples of preconditions and postconditions for assignment commands:

Example 1: $\{ k = 5 \} k := k + 1 \{ k = 6 \}$

Example 2: $\{ j = 3 \text{ and } k = 4 \} j := j + k \{ j = 7 \text{ and } k = 4 \}$

Example 3: $\{ a > 0 \} a := a - 1 \{ a \geq 0 \}$.

For these simple examples, correctness is easy to prove either proceeding from the precondition to the postcondition or from the postcondition to the precondition. However, many times starting with the postcondition and working backward to derive the precondition proves easier (at least initially). We assume expressions with no side effects in the assignment commands, so only the the target variable is changed. “Working backward” means substituting the expression on the right-hand side of the assignment for every occurrence of the target variable in the postcondition and deriving the precondition, following the principle that whatever is true about the target variable after the assignment must be true about the expression before the assignment. Consider the following examples:

Example 1

$\{ k = 6 \}$ postcondition
 $\{ k + 1 = 6 \}$ substituting $k + 1$ for k in postcondition
 $\{ k = 5 \}$ precondition, after simplification.

Example 2

$\{ j = 7 \text{ and } k = 4 \}$ postcondition
 $\{ j + k = 7 \text{ and } k = 4 \}$ substituting $j + k$ for j in postcondition
 $\{ j = 3 \text{ and } k = 4 \}$ precondition, after simplification.

Example 3

$\{ a \geq 0 \}$ postcondition
 $\{ a - 1 \geq 0 \}$ substituting $a - 1$ for a in postcondition
 $\{ a \geq 1 \}$ simplification
 $\{ a > 0 \}$ precondition, since $a \geq 1 \equiv a > 0$ assuming a is an integer.

Given an assignment of the form $V := E$ and a postcondition P , we use the notation $P[V \rightarrow E]$ (as in Chapter 5) to indicate the consistent substitution of E in place of each free occurrence of V in P . This notation enables us to give an axiomatic definition for the assignment command as

$$\{ P[V \rightarrow E] \} V := E \{ P \} \quad (\text{Assign})$$

The substitution operation $P[V \rightarrow E]$ needs to be defined carefully since formulas in the predicate calculus allow both free and bound occurrences of variables. This task will be given as an exercise at the end of this section.

If we view assertions as predicates—namely, Boolean valued expressions with a parameter—the axiom can be stated

$$\{ P(E) \} V := E \{ P(V) \}.$$

A proof of correctness following the assignment axiom can be summarized by writing

$$\begin{aligned}
 \{ a > 0 \} &\supset \\
 \{ a \geq 1 \} &\supset \\
 \{ a - 1 \geq 0 \} &= \{ P(a-1) \} \\
 a &:= a-1 \\
 \{ a \geq 0 \} &= \{ P(a) \}
 \end{aligned}$$

where \supset denotes logical implication. The axiom that specifies “ $V := E$ ” essentially states that if we can prove a property about E before the assignment, the same property about V holds after the assignment.

At first glance the assignment axiom may seem more complicated than it needs to be with its use of substitution in the precondition. To appreciate the subtlety of assignment, consider the following unsound axiom:

$$\{ true \} V := E \{ V = E \}.$$

This apparently reasonable axiom for assignment is unsound because it allows us to prove false assertions—for example,

$$\{ true \} m := m+1 \{ m = m+1 \}.$$

Input and Output

The commands **read** and **write** assume the existence of input and output files. We use “IN = ” and “OUT = ” to indicate the contents of these files in assertions and brackets to represent a list of items in a file; so [1,2,3] represents a file with the three integers 1, 2 and 3. We consider the left side of the list to be the start of the file and the right side to be the end. For example, affixing the value 4 onto the end of the file [1,2,3] is represented by writing [1,2,3][4]. In a similar way, 4 is prefixed to the file [1,2,3] by writing [4][1,2,3]. Juxtaposition means concatenation.

Capital letters are used to indicate some unspecified item or sequence of items; [κ]L thus represents a file starting with the value κ and followed by any sequence L, which may or may not be empty. For contrast, small caps denote numerals, and large caps denote lists of numerals. Exploiting this notation, we specify the semantics of the **read** command as removing the first value from the input file and “assigning” it to the variable that appears in the command.

$$\{ IN = [κ]L \text{ and } P[V \rightarrow κ] \} \text{ read } V \{ IN = L \text{ and } P \} \quad (\text{Read})$$

The **write** command appends the current value denoted by the expression to the end of the output file. Our axiomatic rule also specifies that the value of the expression is not changed and that no other assertions are changed.

$$\{ OUT=L \text{ and } E=κ \text{ and } P \} \text{ write } E \{ OUT=L[κ] \text{ and } E=κ \text{ and } P \} \quad (\text{Write})$$

where P is any arbitrary set of additional assertions.

The symbols acting as variables in the Read axiom serve two different purposes. Those symbols that describe the input list, a numeral and a list of numerals, stay constant throughout any deduction containing them. We refer to symbols of this type as logical variables, meaning that their bindings are frozen during the deduction (see Appendix A for a discussion of logical variables in Prolog). In contrast, the variable V and any variables in the expression E correspond to program variables and may represent different values during a verification. The values of these variables depend on the current

assignment of values to program variables at the point where the assertion containing them occurs. When applying the axioms and proof rules of axiomatic semantics, we will use uppercase letters for logical variables and lowercase letters for individual variables corresponding to program variables.

The axioms and rules of inference in an axiomatic definition of a programming language are really axiom and rule schemes. The symbols “V”, “E”, and “P” need to be replaced by actual variables, expressions, and formulas, respectively, to form instances of the axioms and rules for use in a deduction.

Rules of Inference

For other axiomatic specifications, we introduce rules of inference that have the form

$$\frac{H_1, H_2, \dots, H_n}{H}$$

This notation can be interpreted as

If H_1, H_2, \dots, H_n have all been verified, we may conclude that H is valid.

Note the similarity with the notation used by structural operational semantics in Chapter 8. The sequencing of two commands serves as the first example of a rule of inference:

$$\frac{\{P\} C_1 \{Q\}, \{Q\} C_2 \{R\}}{\{P\} C_1; C_2 \{R\}} \quad \text{(Sequence)}$$

This rule says that if starting with the precondition P we can prove Q after executing C_1 and starting with Q we can prove R after C_2 , we can conclude that starting with the precondition P , R is true after executing $C_1; C_2$. Observe that the middle assertion Q is “forgotten” in the conclusion.

The **if** command involves a choice between alternatives. Two paths lead through an **if** command; therefore, if we can prove each path is correct given the appropriate value of the Boolean expression, the entire command is correct.

$$\frac{\{P \text{ and } B\} C_1 \{Q\}, \{P \text{ and } (\text{not } B)\} C_2 \{Q\}}{\{P\} \text{ if } B \text{ then } C_1 \text{ else } C_2 \text{ end if } \{Q\}} \quad \text{(If-Else)}$$

Note that the Boolean expression B is used as part of the assertions in the premises of the rule. The axiomatic definition for the single alternative **if** is similar, except that for the false branch we need to show that the final assertion can be derived directly from the initial assertion P when the condition B is false.

$$\frac{\{P \text{ and } B\} C \{Q\}, (P \text{ and } (\text{not } B)) \supset Q}{\{P\} \text{ if } B \text{ then } C \text{ end if } \{Q\}} \quad (\text{If-Then})$$

Before presenting the axiomatic definition for **while**, we examine some general rules applicable to all commands and verify a short program. Sometimes the result that is proved is stronger than required. In this case it is possible to weaken the postcondition.

$$\frac{\{P\} C \{Q\}, Q \supset R}{\{P\} C \{R\}} \quad (\text{Weaken})$$

Other times the given precondition is stronger than necessary to complete the proof.

$$\frac{P \supset Q, \{Q\} C \{R\}}{\{P\} C \{R\}} \quad (\text{Strengthen})$$

Finally, it is possible to relate assertions by the logical relationships *and* and *or*.

$$\frac{\{P_1\} C \{Q_1\}, \{P_2\} C \{Q_2\}}{\{P_1 \text{ and } P_2\} C \{Q_1 \text{ and } Q_2\}} \quad (\text{And})$$

$$\frac{\{P_1\} C \{Q_1\}, \{P_2\} C \{Q_2\}}{\{P_1 \text{ or } P_2\} C \{Q_1 \text{ or } Q_2\}} \quad (\text{Or})$$

Example: For a first example of a proof of correctness, consider the following program fragment.

```

read x; read y;
if x < y then write x
    else write y
end if

```

To avoid the runtime error of reading from an empty file, the initial assertion requires two or more items in the input file, which we indicate by writing two items in brackets before the rest of the file. The output file may or may not be empty initially.

Precondition: $P = \{IN = [M, N]L_1 \text{ and } OUT = L_2\}$

The program writes to the output file the minimum of the two values read. To specify this as an assertion, we consider two alternatives:

Postcondition: $Q = \{(OUT = L_2[M] \text{ and } M < N) \text{ or } (OUT = L_2[N] \text{ and } M \geq N)\}$

The correct assertion after the first **read** command is

$$R = \{IN = [N]L_1 \text{ and } OUT = L_2 \text{ and } x = M\},$$

and after the second **read** command the correct assertion is

$$S = \{ IN = L_1 \text{ and } OUT = L_2 \text{ and } x = M \text{ and } y = N \}.$$

We obtain these assertions by working the axiom for the **read** command backward through the first two commands. The verification of these assertions can then be presented in a top-down manner as follows:

$$\begin{aligned} & \{ IN = [M,N]L_1 \text{ and } OUT = L_2 \} \supset \\ & \{ IN = [M,N]L_1 \text{ and } OUT = L_2 \text{ and } M = M \} = P' \\ & \quad \mathbf{read\ x;} \\ & \{ IN = [N]L_1 \text{ and } OUT = L_2 \text{ and } x = M \} \supset \\ & \{ IN = [N]L_1 \text{ and } OUT = L_2 \text{ and } x = M \text{ and } N = N \} = R' \\ & \quad \mathbf{read\ y;} \\ & \{ IN = L_1 \text{ and } OUT = L_2 \text{ and } x = M \text{ and } y = N \} = S. \end{aligned}$$

Since $\{ x < y \text{ or } x \geq y \}$ is always true, we can add it to our assertion without changing its truth value. After manipulating this assertion using the logical equivalence (using the symbol \equiv for equivalence),

$$(P_1 \text{ and } (P_2 \text{ or } P_3)) \equiv ((P_1 \text{ and } P_2) \text{ or } (P_1 \text{ and } P_3)),$$

we have the assertion:

$$\begin{aligned} S' &= \{ (IN = L_1 \text{ and } OUT = L_2 \text{ and } x = M \text{ and } y = N \text{ and } (x < y \text{ or } x \geq y)) \} \equiv \\ & \{ (IN = L_1 \text{ and } OUT = L_2 \text{ and } x = M \text{ and } y = N \text{ and } x < y) \text{ or } \\ & \quad (IN = L_1 \text{ and } OUT = L_2 \text{ and } x = M \text{ and } y = N \text{ and } x \geq y) \}. \end{aligned}$$

Representing this assertion as $\{ P_1 \text{ or } P_2 \}$, we now must prove the validity of

$$\begin{aligned} & \{ P_1 \text{ or } P_2 \} \\ & \quad \mathbf{if\ } x < y \text{ \textbf{then write } } x \\ & \quad \quad \mathbf{else write } y \\ & \quad \mathbf{end\ if} \\ & \{ Q \} \end{aligned}$$

where Q is $\{ (OUT = L_2[M] \text{ and } M < N) \text{ or } (OUT = L_2[N] \text{ and } M \geq N) \}$. Therefore we must prove valid

$$\{ (P_1 \text{ or } P_2) \text{ and } B \} \mathbf{write\ } x \{ Q \}$$

$$\text{and } \{ (P_1 \text{ or } P_2) \text{ and } (\text{not } B) \} \mathbf{write\ } y \{ Q \}$$

where B is $\{ x < y \}$.

