7.1.2 Well-Formed Formulas

To give a precise description of a first-order predicate calculus, we need an alphabet of symbols. For this discussion we'll use several kinds of letters and symbols, described as follows:

Individual variables:	x, y, z
Individual constants:	a, b, c
Function constants:	f, g, h
Predicate constants:	p, q, r
Connective symbols:	$\neg$, $\rightarrow$, $\wedge$, $\vee$
Quantifier symbols:	$\exists, \forall$
Punctuation symbols:	(,)

From time to time we will use other letters, or strings of letters, to denote variables or constants. We'll also allow letters to be subscripted. The number of arguments for a predicate or function will normally be clear from the context. A predicate with no arguments is considered to be a proposition.

A *term* is either a variable, a constant, or a function applied to arguments that are terms. For example, x, a, and $f(x, g(b))$ are terms. An *atomic formula* (or simply *atom*) is a predicate applied to arguments that are terms. For example, $p(x, a)$ and $q(y, f(c))$ are atoms.

We can define the wffs—the well-formed formulas—of the first-order predicate calculus inductively as follows:

Definition of a Wff (Well-Formed Formula)

1. Any atom is a wff.
2. If W and V are wffs and x is a variable, then the following expressions are also wffs:

$$(W), \neg W, W \vee \mathrm{V}, W \wedge V, W \rightarrow V, \exists x\ W, \text{ and } \forall x\ W.$$

To write formulas without too many parentheses and still maintain a unique meaning, we'll agree that the quantifiers have the same precedence as the negation symbol. We'll continue to use the same hierarchy of precedence for the operators $\neg$, $\wedge$, $\vee$, and $\rightarrow$. Therefore, the hierarchy of precedence now looks like the following:

$\neg, \exists x, \forall y$ (highest, do first)
$\wedge$
$\vee$
(lowest, do last)

If any of the quantifiers or the negation symbol appear next to each other, then the rightmost symbol is grouped with the smallest wff to its right. Here are a few wffs in both unparenthesized form and parenthesized form:

Unparenthesized Form	*Parenthesized Form*
$\forall x \neg \exists y \, \forall z \, p(x, y, z)$	$\forall x \, (\neg \, (\exists y \, (\forall z \, p(x, y, z))))$.
$\exists x \, p(x) \vee q(x)$	$(\exists x \, p(x)) \vee q(x)$.
$\forall x \, p(x) \rightarrow q(x)$	$(\forall x \, p(x)) \rightarrow q(x)$.
$\exists x \neg p(x, y) \rightarrow q(x) \wedge r(y)$	$(\exists x \, (\neg \, p(x, y))) \rightarrow (q(x) \wedge r(y))$.
$\exists x \, p(x) \rightarrow \forall x \, q(x) \vee p(x) \wedge r(x)$	$(\exists x \, p(x)) \rightarrow ((\forall x \, q(x)) \vee (p(x) \wedge r(x)))$.

Scope, Bound, and Free

Now let's discuss the relationship between the quantifiers and the variables that appear in a wff. When a quantifier occurs in a wff, it influences some occurrences of the quantified variable. The extent of this influence is called the scope of the quantifier, which we define as follows:

Definition of Scope

In the wff $\exists x \; W$, W is the *scope* of the quantifier $\exists x$.

In the wff $\forall x \; W$, W is the *scope* of the quantifier $\forall x$.

For example, the scope of $\exists x$ in the wff

$$\exists x \, p(x, y) \rightarrow q(x)$$

is $p(x, y)$ because the parenthesized version of the wff is $(\exists x \, p(x, y)) \rightarrow q(x)$. On the other hand, the scope of $\exists x$ in the wff

$$\exists x \, (p(x, y) \rightarrow q(x))$$

is the conditional $p(x, y) \rightarrow q(x)$. Now let's classify the occurrences of variables that occur in a wff.