$\{ (P_1 \text{ or } P_2) \text{ and } B \}$ simplifies to

$$T_1 = \{ IN = L_1 \text{ and } OUT = L_2 \text{ and } x = M \text{ and } y = N \text{ and } x < y \}.$$

After executing "**write x**", we have

$$\{ IN = L_1 \text{ and } OUT = L_2[M] \text{ and } x = M \text{ and } y = N \text{ and } M < N \}.$$

Call this Q_1 . Similarly $\{(P_1 \text{ or } P_2) \text{ and } (\text{not } B)\}$ simplifies to

$$T_2 = \{IN = L_1 \text{ and } OUT = L_2 \text{ and } x = M \text{ and } y = N \text{ and } x \geq y\}.$$

After the “**write** y ” we have

$$\{IN = L_1 \text{ and } OUT = L_2[N] \text{ and } x = M \text{ and } y = N \text{ and } M \geq N\}.$$

Call this Q_2 . Since $Q_1 \supset (Q_1 \text{ or } Q_2)$ and $Q_2 \supset (Q_1 \text{ or } Q_2)$ we replace each individual assertion with

$$\begin{aligned} Q_1 \text{ or } Q_2 \equiv \\ ((IN = L_1 \text{ and } OUT = L_2[M] \text{ and } x = M \text{ and } y = N \text{ and } M < N) \text{ or} \\ (IN = L_1 \text{ and } OUT = L_2[N] \text{ and } x = M \text{ and } y = N \text{ and } M \geq N)). \end{aligned}$$

Finally we weaken the conclusion by removing the parts of the assertion about the input file and the values of x and y to arrive at our final assertion, the postcondition. Figure 11.1 displays the deduction as proof trees using the abbreviations given above. Note that we omit “**end if**” to save space. ■

$$\frac{\frac{P \supset P', \quad \{P\} \text{ read } x \{R\}}{\{P\} \text{ read } x \{R\}} \quad \frac{R \supset R', \quad \{R'\} \text{ read } y \{S\}, \quad S \supset S', \quad S' \supset (P_1 \text{ or } P_2)}{\{R\} \text{ read } y \{P_1 \text{ or } P_2\}}}{\{P\} \text{ read } x ; \text{ read } y \{P_1 \text{ or } P_2\}}$$

$$\frac{(((P_1 \text{ or } P_2) \text{ and } B) \supset T_1), \quad \{T_1\} \text{ write } x \{Q_1\}}{\frac{\{P_1 \text{ or } P_2\} \text{ and } B\} \text{ write } x \{Q_1\}, \quad Q_1 \supset (Q_1 \text{ or } Q_2)}{\{P_1 \text{ or } P_2\} \text{ and } B\} \text{ write } x \{Q_1 \text{ or } Q_2\}}}$$

$$\frac{(((P_1 \text{ or } P_2) \text{ and } (\text{not } B)) \supset T_2), \quad \{T_2\} \text{ write } y \{Q_2\}}{\frac{\{P_1 \text{ or } P_2\} \text{ and } (\text{not } B)\} \text{ write } y \{Q_2\}, \quad Q_2 \supset (Q_1 \text{ or } Q_2)}{\{P_1 \text{ or } P_2\} \text{ and } (\text{not } B)\} \text{ write } y \{Q_1 \text{ or } Q_2\}}}$$

$$\frac{\{P_1 \text{ or } P_2\} \text{ and } B\} \text{ write } x \{Q_1 \text{ or } Q_2\}, \quad \{P_1 \text{ or } P_2\} \text{ and } (\text{not } B)\} \text{ write } y \{Q_1 \text{ or } Q_2\}}{\frac{\{P_1 \text{ or } P_2\} \text{ if } x < y \text{ then write } x \text{ else write } y \{Q_1 \text{ or } Q_2\}, \quad (Q_1 \text{ or } Q_2) \supset Q}{\{P_1 \text{ or } P_2\} \text{ if } x < y \text{ then write } x \text{ else write } y \{Q\}}}$$

$$\frac{\{P\} \text{ read } x ; \text{ read } y \{P_1 \text{ or } P_2\}, \quad \{P_1 \text{ or } P_2\} \text{ if } x < y \text{ then write } x \text{ else write } y \{Q\}}{\{P\} \text{ read } x ; \text{ read } y ; \text{ if } x < y \text{ then write } x \text{ else write } y \{Q\}}$$

Figure 11.1: Derivation Tree for the Correctness Proof

While Command and Loop Invariants

Continuing the axiomatic definition of Wren, we specify the **while** command:

$$\frac{\{ P \text{ and } B \} C \{ P \}}{\{ P \} \text{ while } B \text{ do } C \text{ end while } \{ P \text{ and } (\text{not } B) \}} \quad (\text{While})$$

In this definition P is called the **loop invariant**. This assertion captures the essence of the **while** loop: It must be true initially, it must be preserved after the loop body executes, and, combined with the exit condition, it implies the assertion that follows the loop. Figure 11.2 illustrates the situation.

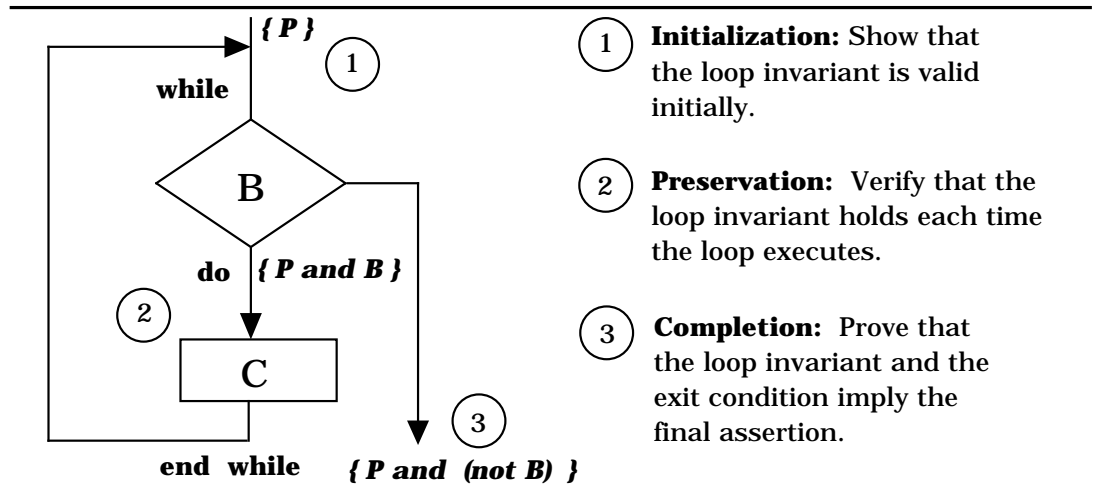


Figure 11.2: Structure of the While Rule

The purpose of the Preservation step is to verify the premise for the While rule shown above. The Initialization and Completion steps are used to tie the **while** loop into its surrounding code and assertions.

Example: Discovering the loop invariant requires insight. Consider the following program fragment that calculates factorial, as indicated by the final assertion. Remember, we use lowercase letters for variables and uppercase (small caps) to represent numerals that remain constant.

```

{ N ≥ 0 }
  k := N; f := 1;
  while k > 0 do      { loop invariant }
    f := f * k; k := k - 1;
  end while
{ f = N! }
    
```

The loop invariant involves a relationship between variables that remains the same no matter how many times the loop executes. The loop invariant also involves the **while** loop condition, $k > 0$ in the example above, modified to include the exit case, which is $k = 0$ in this case. Combining these conditions, we have $k \geq 0$ as part of the loop invariant. Other components of the loop invariant involve the variables that change values as a result of loop execution, f and k in the program above. We also look at the final assertion after the loop and notice that $N!$ needs to be involved. For this program, we can discover the loop invariant by examining how $N!$ is calculated for a simple case, say $N = 5$. We examine the calculation in progress at the end of the loop where k has just been decremented to 3.

$$N! = 5 \cdot 4 \cdot \overset{k}{\downarrow} 3 \cdot 2 \cdot 1$$

$\underbrace{\hspace{1.5cm}}_f \qquad \underbrace{\hspace{1.5cm}}_{k!}$

The variable f has stored the part of the computation completed so far, $5 \cdot 4$, and k has the starting value for the remaining computation. So $k!$ represents the rest of the value to be computed. The complete value is $f \cdot k!$, which, at all times, must equal $N!$. We can show this in a table:

k	k!	f	f•k!
5	120	1	120
4	24	5	120
3	6	20	120
2	2	60	120
1	1	120	120
0	1	120	120

Now we have our loop invariant: $\{ f \cdot k! = N! \text{ and } k \geq 0 \}$.

We show the loop invariant is initially true by deriving it from the initialization commands and the precondition.

$$\begin{aligned}
 \{ N \geq 0 \} &\supset \\
 \{ N! = N! \text{ and } N \geq 0 \} & \\
 \quad k := N; & \\
 \{ k! = N! \text{ and } k \geq 0 \} &\supset \\
 \{ 1 \cdot k! = N! \text{ and } k \geq 0 \} & \\
 \quad f := 1; & \\
 \{ f \cdot k! = N! \text{ and } k \geq 0 \} &
 \end{aligned}$$

Note that $N! = N!$ is a tautology when $N \geq 0$, so we can replace it with true. We also know for any clause P that $(P \text{ and true})$ is equivalent to P . Thus we can begin with the initial assertion $N \geq 0$. Some of these implications are actually logical equivalences, but we write implications because that is all we need for the proofs.

To show that the loop invariant is preserved, we start with the invariant at the bottom of the loop and push it back through the body of the loop to prove $\{P \text{ and } B\}$, the loop invariant combined with the entry condition at the top of the loop. Summarizing the proof gives us the following:

$$\begin{aligned} & \{f \bullet k! = N! \text{ and } k > 0\} \supset \\ & \{f \bullet k \bullet (k-1)! = N! \text{ and } k > 0\} \\ & \quad f := f * k; \\ & \{f \bullet (k-1)! = N! \text{ and } k > 0\} \supset \\ & \{f \bullet (k-1)! = N! \text{ and } k-1 \geq 0\} \\ & \quad k := k - 1; \\ & \{f \bullet k! = N! \text{ and } k \geq 0\} \end{aligned}$$

We rely on the fact that k is an integer to transform the condition $k > 0$ into the equivalent condition $k-1 \geq 0$.

Finally, we must prove the assertion after the **while** loop can be derived from $(P \text{ and not } B)$.

$$\begin{aligned} & \{f \bullet k! = N! \text{ and } k \geq 0 \text{ and (not } k > 0)\} \supset \\ & \{f \bullet k! = N! \text{ and } k \geq 0 \text{ and } k \leq 0\} \supset \\ & \{f \bullet k! = N! \text{ and } k = 0\} \supset \\ & \{f = N! \text{ and } k = 0\} \supset \{f = N!\} \end{aligned}$$

The last simplification is a weakening of the assertion $\{f = N! \text{ and } k = 0\}$. ■

While proving this algorithm to be correct, we avoid some problems that occur when the algorithm is executed on a real computer. For example, the factorial function grows very rapidly, and it does not take a large value of N for $N!$ to exceed the storage capacity for integers on a particular machine. However, we want to develop a machine-independent definition of the semantics of a programming language, so we ignore these restrictions. We summarize our axiomatic definitions for Wren in Figure 11.3, including the Skip axiom, which makes no change in the assertion.

Assign	$\{ P[V \rightarrow E] \} V := E \{ P \}$
Read	$\{ IN = [\kappa]L \text{ and } P[V \rightarrow K] \} \text{ read } V \{ IN = L \text{ and } P \}$
Write	$\{ OUT = [L] \text{ and } E = \kappa \text{ and } P \} \text{ write } E \{ OUT = L[\kappa] \text{ and } E = \kappa \text{ and } P \}$
Skip	$\{ P \} \text{ skip } \{ P \}$
Sequence	$\frac{\{ P \} C_1 \{ Q \}, \{ Q \} C_2 \{ R \}}{\{ P \} C_1; C_2 \{ R \}}$
If-Then	$\frac{\{ P \text{ and } B \} C \{ Q \}, (P \text{ and not } B) \supset Q}{\{ P \} \text{ if } B \text{ then } C \text{ end if } \{ Q \}}$
If-Else	$\frac{\{ P \text{ and } B \} C_1 \{ Q \}, \{ P \text{ and not } B \} C_2 \{ Q \}}{\{ P \} \text{ if } B \text{ then } C_1 \text{ else } C_2 \text{ end if } \{ Q \}}$
While	$\frac{\{ P \text{ and } B \} C \{ P \}}{\{ P \} \text{ while } B \text{ do } C \text{ end while } \{ P \text{ and not } B \}}$
Weaken Postcondition	$\frac{\{ P \} C \{ Q \}, Q \supset R}{\{ P \} C \{ R \}}$
Strengthen Precondition	$\frac{P \supset Q, \{ Q \} C \{ R \}}{\{ P \} C \{ R \}}$
And	$\frac{\{ P \} C \{ Q \}, \{ P' \} C \{ Q' \}}{\{ P \text{ and } P' \} C \{ Q \text{ and } Q' \}}$
Or	$\frac{\{ P \} C \{ Q \}, \{ P' \} C \{ Q' \}}{\{ P \text{ or } P' \} C \{ Q \text{ or } Q' \}}$

Figure 11.3 Axiomatic Semantics for Wren

More on Loop Invariants

Constructing loop invariants for **while** commands in a program provides the main challenge when proving correctness with an imperative language. Although no simple formula solves this problem, several general principles can help in analyzing the logic of the loop when finding an invariant.

- A loop invariant describes a relationship among the variables that does not change as the loop is executed. The variables may change their values, but the relationship stays constant.
- Constructing a table of values for the variables that change often reveals a property among variables that does not change.
- Combining what has already been computed at some stage in the loop with what has yet to be computed may yield a constant of some sort.

- An expression related to the test B for the loop can usually be combined with the assertion $\{ \text{not } B \}$ to produce part of the postcondition.
- A possible loop invariant can be assembled to attempt to carry out the proof. We need enough to produce the final postcondition but not so much that we cannot establish the initialization step or prove the preservation of the loop invariant.

Example: Consider a short program that computes the exponential function for two nonnegative integers, M and N . The code specified by means of a precondition and postcondition follows:

```

{  $M > 0$  and  $N \geq 0$  }
  a := M; b := N; k := 1;
  while b > 0 do
    if b = 2 * (b/2)
      then a := a * a; b := b/2
      else b := b - 1; k := k * a
    end if
  end while
{  $k = M^N$  }

```

Recall that division in Wren is integer division. We begin by tracing the algorithm with two small numbers, $M=2$ and $N=7$, and thereby build a table of values to search for a suitable loop invariant. The value $M^N = 128$ remains constant throughout the execution of the loop. Since the goal of the code is to compute the exponential function, we add a column to the table for the value of a^b , since a is the variable that gets multiplied.

a	b	k	a^b
2	7	1	128
2	6	2	64
4	3	2	64
4	2	8	16
16	1	8	16
16	0	128	1

Observe that a^b changes exactly when k changes. In fact, their product is constant, namely 128. This relationship suggests that $k \cdot a^b = M^N$ will be part of the invariant. Furthermore, the loop variable b decreases to 0 but always stays nonnegative. The relation $b \geq 0$ seems to be invariant, and when combined with “not B ”, which is $b \leq 0$, establishes $b=0$ at the end of the loop. When $b=0$ is joined with $k \cdot a^b = M^N$, we get the postcondition $k = M^N$. Thus we have as a loop invariant:

$$\{ b \geq 0 \text{ and } k \cdot a^b = M^N \}.$$

Finally, we verify the program by checking that the loop invariant is consistent with an application of the rule for the **while** command in the given setting.

Initialization

$$\begin{aligned} & \{ M > 0 \text{ and } N \geq 0 \} \supset \\ & \{ M = M > 0 \text{ and } N = N \geq 0 \text{ and } 1 = 1 \} \\ & \quad a := M; \quad b := N; \quad k := 1; \\ & \{ a = M > 0 \text{ and } b = N \geq 0 \text{ and } k = 1 \} \supset \\ & \{ b \geq 0 \text{ and } k \cdot a^b = M^N \} \end{aligned}$$

Preservation

Case 1: b is even, that is, $b = 2i \geq 0$ for some $i \geq 0$.

$$\begin{aligned} & \text{Then } b = 2 \cdot (b/2) \geq 0 \text{ and } b/2 = i \geq 0. \\ & \{ b \geq 0 \text{ and } k \cdot a^b = M^N \text{ and } b > 0 \} \supset \\ & \{ b > 0 \text{ and } k \cdot a^b = M^N \} \supset \\ & \{ b/2 > 0 \text{ and } k \cdot (a \cdot a)^{b/2} = M^N \} \\ & \quad a := a \cdot a; \quad b := b/2 \\ & \{ b > 0 \text{ and } k \cdot a^b = M^N \} \supset \{ b \geq 0 \text{ and } k \cdot a^b = M^N \} \end{aligned}$$

Case 2: b is odd, that is, $b = 2i + 1 > 0$ for some $i \geq 0$.

$$\begin{aligned} & \text{Then } b < 2 \cdot (b/2). \\ & \{ b \geq 0 \text{ and } k \cdot a^b = M^N \text{ and } b > 0 \} \supset \\ & \{ b > 0 \text{ and } k \cdot a^b = M^N \} \supset \\ & \{ b - 1 \geq 0 \text{ and } k \cdot a \cdot a^{b-1} = M^N \} \\ & \quad b := b - 1; \quad k := k \cdot a \\ & \{ b \geq 0 \text{ and } k \cdot a^b = M^N \} \end{aligned}$$

These two cases correspond to the premises in the rule for the **if** command. The conclusion of the axiom establishes:

$$\begin{aligned} & \{ b \geq 0 \text{ and } k \cdot a^b = M^N \text{ and } b > 0 \} \\ & \quad \mathbf{if } b = 2 \cdot (b/2) \mathbf{ then } a := a \cdot a; \quad b := b/2 \\ & \quad \quad \mathbf{else } b := b - 1; \quad k := k \cdot a \mathbf{ end if} \\ & \{ b \geq 0 \text{ and } k \cdot a^b = M^N \} \end{aligned}$$

Completion

$$\begin{aligned} & \{ b \geq 0 \text{ and } k \cdot a^b = M^N \text{ and } b \leq 0 \} \supset \\ & \{ b = 0 \text{ and } k \cdot a^b = M^N \} \supset \{ k = M^N \} \quad \blacksquare \end{aligned}$$

Nested While Loops

Example: We now consider a more complex algorithm with nested **while** loops. In addition to a precondition and postcondition specifying the goal of the code, each **while** loop is annotated by a loop invariant to be supplied in the proof.


```

{ IN = [A] and OUT = [] and A ≥ 0 }
  read x;
  m := 0; n := 0; s := 0;
  while x>0 do { outer loop invariant: C }
    x := x-1; n := m+2; m := m+1;
    while m>0 do { inner loop invariant: D }
      m := m-1; s := s+1
    end while;
  m := n
end while;
write s
{ OUT = [A2] }

```

Imagine for now that an oracle has provided the invariants for this program. Later we discuss how the invariants might be discovered. Given the complexity of the problem, it is convenient to introduce predicate notation to refer to the invariants. The outer invariant C is

$$C(x,m,n,s) = (x \geq 0 \text{ and } m=2(A-x) \text{ and } m=n \geq 0 \text{ and } s=(A-x)^2 \text{ and } OUT=[]).$$

Initialization (outer loop): First we prove that this invariant is true initially by working through the initialization code. Check the deduction from bottom to top.

```

{ IN = [A] and OUT = [] and A ≥ 0 } ⊃
{ A ≥ 0 and 0=2(A-A) and 0=(A-A)2 and IN = [A][ ] and OUT=[] }
  read x;
{ x ≥ 0 and 0=2(A-x) and 0=(A-x)2 and IN = [ ] and OUT=[] } ⊃
{ x ≥ 0 and 0=2(A-x) and 0=0 and 0=(A-x)2 and IN = [ ] and OUT=[] }
  m := 0;
{ x ≥ 0 and m=2(A-x) and m=0 and 0=(A-x)2 and OUT=[] } ⊃
{ x ≥ 0 and m=2(A-x) and m=0 and 0 ≥ 0 and 0=(A-x)2 and OUT=[] }
  n := 0;
{ x ≥ 0 and m=2(A-x) and m=n and n ≥ 0 and 0=(A-x)2 and OUT=[] }
  s := 0;
{ x ≥ 0 and m=2(A-x) and m=n ≥ 0 and s=(A-x)2 and OUT=[] }.

```

Completion (outer loop): Next we show that the outer loop invariant and the exit condition, followed by the **write** command, produce the desired final assertion.

```

{ C(x,m,n,s) and x ≤ 0 }
  ⊃ { x ≥ 0 and m=2(A-x) and m=n ≥ 0 and s=(A-x)2 and OUT=[] and x ≤ 0 }
  ⊃ { x=0 and m=2(A-x) and m=n ≥ 0 and s=(A-x)2 and OUT=[] }
  ⊃ { s=A2 and OUT=[] }
and
{ s=A2 and OUT=[] } write s { s=A2 and OUT=[A2] } ⊃ { OUT=[A2] }.

```

Preservation (outer loop): Showing preservation of the outer loop invariant involves executing the inner loop; we thus introduce the inner loop invariant D , again obtained from the oracle:

$$D(x,m,n,s) = (x \geq 0 \text{ and } n=2(A-x) \text{ and } m \geq 0 \text{ and } n \geq 0 \text{ and } m+s=(A-x)^2 \text{ and } OUT=[]).$$

Initialization (inner loop): We show that the inner loop invariant is initially true by starting with the outer loop invariant, combined with the loop entry condition, and pushing the result through the assignment commands before the inner loop.

$$\begin{aligned} & \{ C(x,m,n,s) \text{ and } x > 0 \} \\ & \equiv \{ x \geq 0 \text{ and } m=2(A-x) \text{ and } m=n \geq 0 \text{ and } s=(A-x)^2 \text{ and } OUT=[] \text{ and } x > 0 \} \\ & \supset \{ x-1 \geq 0 \text{ and } m+2=2(A-x+1) \text{ and } m+1 \geq 0 \text{ and } m+2 \geq 0 \\ & \qquad \qquad \qquad \text{and } m+1+s=(A-x+1)^2 \text{ and } OUT=[] \} \\ & \equiv \{ D(x-1,m+1,m+2,s) \} \\ & \text{since } (s=(A-x)^2 \text{ and } m+2=2(A-x+1)) \supset m+1+s=(A-x+1)^2. \end{aligned}$$

Therefore, using the assignment rule, we have

$$\begin{aligned} & \{ C(x,m,n,s) \text{ and } x > 0 \} \supset \{ D(x-1,m+1,m+2,s) \} \\ & \quad x := x-1; \quad n := m+2; \quad m := m+1 \\ & \{ D(x,m,n,s) \}. \end{aligned}$$

Preservation (inner loop): Next we need to show that the inner loop invariant is preserved, that is,

$$\{ D(x,m,n,s) \text{ and } m > 0 \} \quad m := m-1; \quad s := s+1 \quad \{ D(x,m,n,s) \}.$$

It suffices to show

$$\begin{aligned} & (D(x,m,n,s) \text{ and } m > 0) \\ & \supset (x \geq 0 \text{ and } n=2(A-x) \text{ and } m \geq 0 \text{ and } n \geq 0 \\ & \qquad \qquad \qquad \text{and } m+s=(A-x)^2 \text{ and } OUT=[] \text{ and } m > 0) \\ & \supset (x \geq 0 \text{ and } n=2(A-x) \text{ and } m-1 \geq 0 \text{ and } n \geq 0 \\ & \qquad \qquad \qquad \text{and } m-1+s+1=(A-x)^2 \text{ and } OUT=[]) \\ & \equiv D(x,m-1,n,s+1). \end{aligned}$$

The preservation step is complete because after the assignments, m replaces $m-1$ and s replaces $s+1$ to produce the loop invariant $D(x,m,n,s)$.