Bound and Free Variables

An occurrence of a variable x in a wff is said to be *bound* if it lies within the scope of either $\exists x$ or $\forall x$ or if it is the quantifier variable x itself. Otherwise, an occurrence of x is said to be *free* in the wff.

For example, consider the following wff:

$$\exists x\, p(x, y) \rightarrow q(x).$$

The first two occurrences of x are bound because the scope of $\exists x$ is $p(x, y)$. The only occurrence of y is free, and the third occurrence of x is free.

So every occurrence of a variable in a wff can be classified as either bound or free, and this classification is determined by the scope of the quantifiers in the wff. Now we're in a position to discuss the meaning of wffs.

7.1.3 Semantics and Interpretations

Up to this point a wff is just a string of symbols with no meaning attached. For a wff to have a meaning, we must give an interpretation to its symbols so that the wff can be read as a statement that is true or false. For example, suppose we let $p(x)$ mean "x is an even integer" and we let x be the number 236. With this interpretation, $p(x)$ becomes the statement "236 is an even integer," which is true.

As another example, let's give an interpretation to the wff

$$\forall x\, \exists y\, s(x, y).$$

We'll let $s(x, y)$ mean that the successor of x is y, where the variables x and y take values from the set of natural numbers $\mathbb{N}$. With this interpretation the wff becomes the statement "For every natural number x there exists a natural number y such that the successor of x is y," which is true.

Before we proceed any further, we need to make a precise definition of the idea of an interpretation. Here's the definition in all its detail.

Definition of Interpretation

An *interpretation* for a wff consists of a nonempty set D, which is called the *domain* of the interpretation, together with an assignment that associates the symbols of the wff to values in D as follows:

1. Each predicate letter is assigned to a relation over D. A predicate with no arguments is a proposition and must be assigned a truth value.
2. Each function letter is assigned to a function over D.
3. Each free variable is assigned to a value in D. All free occurrences of a variable x are assigned to the same value in D.
4. Each constant is assigned to a value in D. All occurrences of the same constant are assigned to the same value in D.

6. Prove each of the following equivalences.
   a. $p(x) \equiv \exists y\ ((x = y) \wedge p(y))$.
   b. $p(x) \equiv \forall y\ ((x = y) \rightarrow p(y))$.

Formalizing English Sentences

7. Formalize the definition for each statement about the integers.
   a. odd(x) means x is odd.
   b. even(x) means x is even.
   c. div(a, b) means a divides b.
   d. $r = a \bmod b$.
   e. $d = \gcd(a, b)$.

8. Formalize each of the following statements.
   a. There is at most one x such that $A(x)$ is true.
   b. There are exactly two x and y such that that $A(x)$ and $A(y)$ are true.
   c. There are at most two x and y such that $A(x)$ and $A(y)$ are true.

9. Students were asked to formalize the statement "There is a unique x such that $A(x)$ is true." The following wffs were given as answers.
   a. $\exists x\ (A(x) \wedge \forall y\ (A(y) \rightarrow (x = y)))$.
   b. $\exists x\ A(x) \wedge \forall x\ \forall y\ (A(x) \wedge A(y) \rightarrow (x = y))$.

   Prove that wffs (a) and (b) are equivalent. *Hint:* Do two indirect proofs showing that each statement implies the other.

8.2 Program Correctness

An important and difficult problem of computer science can be stated as

"Prove that a program is correct." (8.9)

This takes some discussion. One major question to ask before we can prove that a program is correct is "What is the program supposed to do?" If we can state in English what a program is supposed to do, and English is the programming language, then the statement of the problem may itself be a proof of its correctness.

Normally, a problem is stated in some language X, and its solution is given in some language Y. For example, the statement of the problem might use English mixed with some symbolic notation, while the solution might be in a programming language. How do we prove correctness in cases like this? Often the answer depends on the programming language. As an example, we'll look at a formal theory for proving the correctness of imperative programs.

8.2.1 Imperative Program Correctness

An *imperative program* consists of a sequence of statements that represent commands. The most important statement is the assignment statement. Other statements are used for control, such as looping and taking alternate paths. To prove things about such programs, we need a formal theory consisting of wffs, axioms, and inference rules.

Suppose we want to prove that a program does some particular thing. We must represent the thing that we want to prove in terms of a precondition P, which states what is supposed to be true before the program starts, and a postcondition Q, which states what is supposed to be true after the program halts. If S denotes the program, then we will describe this informal situation with the following wff, which is called a *Hoare triple:*

$$\{P\}\ S\ \{Q\}.$$

The letters P and Q denote logical statements that describe properties of the variables that occur in S. P is called a *precondition* for S, and Q is called a *postcondition* for S. We assume that P and Q are wffs from a first-order theory with equality that depends on the program S. For example, if the program manipulates numbers, then the first-order theory must include the numerical operations and properties that are required to describe the problem at hand. If the program processes strings, then the first-order theory must include the string operations.

For example, suppose S is the single assignment statement $x := x + 1$. Then the following expression is a wff in our logic:

$$\{x > 4\}\ x := x + 1\ \{x > 5\}.$$

To have a logic for program correctness, we must have a meaning assigned to each wff of the form $\{P\}\ S\ \{Q\}$. In other words, we need to assign a truth value to each wff of the form $\{P\}\ S\ \{Q\}$.

The Meaning of $\{P\}\ S\ \{Q\}$

The meaning of $\{P\}\ S\ \{Q\}$ is the truth value of the following statement:

> If P is true before S is executed and the execution of S terminates, then Q is true after the execution of S.

If $\{P\}\ S\ \{Q\}$ is true, we say S is *correct* with respect to precondition P and postcondition Q. Strictly speaking, we should say that S is *partially correct* because the truth of $\{P\}\ S\ \{Q\}$ is based on the assumption that S terminates. If we also know that S terminates, then we say S is *totally correct.* We'll discuss termination at the end of the section.

Sometimes it's easy to observe whether $\{P\}\ S\ \{Q\}$ is true. For example, from our knowledge of the assignment statement, most of us will agree that the following wff is true:

$$\{x > 4\}\ x := x + 1\ \{x > 5\}.$$

On the other hand, most of us will also agree that the following wff is false:

$$\{x > 4\}\ x := x + 1\ \{x > 6\}.$$

But we need some proof methods to verify our intuition. A formal theory for proving the correctness of programs needs some axioms and some inference rules. So let's start.

The Assignment Axiom

The axioms depend on the types of assignments allowed by the assignment statement. The inference rules depend on the control structures of the language. So we had better agree on a language before we go any further in our discussion. To keep things simple, we'll assume that the assignment statement has the following form, where x is a variable and t is a term:

$$x := t.$$

So the only thing we can do is assign a value to a variable. This effectively restricts the language so that it cannot use other structures, such as arrays and records. In other words, we can't make assignments like $a[i] := t$ or $a.b := t$.

Since our assignment statement is restricted to the form $x := t$, we need only one axiom. It's called the *assignment axiom*, and we'll motivate the discovery of the axiom by an example. Suppose we're told that the following wff is correct:

$$\{P\}\ x := 4\ \{y > x\}.$$

In other words, if P is true before the execution of the assignment statement, then after its execution the statement $y > x$ is true. What should P be? From our knowledge of the assignment statement we might guess that P has the following definition:

$$P = (y > 4).$$

This is about the most general statement we can make. Notice that P can be obtained from the postcondition $y > x$ by replacing x by 4. The assignment axiom generalizes this idea in the following way.

Assignment Axiom (AA) **(8.10)**

$$\{Q(x/t)\}\ x := t\ \{Q\}.$$

The notation $Q(x/t)$ denotes the wff obtained from Q by replacing all free occurrences of x by t. The axiom is often called the "backwards" assignment axiom because the precondition is constructed from the postcondition.