Completion (inner loop): To complete our proof, we need to show that the inner loop invariant, combined with the inner loop exit condition and pushed through the assignment $m := n$, results in the outer loop invariant:

$$\{ D(x,m,n,s) \text{ and } m \leq 0 \} \quad m := n \quad \{ C(x,m,n,s) \}.$$

It suffices to show $(D(x,m,n,s) \text{ and } m \leq 0) \supset C(x,n,n,s)$:

$(D(x,m,n,s) \text{ and } m \leq 0)$
 $\supset (x \geq 0 \text{ and } n = 2(A-x) \text{ and } m \geq 0 \text{ and } n \geq 0$
 $\qquad \qquad \qquad \text{and } m+s = (A-x)^2 \text{ and } \text{OUT} = [] \text{ and } m \leq 0)$
 $\supset (x \geq 0 \text{ and } n = 2(A-x) \text{ and } n = n \geq 0 \text{ and } s = (A-x)^2 \text{ and } \text{OUT} = [])$
 $\equiv C(x,n,n,s).$

Thus the outer loop invariant is preserved. ■

The previous verification suggests a derived rule for assignment commands:

$$\frac{P \supset Q[V \rightarrow E]}{\{ P \} V := E \{ Q \}}$$

We used an application of this derived rule when we proved

$$(C(x,m,n,s) \text{ and } x > 0) \supset D(x-1, m+1, m+2, s)$$

from which we deduced

$$\{ C(x,m,n,s) \text{ and } x > 0 \}$$

$$x := x-1; n := m+2; m := m+1$$

$$\{ D(x,m,n,s) \}.$$

Proving a program correct is a fairly mechanical process once the loop invariants are known. We have already suggested that one way to discover a loop invariant is to make a table of values for a simple case and to trace values for the relevant variables. To see how tracing can be used, let $A = 3$ in the previous example and hand execute the loops. The table of values is shown in Figure 11.4.

The positions where the invariant $C(x,m,n,s)$ for the outer loop should hold are marked by arrows. Note how the variable s takes the values of the perfect squares—namely, 0, 1, 4, and 9—at these locations. The difficulty is to determine what s is the square of as its value increases.

Observe that x decreases as the program executes. Since A is constant, this means the value $A-x$ increases: 0, 1, 2, and 3. This gives us the relationship $s = (A-x)^2$. We also note that m is always even and increases: 0, 2, 4, 6. This produces the relation $m = 2(A-x)$ in the outer invariant.

For the inner loop invariant, s is not always a perfect square, but $m+s$ is. Also, in the inner loop, n preserves the final value for m as the loop executes. So n also obeys the relationship $n = 2(A-x)$.

Finally, the loop entry conditions are combined with the exit condition. For the outer loop, $x > 0$ is combined with $x = 0$ to produce the condition $x \geq 0$ for the outer loop invariant. In a similar way, $m > 0$ is combined with $m = 0$ to give $m \geq 0$ in the inner loop invariant.

	x	m	n	s	A-x																												
→	3	0	0	0	0																												
	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; border-right: 1px solid black; padding: 2px 5px;">2</td> <td style="width: 15%; padding: 2px 5px;">1</td> <td style="width: 15%; padding: 2px 5px;">2</td> <td style="width: 15%; padding: 2px 5px;">0</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">→</td> <td style="border: 1px solid black; border-radius: 10px; padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">1</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">1</td> </tr> </table>				2	1	2	0	→	0	2	1	2	2	2	1	1																
2	1	2	0																														
→	0	2	1																														
2	2	2	1																														
	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; border-right: 1px solid black; padding: 2px 5px;">1</td> <td style="width: 15%; padding: 2px 5px;">3</td> <td style="width: 15%; padding: 2px 5px;">4</td> <td style="width: 15%; padding: 2px 5px;">1</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">→</td> <td style="border: 1px solid black; border-radius: 10px; padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">4</td> <td style="padding: 2px 5px;">2</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">4</td> <td style="padding: 2px 5px;">4</td> <td style="padding: 2px 5px;">3</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">→</td> <td style="border: 1px solid black; border-radius: 10px; padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">4</td> <td style="padding: 2px 5px;">4</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">4</td> <td style="padding: 2px 5px;">4</td> <td style="padding: 2px 5px;">4</td> </tr> </table>				1	3	4	1	→	2	4	2	1	4	4	3	→	0	4	4	1	4	4	4	2								
1	3	4	1																														
→	2	4	2																														
1	4	4	3																														
→	0	4	4																														
1	4	4	4																														
	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; border-right: 1px solid black; padding: 2px 5px;">0</td> <td style="width: 15%; padding: 2px 5px;">5</td> <td style="width: 15%; padding: 2px 5px;">6</td> <td style="width: 15%; padding: 2px 5px;">4</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">→</td> <td style="border: 1px solid black; border-radius: 10px; padding: 2px 5px;">4</td> <td style="padding: 2px 5px;">6</td> <td style="padding: 2px 5px;">5</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">6</td> <td style="padding: 2px 5px;">6</td> <td style="padding: 2px 5px;">6</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">→</td> <td style="border: 1px solid black; border-radius: 10px; padding: 2px 5px;">3</td> <td style="padding: 2px 5px;">6</td> <td style="padding: 2px 5px;">7</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">6</td> <td style="padding: 2px 5px;">6</td> <td style="padding: 2px 5px;">8</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">→</td> <td style="border: 1px solid black; border-radius: 10px; padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">6</td> <td style="padding: 2px 5px;">9</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">6</td> <td style="padding: 2px 5px;">6</td> <td style="padding: 2px 5px;">9</td> </tr> </table>				0	5	6	4	→	4	6	5	0	6	6	6	→	3	6	7	0	6	6	8	→	2	6	9	0	6	6	9	3
0	5	6	4																														
→	4	6	5																														
0	6	6	6																														
→	3	6	7																														
0	6	6	8																														
→	2	6	9																														
0	6	6	9																														

Figure 11.4: Tracing Variable Values

Since finding the loop invariant is the most difficult part of proving a program correct, we present one more example. Consider the following program:

```

{ IN = [A] and A ≥ 2 }
  read n; b := true; d := 2;
  while d < n and b do { loop invariant }
    if n = d*(n/d) then b := false end if;
    d := d+1
  end while
{ b ≡ ∀k[2 ≤ k < A ⊃ not ∃j[k•j = A]] }
    
```

The Boolean variable *b* is a flag, remaining true if no divisor of *n*, other than 1, is found—in other words, if *n* is prime. If a divisor is found, *b* is set to false and remains false. Here the invariant needs to record the partial results computed so far as the loop is executed.

At each stage in the loop, the potential divisors have been checked successfully up to but not including the current value of *d*. We use the final assertion

as a guide for constructing the invariant that expresses the portion of the computation completed so far.

Invariant = $([b \equiv \forall k[2 \leq k < d \supset \text{not } \exists j[k \cdot j = A]]]$ and $n=A \geq 2$ and $2 \leq d \leq n$).

The remainder of the proof is left as an exercise.

Exercises

1. Give a deduction that verifies the specification of the following program fragment:

$$\{ x=A \text{ and } y=B \} \ z:=x; \ x:=y; \ y:=z \ \{ x=B \text{ and } y=A \}.$$

2. Define a proof rule for the repeat command.

$$\{ P \} \ \mathbf{repeat} \ C \ \mathbf{until} \ B \ \{ Q \text{ and } B \}$$

Use this proof rule to verify the partial correctness of the program segment shown below:

$$\begin{aligned} &\{ m = A > 0 \text{ and } n = B \geq 0 \} \\ &\quad p := 1; \\ &\quad \mathbf{repeat} \\ &\quad \quad p := p * n; \ m := m - 1 \\ &\quad \mathbf{until} \ m = 0 \\ &\{ p = B^A \} \end{aligned}$$

3. Prove the partial correctness of the following program for integer multiplication by repeated addition.

$$\begin{aligned} &\{ B \geq 0 \} \\ &\quad x := A; \ y := B; \ \mathbf{product} := 0; \\ &\quad \mathbf{while} \ y > 0 \ \mathbf{do} \\ &\quad \quad \mathbf{product} := \mathbf{product} + x; \ y := y - 1 \\ &\quad \mathbf{end} \ \mathbf{while} \\ &\{ \mathbf{product} = A \cdot B \} \end{aligned}$$

4. Prove the partial correctness of this more efficient integer multiplication program.

$$\begin{aligned} &\{ m = A \text{ and } n = B \geq 0 \} \\ &\quad x := m; \ y := n; \ \mathbf{product} := 0; \\ &\quad \mathbf{while} \ y > 0 \ \mathbf{do} \\ &\quad \quad \mathbf{if} \ 2 * (y/2) <> y \ \mathbf{then} \ \mathbf{product} := \mathbf{product} + x \ \mathbf{end} \ \mathbf{if}; \\ &\quad \quad x := 2 * x; \ y := y / 2 \\ &\quad \mathbf{end} \ \mathbf{while}; \\ &\{ \mathbf{product} = A \cdot B \} \end{aligned}$$

Hint: Consider the two cases where y is even ($y = 2k$) and y is odd ($y = 2k+1$). Remember that $/$ denotes integer division.

5. Finish the proof of the prime number detection program.
6. The **least common multiple** of two positive integers m and n , $\text{LCM}(m,n)$, is the smallest integer k such that $k=i*m$ and $k=j*n$ for some integers i and j . Write a Wren program segment that for integer variables m and n will set another variable, say k , to the value $\text{LCM}(m,n)$. Give a formal proof of the partial correctness of the program fragment.
7. Provide postconditions for these code fragments and show their partial correctness.
- a) $\{ m = A \geq 0 \}$
 $r := 0;$
while $(r+1)*(r+1) \leq m$ **do** $r:=r+1$ **end while**
{ Postcondition }
- b) $\{ m = A \geq 0 \}$
 $x:=0; \text{ odd}:=1; \text{ sum}:=1;$
while $\text{sum} \leq m$ **do**
 $x:=x+1; \text{ odd}:=\text{odd}+2; \text{ sum}:=\text{sum}+\text{odd}$
end while
{ Postcondition }
- c) $\{ A \geq 0 \text{ and } B \geq 0 \}$
 $\text{sum}:=0; m:=A;$
while $m \geq 0$ **do**
 $\text{count} := 0;$
while $\text{count} \leq B$ **do**
 $\text{sum} := \text{sum}+1; \text{count} := \text{count}+1$
end while;
 $m := m-1$
end while
{ Postcondition }
8. Write a fragment of Wren code C satisfying the following specification:
 $\{ M \geq 0 \text{ and } K \geq 0 \}$
 C
 $\{ \text{result} = b_K \text{ and } M = b_0 + b_1 \cdot 2 + \dots + b_j \cdot 2^j + \dots \text{ where } b_j = 0 \text{ or } 1 \}.$
 Prove that the code is partially correct with respect to the specification.
9. Carefully define the substitution operation $P[V \rightarrow E]$ for the predicate calculus. Be careful to avoid the problem of free variable capture. See substitution for the lambda calculus in Chapter 5.

10. Supply proofs of partial correctness for the following examples:

- a) $\{ N \geq 0 \}$
 sum:=0; exp:=0; term:=1;
while exp<N **do**
 sum := sum+term; exp := exp+1; term := term*2
end while
 $\{ \text{sum} = 2^N - 1 \}$
- b) $\{ N \geq 0 \text{ and } D > 0 \}$
 q:=0; r:=N;
while r>=D **do**
 r := r-D; q := q+1
end while
 $\{ N = q \cdot D + r \text{ and } 0 \leq r < D \}$
- c) $\{ \text{true} \}$
 k:=1; c:=0; sum:=0;
while sum<=1000 **do**
 sum := sum+k*k; c := c+1; k := k+1
end while
 $\{ \text{"c is the smallest number of consecutive squares starting at 1 whose sum is greater than 1000"} \}$
- d) $\{ N > 0 \text{ and } N \text{ is odd} \}$
 sum:=1; term:=1;
while term<>N **do**
 term := term+2; sum := sum+2*term-1;
end while
 $\{ \text{sum} = N \cdot (N+1) / 2 \}$
- e) $\{ \text{true} \}$
 sum:=0; term:=1;
while term<10000 **do**
 sum := sum+term; term := 10*term;
end while
 $\{ \text{sum} = 1111 \}$
- f) $\{ N \geq 2 \}$
 k:=N; fact:=1; p:=1;
while k<>1 **do**
 k := k-1; temp := fact;
 fact := k*(p+fact); p := p+temp
end while
 $\{ \text{fact} = N! \}$

```

g) {  $A \geq 0$  and  $B \geq 0$  }
   m := A; n := B; product := 0;
   while m <> 0 do
     while 2*(m/2)=m do
       n := 2*n; m := m/2
     end while;
   product := product+n; m := m-1
   end while
{ product = A*B }

```

11. Suppose Wren has been extended to include an exponentiation operation \uparrow . Prove the partial correctness of the following code segment.

```

{  $m = A \geq 1$  }
s := 1; k := 0;
while s < m do
  s := s + 2k; k := k+1
end while
{  $\log_2 A \leq k < 1 + \log_2 A$  }

```

11.3 AXIOMATIC SEMANTICS FOR PELICAN

Pelican, first introduced in Chapter 9, is an extension of Wren that includes the following features:

- Declarations of constants, procedures with no parameters, and procedures with a single parameter.
- Anonymous blocks with a declaration section and a command section.
- Procedure calls as commands.

Figure 11.5 restates the abstract syntax of Pelican.

Now we need to include the declarations in the axiomatic semantics. We assume that all programs have been checked independently to satisfy all syntactic rules and that only syntactically valid programs, including those that adhere to the context sensitive-conditions, are analyzed semantically. Some restrictions on the choice of identifier names will be introduced so that our presentation of the axiomatic semantics of Pelican does not become bogged down with syntactic details.

Since we do not have an underlying model for environments that can differentiate between different uses of the same identifier in different scopes, we require that all identifiers be named uniquely throughout the program. No generality is lost by such a restriction since any program with duplicate identifier names can be transformed into a program with unique names by sys-

tematic substitutions of identifier names within the scope of the identifier. For example, consider the following Pelican program with duplicate identifier names:

```

program squaring is
  var x, y: integer;
  procedure square(x : integer) is
    var y: integer;
    begin
      y := x * x; write y
    end
  begin
    read x; read y; square(x); square(y)
  end

```

Abstract Syntactic Domains

P : Program C : Command N : Numeral
 B : Block E : Expression I : Identifier
 D : Declaration O : Operator L : Identifier⁺
 T : Type

Abstract Production Rules

Program ::= **program** Identifier **is** Block
 Block ::= Declaration **begin** Command **end**
 Declaration ::= ε | Declaration Declaration
 | **const** Identifier = Expression
 | **var** Identifier : Type | **var** Identifier Identifier⁺ : Type
 | **procedure** Identifier **is** Block
 | **procedure** Identifier (Identifier : Type) **is** Block
 Type ::= **integer** | **boolean**
 Command ::= Command ; Command | Identifier := Expression
 | **read** Identifier | **write** Expression | **skip** | **declare** Block
 | **if** Expression **then** Command **else** Command
 | **while** Expression **do** Command | Identifier
 | **if** Expression **then** Command | Identifier(Expression)
 Expression ::= Numeral | Identifier | **true** | **false** | - Expression
 | Expression Operator Expression | **not**(Expression)
 Operator ::= + | - | * | / | **or** | **and** | <= | < | = | > | >= | <>

Figure 11.5: Abstract Syntax for Pelican

The renaming works as follows: The first occurrence of the identifier name remains unchanged while each other occurrence in a different scope is systematically substituted with the same name followed by a numeric suffix (1, 2, 3, ..., as needed) that makes the name unique. To make sure this substitution does not result in duplication of other declarations, we mark it with a unique character, such as the sharp sign # shown below, that is not allowed in the original syntax. Using this scheme, the program given above becomes:

```
program squaring is
  var x, y: integer;
  procedure square(x#1 : integer) is
    var y#1: integer;
    begin
      y#1 := x#1 * x#1; write y#1
    end
  begin
    read x; read y; square(x); square(y)
  end
```

We inherit all of the axioms from Wren: Assign, Read, Write, Skip, Sequence, If-Then, If-Else, While, Weaken Postcondition, Strengthen Precondition, And, and Or. We also need to introduce an alternative form for rules of inference:

$$\frac{H_1, H_2, \dots, H_n \vdash H_{n+1}}{H}$$

This rule can be interpreted as follows:

If H_{n+1} can be derived from H_1, H_2, \dots, H_n , we may conclude H .

Blocks

Although we do not need to retain declaration information for context checking, which we assume has already been performed, we do need a mechanism for retaining pertinent declaration information, such as constant values, the bodies of procedure declarations, and their formal parameters, if applicable. This task is accomplished by two assertions, Procs and Const, which will depend on the declarations in the program being analyzed. We define Procs to be a set of assertions constructed as follows:

- If p is a declared parameterless procedure with body B , add $\text{body}(p) = B$ to Procs.
- If p is a declared procedure with formal parameter F and body B , add $\text{parameter}(p)=F$ and $\text{body}(p)=B$ to Procs.

Constant declarations are handled by adding an assertion Const such that, for each declared constant c with value N , Const contains an assertion $c = N$.

For a constant declaration with an arbitrary expression, $c = E$, the assertion takes the form $c = \kappa$ where κ is the current value of E . In the event that there are no declared constants, $\text{Const} \equiv \text{true}$. With these mechanisms, we can give an axiomatic definition for a block:

$$\frac{\text{Procs} \vdash \{ P \text{ and } \text{Const} \} \ C \ \{ Q \}}{\{ P \} \ D \ \mathbf{begin} \ C \ \mathbf{end} \ \{ Q \}} \quad (\text{Block})$$

Example: Before continuing with the development of other new axiomatic definitions, we demonstrate how the block rule works for the following anonymous block, **declare** B, with a constant declaration:

```
declare
  constant x = 10;
  var y : integer;
begin
  read y; y := x + y; write y
end
```

Suppose we want to prove that

$$\{ IN = [7]L \text{ and } OUT = [] \} \ B \ \{ OUT = [17] \}.$$

Since no procedures are declared, Procs contains no assertions, but Const contains the assertion $x = 10$. We must show

$$\begin{aligned} &\{ IN = [7]L \text{ and } OUT = [] \text{ and } x = 10 \} \\ &\quad \mathbf{read} \ y; \ y := x + y; \ \mathbf{write} \ y \\ &\{ OUT = [17] \}. \end{aligned}$$

The proof proceeds as follows:

$$\begin{aligned} &\{ IN = [7]L \text{ and } OUT = [] \text{ and } x = 10 \} \supset \\ &\{ IN = [7]L \text{ and } OUT = [] \text{ and } x = 10 \text{ and } 7 = 7 \} \\ &\quad \mathbf{read} \ y \\ &\{ IN = L \text{ and } OUT = [] \text{ and } x = 10 \text{ and } y = 7 \} \supset \\ &\{ IN = L \text{ and } OUT = [] \text{ and } x = 10 \text{ and } x+y = 10+7 \} \\ &\quad y := x + y \\ &\{ IN = L \text{ and } OUT = [] \text{ and } x = 10 \text{ and } y = 17 \} \\ &\quad \mathbf{write} \ y \\ &\{ IN = L \text{ and } OUT = [17] \text{ and } x = 10 \text{ and } y = 17 \} \supset \\ &\{ OUT = [17] \}. \end{aligned}$$

Nonrecursive Procedures

Pelican requires four separate axiomatic definitions for procedure calls: nonrecursive calls without and with a parameter and recursive calls without and with a parameter. Calling a nonrecursive procedure without a parameter involves proving the logical relation of assertions around the execution of the body of the procedure. The subscript on the name of the rule indicates no parameter for the procedure.

$$\frac{\{P\} B \{Q\}, \text{ body(proc)} = B}{\{P\} \text{ proc } \{Q\}} \quad (\text{Call}_0)$$

Example: Consider this anonymous block **declare** B that squares the existing value of x:

```

declare
  procedure square is
    begin
      x := x * x
    end
begin
  square
end

```

For this block, Procs is the assertion

$$\text{body(square)} = (x := x * x)$$

and Const is the true assertion. So, using the Block rule, we need to show

$$\text{body(square)} = (x := x * x) \vdash \{x = N \text{ and true}\} \text{ square } \{x = N * N\}.$$

The first assertion in the hypothesis of Call_0 requires that we prove $\{x = N \text{ and true}\} B \{x = N * N\}$.

Since $\{P \text{ and true}\}$ is equivalent to P , using the rule for a procedure invocation without a parameter, we need to prove

$$\{x = N\} x := x * x \{x = N * N\}.$$

Substituting $x * x$ for x in the postcondition, we have $\{x * x = N * N\}$. Because we know $\{x = N\} \supset \{x * x = N * N\}$, we strengthen the precondition to obtain the initial assertion. ■

If a procedure P has a formal parameter F and the procedure invocation has an expression E as the actual parameter, we add the binding of F to E in both the precondition and postcondition to prove the procedure call is correct.

$$\frac{\{P\} B \{Q\}, \text{ body(proc)} = B, \text{ parameter(proc)} = F}{\{P[F \rightarrow E]\} \text{ proc}(E) \{Q[F \rightarrow E]\}} \quad (\text{Call}_1)$$

If we can show the relation $\{P\} B \{Q\}$ is true about F where $B = \text{body}(\text{proc})$, we may conclude that the relation $\{P[F \rightarrow E]\} \text{ proc}(E) \{Q[F \rightarrow E]\}$ is true about E .

Example: Consider an anonymous block **declare** B that increments the existing value of a nonlocal variable x by an amount specified as a parameter:

```

declare
  procedure increment(step : integer) is
    begin
      x := x + step
    end
begin
  increment(y)
end

```

We want to prove $\{x = M \text{ and } y = N\} B \{x = M + N \text{ and } y = N\}$.