Let's see how the assignment axiom works in a backwards manner. When using AA, always start by writing down the form of (8.10) with an empty precondition as follows:

$$\{\quad\}\ x := t\ \{Q\}.$$

Now the task is to construct the precondition by replacing all free occurrences of x in Q by t.

For example, suppose we know that $x < 5$ is the postcondition for the assignment statement $x := x + 1$. We start by writing down the following partially completed version of AA:

$$\{\quad\}\ x := x + 1\ \{x < 5\}.$$

Then we use AA to construct the precondition. In this case we replace the x by $x + 1$ in the postcondition $x < 5$. This gives us the precondition $x + 1 < 5$, and we can write down the completed instance of the assignment axiom:

$$\{x + 1 < 5\}\ x := x + 1\ \{x < 5\}.$$

The Consequence Rule

It happens quite often that the precondition constructed by AA doesn't quite match what we're looking for. For example, most of us will agree that the following wff is correct.

$$\{x < 3\}\ x := x + 1\ \{x < 5\}.$$

But we've already seen that AA applied to this assignment statement gives

$$\{x + 1 < 5\}\ x := x + 1\ \{x < 5\}.$$

Since the two preconditions don't match, we have some more work to do. In this case we know that for any number x we have $(x < 3) \rightarrow (x + 1 < 5)$.

Let's see why this is enough to prove that $\{x < 3\}\ x := x + 1\ \{x < 5\}$ is correct. If $x < 3$ is true before the execution of $x := x + 1$, then we also know that $x + 1 < 5$ is true before execution of $x := x + 1$. Now AA tells us that $x < 5$ is true after execution of $x := x + 1$. So $\{x < 3\}\ x := x + 1\ \{x < 5\}$ is correct.

This kind of argument happens so often that we have an inference rule to describe the situation for any program S. It's called the *consequence rule*:

Consequence Rules **(8.11)**

$$\frac{P \rightarrow R \text{ and } \{R\}\,S\,\{Q\}}{\therefore \{P\}\,S\,\{Q\}} \quad \text{and} \quad \frac{\{P\}\,S\,\{T\} \text{ and } T \rightarrow Q,}{\therefore \{P\}\,S\,\{Q\}}.$$

Notice that each consequence rule requires two proofs: a proof of correctness and a proof of an implication. Let's do an example.

example 8.5 Using the Assignment Axiom and the Consequence Rule

We'll prove the correctness of the following wff:

$$\{x < 5\}\ x := x + 1\ \{x < 7\}.$$

To start things off, we'll apply (8.10) to the assignment statement and the post-condition to obtain the following wff:

$$\{x + 1 < 7\}\ x := x + 1\ \{x < 7\}.$$

This isn't what we want. We got the precondition $x + 1 < 7$, but we need the precondition $x < 5$. Let's see whether we can apply (8.11) to the problem. In other words, let's see whether we can prove the following statement:

$$(x < 5) \rightarrow (x + 1 < 7).$$

This statement is certainly true, and we'll include its proof in the following formal proof of correctness of the original wff.

Proof:			
	1.	$\{x + 1 < 7\}\ x := x + 1\ \{x < 7\}$	AA
	2.	$x < 5$	P
	3.	$x + 1 < 6$	2, T
	4.	$6 < 7$	T
	5.	$x + 1 < 7$	3, 4, Transitive
	6.	$(x < 5) \rightarrow (x + 1 < 7)$	2, 5, CP
		QED	1, 6, Consequence.

end example

Although assignment statements are the core of imperative programming, we can't do much programming without control structures. So let's look at a few fundamental control structures together with their corresponding inference rules.

The Composition Rule

The most basic control structure is the composition of two statements S_1 and S_2, which we denote by S_1; S_2. This means execute S_1 and then execute S_2. The *composition rule* can be used to prove the correctness of the composition of two statements.