For this block, Procs contains the conjunction of the assertions

```

body(increment) = (x := x+step)
parameter(increment) = step,

```

and Const is the true assertion. We thus need to show

$$\text{body}(\text{increment}) = (x := x + \text{step}), \text{ parameter}(\text{increment}) = \text{step} \\ \vdash \{x = M \text{ and } y = N \text{ and true}\} \text{ increment}(y) \{x = M + N \text{ and } y = N\}.$$

We can eliminate the “*and true*”; then using our rule for a procedure invocation with parameter, we have to show

$$\{x = M \text{ and } \text{step} = N\} \\ x := x + \text{step} \\ \{x = M + N \text{ and } \text{step} = N\}$$

Substituting “ $x + \text{step}$ ” for x in the postcondition, we have

$$\{x + \text{step} = M + N \text{ and } \text{step} = N\} \supset \\ \{x + N = M + N \text{ and } \text{step} = N\} \supset \\ \{x = M \text{ and } \text{step} = N\}$$

the desired precondition. Therefore, by the rule Call_1 , we may conclude

$$\{x = M \text{ and } y = N \text{ and true}\} \text{ increment}(y) \{x = M + N \text{ and } y = N\}. \quad \blacksquare$$

Although not illustrated by the previous example, we must introduce some restrictions on parameter usage so as to avoid aliasing and thereby proving false assertions. Neither of these restrictions results in any loss of generality. Since we want to have parameters passed by value, any changes in the for-

mal parameter inside the procedure should not be visible outside the procedure. This situation becomes a problem if the actual parameter is a variable.

We avoid the problem by not allowing the formal parameter to change value inside the procedure command sequence. Any program violating this restriction can be transformed into an equivalent program that obeys the restriction by declaring a new local variable, assigning this variable the value of the parameter, and then using the local variable in the place of the parameter throughout the procedure. For example, the code on the left allows the formal parameter f to change value but the corresponding code on the right permits only a local variable to change value.

```
procedure p (f : integer) is
begin
  f := f * f;
  write f
end
```

```
procedure p (f : integer) is
  var local#f : integer;
begin
  local#f := f;
  local#f := local#f * local#f;
  write local#f
end
```

The second restriction requires that if the actual parameter is a variable that is manipulated globally inside the procedure body, no change is made to the value of the formal parameter for which it is substituted. The procedure given below changes two nonlocal variables. We are concerned only with changes made to the variable x , which happens to be the actual parameter. The constraint adds a new variable at the level of invocation, assigning the value of the “manipulated” variable to the new variable, and passing the new variable as a parameter. This transformation is illustrated below by altering the variable “ x ” by appending “new#” in the calling environment and passing “new# x ” as the actual parameter.

```
procedure q (f : integer ) is
begin
  read x;
  y := y + f
end
:
p(x);
```

```
procedure q (f : integer ) is
begin
  read x;
  y := y + f
end
:
new#x := x;
p(new#x);
```

Exercises at the end of this section provide Pelican programs for which erroneous semantics can be proved using the Call_1 rule when these transformations are ignored.

Recursive Procedures

Next we discuss recursive procedures without a parameter. Consider the following procedure that reads and discards all zeros until the first nonzero value is encountered.

```

procedure nonzero is
  begin
    read x;
    if x = 0 then nonzero end if
  end

```

We cannot use the rule for a nonrecursive procedure without a parameter because we will have an endless sequence of applications of the same rule. To see how to avoid this problem, we use a technique similar to mathematical induction. Recall that with induction we have to show a base case and to prove that the proposition is true for n assuming that it is true for $n-1$. With recursion, we use a similar approach: We prove that the current call is correct if we assume that the result from any previous call is correct. The basis case corresponds to the situation in which the procedure is called, but it does not call itself again.

$$\frac{\{P\} \text{ proc } \{Q\} \vdash \{P\} \text{ C } \{Q\}, \text{ body}(\text{proc}) = \text{C}}{\{P\} \text{ proc } \{Q\}} \quad (\text{Recursion}_0)$$

Example: For the procedure nonzero given above, suppose that the input file contains a sequence Z of zero or more 0's followed by a nonzero value, call it N , followed by any sequence of values L . We want to prove

$$\{IN = Z[N]L \text{ and } Z \text{ contains only zeros and } N \neq 0\} = P$$

nonzero

$$\{IN = L \text{ and } x = N \neq 0\} = Q.$$

To prove the correctness of the procedure call relative to the given specification, we need to show the following correctness specification for the body of the procedure

$$\{IN = Z[N]L \text{ and } Z \text{ contains only zeros and } N \neq 0\} = P$$

```

read x;
if x = 0 then nonzero end if

```

$$\{IN = L \text{ and } x = N \neq 0\} = Q$$

where we are allowed to use the recursive assumption when nonzero is called from within itself. We make an assertion between the **read** command and the **if** command that takes into account two cases: Either x is zero or x is nonzero.

In the case that the sequence of zeros is not empty, we can write

$$Z = [0]Z', \text{ where } Z' \text{ contains zero or more } 0\text{'s,}$$

and in the other case, Z is empty. Therefore the precondition P is equivalent to

$$((IN = [0]Z'[N]L \text{ and } Z' \text{ contains only zeros and } N \neq 0) \text{ or } (IN = [N]L \text{ and } N \neq 0))$$

Case 1: Z is not empty.

$$\{ IN = [0]Z'[N]L \text{ and } Z' \text{ contains only zeros and } N \neq 0 \}$$

read x

$$\{ IN = Z'[N]L \text{ and } Z' \text{ contains only zeros and } N \neq 0 \text{ and } x = 0 \}.$$

Case 2: Z is empty.

$$\{ IN = [N]L \text{ and } N \neq 0 \} \text{ **read** } x \{ IN = L \text{ and } x = N \neq 0 \}.$$

Applying the Or rule allows us to conclude the following assertion, called R, after the **read** command:

$$R = ((IN = Z'[N]L \text{ and } Z' \text{ contains only zeros and } N \neq 0 \text{ and } x = 0) \\ \text{ or } (IN = L \text{ and } x = N \neq 0)).$$

Using the If-Then rule, we must show:

$$\{ R \text{ and } x = 0 \} \text{ nonzero } \{ IN = L \text{ and } x = N \neq 0 \} \text{ and} \\ (R \text{ and } x \neq 0) \supset (IN = L \text{ and } x = N \neq 0).$$

The second assertion holds directly since $(R \text{ and } x \neq 0)$ implies the final assertion. The first assertion involving the recursive call simplifies to

$$\{ IN = Z'[N]L \text{ and } N \neq 0 \text{ and } x = 0 \} \text{ nonzero } \{ IN = L \text{ and } x = N \neq 0 \}.$$

This is a stronger precondition than we require, so it suffices to prove:

$$\{ IN = Z'[N]L \text{ and } N \neq 0 \} \text{ nonzero } \{ IN = L \text{ and } x = N \neq 0 \}.$$

But this is exactly the recursive assertion, $\{P\} \text{ nonzero } \{Q\}$, which we may assume to be true (the induction hypothesis), so the proof is complete. ■

Finally, we consider an inference rule for a recursively defined procedure with a parameter. The axiomatic definition follows directly from recursion without a parameter, modified by the changes inherent in calling a procedure with a parameter.

$$\frac{\forall f (\{P[F \rightarrow f]\} \text{ proc}(f) \{Q[F \rightarrow f]\}) \vdash \{P\} C \{Q\}, \text{ body}(\text{proc})=C, \text{ parameter}(\text{proc})=F}{\{P[F \rightarrow E]\} \text{ proc}(E) \{Q[F \rightarrow E]\}} \quad (\text{Recursion}_1)$$

The induction hypothesis allows us to assume the correctness of a recursive call of the procedure with any expression that satisfies the precondition as the actual parameter.

Example: To see how this rule works, we prove the correctness of a recursively defined factorial program. Since we do not have procedures that return values, we depend on a global variable “fact” to hold the current value as we return from the recursive calls.

```

procedure factorial(n : integer) is
  begin
    if n = 0 then fact := 1
      else factorial(n-1); fact := n*fact;
    end if;
  end;

```

We want to prove

$$\{ \text{num} = \kappa \geq 0 \} \text{ factorial}(\text{num}) = \text{P}[\text{F} \rightarrow \text{E}]$$

$$\{ \text{fact} = \text{num}! \text{ and } \text{num} = \kappa \} = \text{Q}[\text{F} \rightarrow \text{E}], \text{ which implies } \text{fact} = \kappa!.$$

In the proof below, “num” refers to the original actual parameter (called E in the rule) and “n” refers to the formal parameter (called F) in the recursive definition. Substituting the body of the procedure, we must show

$$\{ n = \kappa \geq 0 \} = \text{P}$$

```

if n = 0 then fact := 1
  else factorial(n-1); fact := n*fact;
end if;

```

$$\{ \text{fact} = n! \text{ and } n = \kappa \} = \text{Q}$$

assuming as an induction hypothesis

$$\forall f(\{ f = \kappa \geq 0 \} \text{ factorial}(f) = \text{P}[\text{F} \rightarrow f])$$

$$\{ \text{fact} = f! \text{ and } f = \kappa \} = \text{Q}[\text{F} \rightarrow f].$$

Case 1: $n = 0$.

Use the If-Else rule for the case when the condition is true:

$$\{ n = \kappa \geq 0 \text{ and } n = 0 \} \supset$$

$$\{ n = \kappa = 0 \text{ and } 1 = 0! = \kappa! \}$$

$$\text{fact} := 1$$

$$\{ n = \kappa = 0 \text{ and } \text{fact} = 0! = n! \} \supset \{ \text{fact} = n! \text{ and } n = \kappa \}.$$

Case 2: $n > 0$.

The recursive assumption with $f=n-1$ gives

$$\begin{aligned} & \{ n = \kappa \geq 0 \text{ and } n > 0 \} \supset \\ & \{ n-1 = \kappa-1 \geq 0 \} \\ & \quad \text{factorial}(n-1) \\ & \{ \text{fact} = (n-1)! \text{ and } n-1 = \kappa-1 \} \supset \\ & \{ \text{fact} = (n-1)! \} \end{aligned}$$

The Assign rule gives

$$\begin{aligned} & \{ \text{fact} = (n-1)! \} \supset \\ & \{ n \cdot \text{fact} = n \cdot (n-1)! \} \\ & \quad \text{fact} := n * \text{fact} \\ & \{ \text{fact} = n \cdot (n-1)! = n! \}, \text{ which is the desired postcondition.} \quad \blacksquare \end{aligned}$$

The complete axiomatic definition for Pelican is presented in Figure 11.6.

Assign	$\{ P[V \rightarrow E] \} V := E \{ P \}$
Read	$\{ IN = [\kappa]L \text{ and } P[V \rightarrow K] \} \text{ read } V \{ IN = L \text{ and } P \}$
Write	$\{ OUT = [L] \text{ and } E = \kappa \text{ and } P \} \text{ write } E \{ OUT = L[\kappa] \text{ and } E = \kappa \text{ and } P \}$
Skip	$\{ P \} \text{ skip } \{ P \}$
Sequence	$\frac{\{ P \} C_1 \{ Q \}, \{ Q \} C_2 \{ R \}}{\{ P \} C_1; C_2 \{ R \}}$
If-Then	$\frac{\{ P \text{ and } B \} C \{ Q \}, (P \text{ and not } B) \supset Q}{\{ P \} \text{ if } B \text{ then } C \text{ end if } \{ Q \}}$
If-Else	$\frac{\{ P \text{ and } B \} C_1 \{ Q \}, \{ P \text{ and not } B \} C_2 \{ Q \}}{\{ P \} \text{ if } B \text{ then } C_1 \text{ else } C_2 \text{ end if } \{ Q \}}$
While	$\frac{\{ P \text{ and } B \} C \{ P \}}{\{ P \} \text{ while } B \text{ do } C \text{ end while } \{ P \text{ and not } B \}}$
Block	$\frac{\text{Procs} \vdash \{ P \text{ and Const } \} C \{ Q \}}{\{ P \} D \text{ begin } C \text{ end } \{ Q \}}$

where for all declarations “**procedure I is B**” in D,
“body(I) = B” is contained in Procs;
for all declarations “**procedure I(F) is B**” in D,
“body(I) = B and parameter(I) = F” is contained in Procs; and
Const consists of a conjunction of true and $c_i = E_i$
for each constant declaration of the form “**const** $c_i = E_i$ ” in D.

Figure 11.6: Axiomatic Semantics for Pelican (Part 1)

Call without Parameter (Call₀)

$$\frac{\{P\} B \{Q\}, \text{ body(proc)} = B}{\{P\} \text{ proc } \{Q\}}$$

Call with Parameter (Call₁)

$$\frac{\{P\} B \{Q\}, \text{ body(proc)} = B, \text{ parameter(proc)} = F}{\{ P[F \rightarrow E] \} \text{ proc}(E) \{ Q[F \rightarrow E] \}}$$

Recursion without Parameter (Recursion₀)

$$\frac{\{P\} \text{ proc } \{Q\} \vdash \{P\} B \{Q\}, \text{ body(proc)} = B}{\{P\} \text{ proc } \{Q\}}$$

Recursion with Parameter (Recursion₁)

$$\frac{\forall f(\{P[F \rightarrow f]\} \text{ proc}(f)\{Q[F \rightarrow f]\}) \vdash \{P\} B \{Q\}, \text{ body(proc)}=B, \text{ parameter(proc)}=F}{\{ P[F \rightarrow E] \} \text{ proc}(E) \{ Q[F \rightarrow E] \}}$$

Weaken
Postcondition

$$\frac{\{P\} C \{Q\}, Q \supset R}{\{P\} C \{R\}}$$

Strengthen
Precondition

$$\frac{P \supset Q, \{Q\} C \{R\}}{\{P\} C \{R\}}$$

And

$$\frac{\{P\} C \{Q\}, \{P'\} C \{Q'\}}{\{P \text{ and } P'\} C \{Q \text{ and } Q'\}}$$

Or

$$\frac{\{P\} C \{Q\}, \{P'\} C \{Q'\}}{\{P \text{ or } P'\} C \{Q \text{ or } Q'\}}$$

Figure 11.6: Axiomatic Semantics for Pelican (Part 2)

Exercises

1. Prove that the following two program fragments are semantically equivalent, assuming the declaration of the procedure increment given in this section.

read x;
write x

read x;
increment(-4);
increment(1);
increment(3);
write x

2. Give an example where an invalid assertion can be proved if we allow duplicate identifiers to occur at different levels of scope.

3. Prove that the following procedure copies all nonzero values from the input file to the output file up to, but not including, the first zero value.

```
procedure copy is
  var n : integer;
  begin
    read n; if n  $\neq$  0 then write n; copy end if
  end
```

4. Prove that the procedure “power” raises a to the power specified by the parameter value and leaves the result in the global variable product.

```
procedure power(b: integer) is
  begin
    if b = 0 then product := 1
      else power(b - 1); product := product * a
    end if
  end
```

5. Prove the partial correctness of this program relative to its specification.

```
{ B  $\geq$  0 }
program multiply is
  var m,n : integer;
  procedure incrementm(x : integer) is
    begin m := m+x end;
  begin
    m := 0; n := B;
    while n>0 do
      incrementm(A); n := n - 1
    end while
  end
{ m = A•B }
```

6. Consider the following procedure:

```
procedure outputsequence(n: integer) is
  begin
    if n > 0 then write n; outputsequence(n-1) end if
  end
```

Prove that

```
{val = A  $\geq$  0 and OUT = [ ]}
  outputsequence(val)
{OUT = [A, A-1, A-2, ... , 2, 1]}
```

7. Modify outputsequence in problem 6 so that it outputs values from 1 up to A. Prove the procedure correct.

8. Prove the partial correctness of the following Pelican program:

```

{  $k \geq 0$  and  $IN = [k]$  and  $OUT = []$  }
program recurrence is
  var num,ans : integer;
  procedure fun(m : integer) is
    var temp : integer;
  begin
    if m = 0
      then ans := 1
      else temp := 2*m+1; fun(m-1); ans := ans + temp
    end if
  end;
begin
  read num; fun(num); write ans
end
{  $OUT = [(k+1)^2]$  }

```

9. Illustrate the need for the transformation of procedures with a parameter that is changed in the body of the procedure by proving the spurious “correctness” of the following code using the Call₁ rule:

```

{  $OUT = []$  }
program problem1 is
  var a : integer;
  procedure p (b : integer) is
    begin b := 5 end;
  begin
    a := 21; p(a); write a
  end
{  $OUT = [5]$  }

```

10. Justify the need for the transformation of a one parameter procedure that makes a nonlocal change in the actual parameter by proving the spurious “correctness” of the following code using the Call₁ rule:

```

{  $OUT = []$  }
program problem2 is
  var m : integer;
  procedure q (f : integer) is
    begin m := 8 end;
  begin
    m := 55; q(m); write m
  end
{  $OUT = [55]$  }

```

11. Show what modifications will have to be made to the axiomatic definitions of Pelican to allow for procedures with several value parameters.

11.4 PROVING TERMINATION

In the proofs studied so far, we have considered only partial correctness, which means that the program must satisfy the specified assertions *only if* it ever halts, reaching the final assertion. The question of termination is frequently handled as a separate problem.

Termination is not an issue with many commands, such as assignment, selection, input/output, and nonrecursive procedure invocation. That these commands must terminate is contained in their semantics. Two language constructs require proofs of termination:

- Indefinite iteration (**while**)
- Invocation of a recursively defined procedure

The first case can be handled as a consequence of (well-founded) induction on an expression that is computed each pass through the loop, and the second can be managed by induction on some property possessed by each recursive call of the procedure.

Definition: A partial order $>$ or \geq on a set W is **well-founded** if there exists no infinite decreasing sequence of distinct elements from W . ■

This means that given a sequence of elements $\{x_i \mid i \geq 1\}$ from W such that $x_1 \geq x_2 \geq x_3 \geq x_4 \geq \dots$, there must exist an integer k such that $\forall i, j \geq k, x_i = x_j$.

If the partial order is **strict**, meaning that it is irreflexive, any decreasing sequence must have only distinct elements and so must be finite.

Examples of Well-founded Orderings

1. The natural numbers N ordered by $>$.
2. The Cartesian product $N \times N$ ordered by a lexicographic ordering defined as: $\langle m_1, m_2 \rangle > \langle n_1, n_2 \rangle$ if $([m_1 > n_1] \text{ or } [m_1 = n_1 \text{ and } m_2 > n_2])$.
3. The positive integers, P , ordered by the relation “properly divides”: $m > n$ if $(\exists k [m = n \cdot k] \text{ and } m \neq n)$.

Steps in Showing Termination

With indefinite iteration, termination is established by showing two steps:

1. Find a set W with a strict well-founded ordering $>$.
2. Find a **termination expression** E with the following properties:
 - a) Whenever control passes through the beginning of the iterative loop, the value of E is in W .
 - b) E takes a smaller value with respect to $>$ each time the top of the iterative loop is passed.

In the context of a **while** command—for example, “**while** B **do** C **end while**” with invariant P —the two conditions take the following form:

- a) $P \supset E \in W$
- b) $\{P \text{ and } B \text{ and } E=A\} C \{A > E\}$.

Example: Consider the following program that calculates the factorial of a natural number:

```

read n;
k := 0; f := 1;
while k < n do
  k := k + 1; f := k * f
end while;
write f

```

Take $W = \mathbb{N}$, the set of natural numbers, as the well-founded set and $E = n - k$ as the termination expression. Therefore, $m \in W$ if and only if $m \geq 0$. The loop invariant P is

$$(n \geq 0 \text{ and } k \leq n \text{ and } f = k! \text{ and } OUT = []).$$

The conditions on the termination expression must hold at the top of the **while** loop where the invariant holds.

The two conditions follow immediately:

- a) $(n \geq 0 \text{ and } k \leq n \text{ and } f = k! \text{ and } OUT = []) \supset (n - k \geq 0)$
- b) $\{n \geq 0 \text{ and } k \leq n \text{ and } f = k! \text{ and } OUT = [] \text{ and } k < n \text{ and } n - k = A\} \supset$
 $\{n - (k + 1) = A - 1\}$
 $\quad k := k + 1; f := k * f$
 $\{n - k = A - 1 < A\}$ ■

Example: As another example, consider the program with nested loops from section 11.2.

```

read x;
m := 0; n := 0; s := 0;
while x > 0 do
  x := x-1; n := m+2; m := m+1;
  while m > 0 do
    m := m-1; s := s+1
  end while;
  m := n
end while;
write s

```

With nested loops, each loop needs its own termination expression. In this example, they share the natural numbers as the well-founded set. The termination expressions can be defined as follows:

- For the outer loop: $E_o = x$
- For the inner loop: $E_i = m$

The code below shows the loop invariants used to verify that the termination expressions are adequate.

```

read x;
m := 0; n := 0; s := 0;
while x>0 do           {  $x \geq 0$  and  $m=2(A-x)$  and  $m=n \geq 0$  and  $s=(A-x)^2$  }
  x := x-1; n := m+2; m := m+1;
  while m>0 do         {  $x \geq 0$  and  $n=2(A-x)$  and  $m \geq 0$ 
    m := m-1; s := s+1           and  $n \geq 0$  and  $m+s=(A-x)^2$  }
  end while;
  m := n
end while;
write s

```

We leave the verification that the expressions E_o and E_i defined above satisfy the two conditions needed to prove termination as an exercise at the end of this section. ■

Note that the termination expression method described above depends on identifying some loop control “counter” that cannot change forever.

Termination of Recursive Procedures

A procedure defined recursively contains the seeds of an induction proof for termination, if only a suitable property about the problem can be identified on which to base the induction.

Example: Consider a Pelican procedure to read and write input values until the value zero is encountered.

```

procedure copy is
  var n: integer;
  begin
    read n;
    if n  $\neq$  0 then write n; copy end if
  end

```

This procedure terminates (normally) only if the input stream contains the value zero. For a particular invocation of the procedure “copy”, the depth of recursion depends on the number of nonzero integers preceding the first zero. We describe the input stream as $IN = L_1[0]L_2$ where L_1 contains no zero values.

Lemma: Given input of the form $IN = L_1[0]L_2$ where L_1 contains no zero values, the command “copy” halts.

Proof: By induction on the length of L_1 , $\text{length}(L_1)$.

Basis: $\text{length}(L_1)=0$.

Then the input list has the form $IN = [0]L_2$, and after “**read** n”, $n=0$.

Calling copy causes execution of only the code

```

read n;

```

which terminates.

Induction Step: $\text{length}(L_1)=k>0$.

As an induction hypothesis, assume that copy halts when $\text{length}(L_1)=k-1 \geq 0$. Then copy causes the execution of the code

```

read n;
write n;
copy

```

which terminates because for this inner copy, $\text{length}(L_1)=k-1$. ■

The complete proof of correctness of the procedure copy is left as an exercise.