Composition Rule **(8.12)**

$$\frac{\{P\}\, S_1\, \{R\} \text{ and } \{R\}\, S_2\, \{Q\}}{\therefore \{P\}\, S_1; S_2\, \{Q\}}.$$

The composition rule extends naturally to any number of program statements in a sequence. For example, suppose we prove that the following three wffs are correct.

$$\{P\}\ S_1\ \{R\}, \quad \{R\}\ S_2\ \{T\}, \quad \{T\}\ S_3\ \{Q\}.$$

Then we can infer that $\{P\}\ S_1;\ S_2;\ S_3\ \{Q\}$ is correct.

For (8.12) to work, we need an intermediate condition R to place between the two statements. Intermediate conditions often appear naturally during a proof, as the next example shows.

example 8.6 Using the Composition Rule

We'll show the correctness of the following wff:

$$\{(x > 2) \wedge (y > 3)\}\ x := x + 1;\ y := y + x\ \{y > 6\}.$$

This wff matches the bottom of the composition inference rule (8.12). Since the program statements are assignments, we can use the AA rule to move backward from the postcondition to find an intermediate condition to place between the two assignments. Then we can use AA again to move backward from the intermediate condition. Here's the proof.

Proof: First we'll use AA to work backward from the postcondition through the second assignment statement:

1. $\{y + x > 6\}\ y := y + x\ \{y > 6\}$ AA

Now we can take the new precondition and use AA to work backward from it through the first assignment statement:

2. $\{y + x + 1 > 6\}\ x := x + 1\ \{y + x > 6\}$ AA

Now we can use the composition rule (8.12) together with lines 1 and 2 to obtain line 3 as follows:

3. $\{y + x + 1 > 6\}\ x := x + 1;\ y := y + x\ \{y > 6\}$ 1, 2, Comp

At this point the precondition on line 3 does not match the precondition for the wff that we are trying to prove correct. Let's try to apply the consequence rule (8.11) to the situation.

4.	$(x > 2) \wedge (y > 3)$	P
5.	$x > 2$	4, Simp
6.	$y > 3$	4, Simp
7.	$x + y > 2 + y$	5, T
8.	$2 + y > 2 + 3$	6, T
9.	$x + y > 2 + 3$	7, 8, Transitive
10.	$x + y + 1 > 6$	9, T
11.	$(x > 2) \wedge (y > 3) \rightarrow (x + y + 1 > 6)$	4, 10, CP

Now we're in position to apply the consequence rule to lines 3 and 11:

12. $\{(x > 2) \wedge (y > 3)\}\ x := x + 1;\ y := y + x\ \{y > 6\}$
    3, 11, Consequence.

QED

end example

The If-Then Rule

The statement **if** C **then** S means that S is executed if C is true and S is bypassed if C is false. For statements of this form we have the following *if-then rule* of inference.

If-Then Rule (8.13)

$$\frac{\{P \wedge C\}\, S\, \{Q\} \text{ and } P \wedge \neg\, C \rightarrow Q}{\therefore \{P\}\ \textbf{if}\ C\ \textbf{then}\ S\, \{Q\}}.$$

The two wffs in the hypothesis of (8.13) are of different type. The logical wff $P \wedge \neg\, C \rightarrow Q$ needs a proof from the predicate calculus. This wff is necessary in the hypothesis of (8.13) because if C is false, then S does not execute. But we still need Q to be true after C has been determined to be false during the execution of the if-then statement. Let's do an example.

example **8.7 Using the If-Then Rule**

We'll show that the following wff is correct:

$$\{\text{true}\}\ \textbf{if}\ x < 0\ \textbf{then}\ x := -\,x\ \{x \geq 0\}.$$

Proof: Since the wff fits the pattern of (8.13), all we need to do is prove the following two statements:

1. $\{\text{true} \wedge (x < 0)\}\ x := -\,x\ \{x \geq 0\}$.