Exercises

1. Formally prove that the factorial program in section 11.2 terminates. What happens to the termination proof if we remove the precondition $N \geq 0$?

2. Prove that the following program terminates. Also show partial correctness.

```

{ A ≠ 0 and B ≥ 0 }
  m := A; n := B; k := 1;
  while n > 0 do
    if 2*(n/2) = n
      then m := m*m; n := n/2
      else n := n-1; k := k*m
    end if
  end while
{ k = AB }

```

3. For the nested loop problem in this section, verify that the expressions E_0 and E_i satisfy the two conditions needed to prove termination.

4. Prove that the following program terminates. Also show partial correctness.

```

{ A ≥ 0 and B ≥ 0 and (A ≠ 0 or B ≠ 0) }
  m := A; n := B;
  while m > 0 do
    if m ≤ n then n := n-m
      else x := m; m := n; n := x
    end if
  end while
{ n is the greatest common divisor of A and B }

```

Verify each of the following termination expressions:

- $E_1 = \langle m, n \rangle$ with the lexicographic ordering on $N \times N$.
- $E_2 = 2m + n$ with the “greater than” ordering on N .

5. Prove the termination of the prime number program at the end of section 11.2.

6. Prove the termination of the program fragments in exercise 10 of section 11.2.

11.5 INTRODUCTION TO PROGRAM DERIVATION

In the first three sections of this chapter we started with programs or procedures that were already written, added assertions to the programs, and proved the assertions to be correct. In this section we apply axiomatic semantics in a different way, starting with assertions that represent program specifications and then deriving a program to match the assertions.

Suppose that we want to build a table of squares where $T[k]$ contains k^2 . A straightforward approach is to compute $k*k$ for each k and store the values in the table. However, multiplicative operations are inherently inefficient compared with additive operations, so we ask if this table can be generated using addition only. Actually this problem is not difficult; an early Greek investigation of “square” numbers provides a solution. As indicated by the table below, each square is the sum of consecutive odd numbers.

Square	Summation
1	1
4	1 + 3
9	1 + 3 + 5
16	1 + 3 + 5 + 7
25	1 + 3 + 5 + 7 + 9

The algorithm follows directly.

Table of Cubes

We now propose a slight variation of this problem: Construct a table of cubes using only additive methods. Given the ease of the solution for the table of squares, it may seem that we can find the answer quickly with just a little thought by playing with the numbers, but this problem turns out to be non-trivial. During a SIGCSE tutorial, David Gries reported that he assigned this problem to an advanced class in computer science and, even given several weeks, no one was able to come up with a correct solution. However, a solution can be produced directly if the techniques of program derivation are used.

We start with the final assertion that expresses the result of our program:

$$\{ T[k] = k^3 \text{ for all } 0 \leq k \leq N \}.$$

We build the table from the zeroth entry through the N^{th} entry, so for any particular value $m \leq N+1$, we know that all preceding table entries have been generated. This property becomes part of the loop invariant:

$$\{ T[k] = k^3 \text{ for all } 0 \leq k < m \}.$$

The value of m will increase until it reaches $N+1$, at which time the loop terminates. This condition gives us the other part of the loop invariant:

$$\{ 0 \leq m \leq N+1 \}.$$

We now have enough information to begin writing the program, starting with a skeleton describing the structure of the program.

```

m := 0;
while m < N + 1 do      { T[k] = k3 for all 0 ≤ k < m and 0 ≤ m ≤ N+1 }
  T[m] := ???
  :
  m := m + 1
end while
{ T[k] = k3 for all 0 ≤ k ≤ N }.

```

We introduce a new variable x whose value is assigned to $T[m]$ each time the loop executes, adding to our loop invariant the requirement that $x = m^3$. Since x can only be changed by addition, we introduce another variable y and the assignment command $x := x + y$. The new value of x in the next iteration has to be $(m+1)^3$, so we have

$$x + y = (m+1)^3 = m^3 + 3m^2 + 3m + 1.$$

But we already have in our loop invariant the requirement that $x = m^3$, so this means that $y = 3m^2 + 3m + 1$ must be added to the loop invariant. Since m is initially zero, this means the initial values for x and y are 0 and 1, respectively. Here is the derived program so far.

```

m := 0;
x := 0;
y := 1;
while m < N + 1 do      { T[k] = k3 for all 0 ≤ k < m and 0 ≤ m ≤ N+1
                          and x = m3 and y = 3m2 + 3m + 1 }
  T[m] := x;
  x := x + y;
  :
  m := m + 1
end while
{ T[k] = k3 for all 0 ≤ k ≤ N }.

```

The variable y can change only by addition, so we introduce a new variable z and the assignment $y := y + z$. The next time through the loop, m is incremented by one so that value of y must become

$$3(m + 1)^2 + 3(m + 1) + 1 = 3m^2 + 9m + 7.$$

But this new value equals $y + z$, so

$$y + z = 3m^2 + 9m + 7.$$

If we subtract the invariant $y = 3m^2 + 3m + 1$ from this equation, we end up with the requirement

$$z = 6m + 6,$$

which is added to the invariant. This relationship also means that z must be initialized to 6. So the code now becomes

```

m := 0;
x := 0;
y := 1;
z := 6;
while m <> N + 1 do      { T[k] = k3 for all 0 ≤ k < m and 0 ≤ m ≤ N+1
                             and x = m3
                             and y = 3m2 + 3m + 1
                             and z = 6m + 6 }
    T[m] := x;
    x := x + y;
    y := y + z;
    :
    m := m + 1
end while
{ T[k] = k3 for all 0 ≤ k ≤ N }.

```

The next time through the loop, the new value of z must equal

$$6(m + 1) + 6 = 6m + 6 + 6 = (\text{old value of } z) + 6.$$

This equality tells us that z must be incremented by 6 each time through the loop, and therefore the computation meets the requirement of consisting of additive operations. So now we have the complete program.

```

m := 0;
x := 0;
y := 1;
z := 6;
while m < N + 1 do      { T[k] = k3 for all 0 ≤ k < m and 0 ≤ m ≤ N+1
                             and x = m3
                             and y = 3m2 + 3m + 1
                             and z = 6m + 6 }
    T[m] := x;
    x := x + y;
    y := y + z;
    z := z + 6;
    m := m + 1
end while
{ T[k] = k3 for all 0 ≤ k ≤ N }

```

In the event that this formal derivation does not offer convincing enough proof that the above program works as expected, we present a small table of values following the algorithm.

m	x	y	z
0	0	1	6
1	1	7	12
2	8	19	18
3	27	37	24
4	64	61	30

Binary Search

The example above illustrates the technique of program derivation to produce a simple program, but it is tempting to ask if program derivation techniques can generate “really useful” programs. We conclude this section with the derivation of a binary search algorithm, an algorithm commonly presented in the study of data structures. We assume the following precondition for the sorted array A:

*{ A[0..N] is a sequence of integers such that
 $A[i] \leq A[i+1]$ for all $0 \leq i < N$ and x is an integer and $A[0] \leq x < A[N]$ }.*

We want to determine if there exists at least one i such that $0 \leq i < N$ and $x = A[i]$. However, x may not be present so we cannot require $x = A[i]$ as part of the postcondition. Specifying the postcondition takes some insight. Notice that the precondition specifies that x be contained in the interval $[A[0], A[N])$, where $[m,n)$ indicates an interval defined by the set $\{ k \mid m \leq k < n \}$. The basic idea will be to narrow that interval continually until it contains only a single element. We specify this by using indices i and j for the interval limits and requiring that ultimately $j = i + 1$. So the postcondition is

{ $A[i] \leq x < A[j]$ and $j = i + 1$ }.

The test determining whether $A[i] = x$ is made independently of this algorithm, but we will be able to guarantee that if $A[i] \neq x$ then x is not present anywhere in $[A[0], A[N])$.

The basic idea of the algorithm is that the subinterval $[A[i], A[j])$ becomes smaller and smaller, yet always contains x , until the postcondition is satisfied. We can now start construction of our program based on the following observations:

- The loop invariant is $A[i] \leq x < A[j]$.
- The loop will repeat until $j = i+1$, so the loop entry condition is $j \neq i+1$.

- The loop invariant is implied by the precondition if we set i to 0 and j to N .

Here is the initial program framework:

```

{ A[0..N] is a sequence of integers such that
  A[i] ≤ A[i+1] for all 0 ≤ i < N and x is an integer and A[0] ≤ x < A[N] }
i := 0;
j := N;
while j ≠ i + 1 do { A[i] ≤ x < A[j] }
  : : :
end while
{ A[i] ≤ x < A[j] and j = i + 1 }

```

We make the interval $[A[i], A[j])$ shrink by either increasing i or decreasing j . Suppose that we divide the interval “in half” by introducing the variable $k = (i + j)/2$, using integer division. Now x either lies in the interval $[A[i], A[k])$ or the interval $[A[k], A[j])$. It should be pointed out that x might lie in both intervals if A contains duplicate copies of x . However, in this case it does not matter which subinterval is chosen since both satisfy the loop invariant, and our algorithm requires only that we find one index, even though several may exist. If $x < A[k]$, then setting j to k maintains the loop invariant. Otherwise $A[k] ≤ x$ and setting i to k maintains the loop invariant. Here is the completed algorithm.

```

{ A[0..N] is a sequence of integers such that
  A[i] ≤ A[i+1] for all 0 ≤ i < N and x is an integer and A[0] ≤ x < A[N] }
i := 0;
j := N;
while j ≠ i + 1 do          { A[i] ≤ x < A[j] }
  k := (i + j) / 2;
  if x < A[k] then j := k
    else i := k
  end if
end while
{ A[i] ≤ x < A[j] and j = i + 1 }

```

Exercises

1. Derive a program that constructs a table with $T[k] = k^4$, using only additive methods. This is similar to the table of cubes example except that four new variables have to be introduced with four assignment commands changing the values of these variables by addition.

2. Suppose that N is a fixed integer greater than or equal to zero (so the precondition is $\{N \geq 0\}$). Derive a program to find the integer square root of N . The integer square root is the largest integer that is less than or equal to the square root of N . This can be expressed as the postcondition:

$$\{ a \geq 0 \text{ and } 0 \leq a^2 \text{ and } a^2 \leq N \text{ and } N < (a + 1)^2 \}$$

Hint: Use two variables, a and b , initialized to 0 and $N+1$, respectively. As in the binary search problem, find the midpoint of a and b and change one of the values until the desired subinterval is found.

11.6 FURTHER READING

The original idea of verifying the correctness of a program using the techniques of logic first appears in papers by Robert Floyd [Floyd67] and C. A. R. Hoare [Hoare69]. These papers still serve as excellent introductions to axiomatic semantics. An early application of this method to programming language specification can be found in the definition of Pascal in [Hoare73].

The books dealing with the analysis of programs and languages in the framework of the predicate logic can be divided into two groups:

- Books that develop axiomatic methods primarily to prove the correctness of programs as a tool of software engineering [Alagic78], [Backhouse86], [Francez92], [Gries81], and [Gumb89]. These authors concentrate on describing techniques of program construction and verification based on the predicate logic. The discussion of program derivation in section 11.5 falls into this classification. A book on program derivation by Geoff Dromey [Dromey89] gives numerous examples of this approach to program construction. The related method of “weakest precondition” is discussed in [Dijkstra76].
- Books that view axiomatic methods as a means of programming language definition [Meyer90], [Nielson92], [Pagan81], [Tennent91], and [Winskel93]. Although correctness is discussed in these books, the emphasis is on using logic to specify the semantics of programming languages in a manner similar to the presentation of Wren and Pelican in this chapter.

For a review of predicate logic see [Enderton72], [Mendelson79], or [Reeves90].