The proofs are easy. We'll combine them into one formal proof:

Proof:	1.	$\{-x \geq 0\}\ x := -x\ \{x \geq 0\}$	AA
	2.	$\text{true} \wedge (x < 0)$	P
	3.	$x < 0$	2, Simp
	4.	$-x > 0$	3, T
	5.	$-x \geq 0$	4, Add
	6.	$\text{true} \wedge (x < 0) \rightarrow (-x \geq 0)$	2, 5, CP
	7.	$\{\text{true} \wedge (x < 0)\}\ x := -x\ \{x \geq 0\}$	1, 6, Consequence
	8.	$\text{true} \wedge \neg (x < 0)$	P
	9.	$\neg (x < 0)$	8, Simp
	10.	$x \geq 0$	9, T
	11.	$\text{true} \wedge \neg (x < 0) \rightarrow (x \geq 0)$	8, 10, CP
		QED	7, 11, If-then.

end example

The If-Then-Else Rule

The statement **if** C **then** S_1 **else** S_2 means that S_1 is executed if C is true and S_2 is executed if C is false. For statements of this form we have the following *if-then-else rule* of inference.

If-Then-Else Rule **(8.14)**

$$\frac{\{P \wedge C\}\, S_1\, \{Q\} \text{ and } \{P \wedge \neg\, C\}\, S_2\, \{Q\}}{\therefore \{P\}\ \textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\, \{Q\}}.$$

example **8.8 Using the If-Then-Else Rule**

Suppose we're given the following wff, where even(x) means that x is an even integer:

$$\{\text{true}\}\ \textbf{if}\ \text{even}(x)\ \textbf{then}\ y := x\ \textbf{else}\ y := x + 1\ \{\text{even}(y)\}.$$

We'll give a formal proof that this wff is correct. The wff matches the bottom of rule (8.14). Therefore, the wff will be correct by (8.14) if we can show that the following two wffs are correct:

1. $\{\text{true} \wedge \text{even}(x)\}\ y := x\ \{\text{even}(y)\}$.
2. $\{\text{true} \wedge \text{odd}(x)\}\ y := x + 1\ \{\text{even}(y)\}$.

To make the proof formal, we need to give formal descriptions of even(x) and odd(x). This is easy to do over the domain of integers.

$$\text{even}(x) = \exists k\,(x = 2k)\,,$$
$$\text{odd}(x) = \exists k\,(x = 2k + 1)\,.$$

To avoid clutter, we'll use even(x) and odd(x) in place of the formal expressions. If you want to see why a particular line holds, you might make the substitution for even or odd and then see whether the statement makes sense. We'll combine the two proofs into the following formal proof:

Proof:	1.	$\{\text{even}(x)\}\ y := x\ \{\text{even}(y)\}$	AA
	2.	$\text{true} \wedge \text{even}(x)$	P
	3.	$\text{even}(x)$	2, Simp
	4.	$\text{true} \wedge \text{even}(x) \rightarrow \text{even}(x)$	2, 3, CP
	5.	$\{\text{true} \wedge \text{even}(x)\}\ y := x\ \{\text{even}(y)\}$	1, 4, Consequence
	6.	$\{\text{even}(x + 1)\}\ y := x + 1\ \{\text{even}(y)\}$	AA
	7.	$\text{true} \wedge \text{odd}(x)$	P
	8.	$\text{odd}(x)$	7, Simp
	9.	$\text{even}(x + 1)$	8, T
	10.	$\text{true} \wedge \text{odd}(x) \rightarrow \text{even}(x + 1)$	7, 9, CP
	11.	$\{\text{true} \wedge \text{odd}(x)\}\ y := x + 1\ \{\text{even}(y)\}$	6, 10, Consequence
		QED	5, 11, If-then-else.

end example

The While Rule

The last inference rule that we will consider is the *while rule*. The statement **while** C **do** S means that S is executed if C is true, and if C is still true after S has executed, then the process is started over again. Since the body S may execute more than once, there must be a close connection between the precondition and postcondition for S. This can be seen by the appearance of P in all preconditions and postconditions of the rule.

While Rule (8.15)

$$\frac{\{P \wedge C\}\,S\,\{P\}}{\therefore \{P\}\ \textbf{while}\ C\ \textbf{do}\ \ S\,\{P \wedge \neg\, C\}}.$$

The wff P is called a *loop invariant* because it must be true before and after each execution of the body S. Loop invariants can be tough to find in

programs with no documentation. On the other hand, in writing a program, a loop invariant can be a helpful tool for specifying the actions of while loops.

To illustrate the idea of working with while loops, we'll work our way through an example that will force us to discover a loop invariant in order to prove the correctness of a wff. Suppose we want to prove the correctness of the following program to compute the power a^b of two natural numbers a and b, where $a > 0$ and $b \geq 0$:

$\{(a > 0) \wedge (b \geq 0)\}$
$i := 0;$
$p := 1;$
while $i < b$ **do**
  $p := p * a;$
  $i := i + 1$
od
$\{p = a^b\}$

The program consists of three statements. So we can represent the program and its precondition and postcondition in the following form:

$$\{(a > 0) \wedge (b \geq 0)\}\ S_1;\ S_2;\ S_3\ \{p = a^b\}.$$

In this form, S_1 and S_2 are the first two assignment statements, and S_3 represents the while statement. The composition rule (8.12) tells us that we can prove that the wff is correct if we can find proofs of the following three statements for some wffs P and Q.

$$\{(a > 0) \wedge (b \geq 0)\}\ S_1\ \{Q\},$$
$$\{Q\}\ S_2\ \{P\},$$
$$\{P\}\ S_3\ \{p = a^b\}.$$

Where do P and Q come from? If we know P, then we can use AA to work backward through S_2 to find Q. But how do we find P? Since S_3 is a while statement, P should be a loop invariant. So we need to do a little work.

From (8.15) we know that a loop invariant P for the while statement S_3 must satisfy the following form:

$$\{P\}\ \textbf{while}\ i < b\ \textbf{do}\ p := p * a;\ i := i + 1\ \textbf{od}\ \{P \wedge \neg\ (i < b)\}.$$

Let's try some possibilities for P. Suppose we set $P \wedge \neg\ (i \leq b)$ equivalent to the program's postcondition $p = a^b$ and try to solve for P. This won't work because $p = a^b$ does not contain the letter i. So we need to be more flexible in our thinking. Since we have the consequence rule, all we really need is an invariant P such that $P \wedge \neg\ (i < b)$ implies $p = a^b$.

After staring at the program, we might notice that the equation $p = a^i$ holds both before and after the execution of the two assignment statements in the body of the while statement. It's also easy to see that the inequality $i \leq b$ holds before and after the execution of the body. So let's try the following definition for P:

$$(p = a^i) \wedge (i \leq b).$$

This P has more promise. Notice that $P \wedge \neg (i < b)$ implies $i = b$, which gives us the desired postcondition $p = a^b$. Next, by working backward from P through the two assignment statements, we wind up with the statement

$$(1 = a^0) \wedge (0 \leq b).$$

This statement can certainly be derived from the precondition $(a \geq 0) \wedge (b > 0)$. So P does OK from the start of the program down to the beginning of the while loop. All that remains is to prove the following statement:

$$\{P\} \textbf{ while } i < b \textbf{ do } p := p * a;\ i := i + 1 \textbf{ od } \{P \wedge \neg (i < b)\}.$$

By (8.15), all we need to prove is the following statement:

$$\{P \wedge (i < b)\}\ p := p * a;\ i := i + 1\ \{P\}.$$

This can be done easily, working backward from P through the two assignment statements. We'll put everything together in the following example.

example 8.9 Using the While Rule

We'll prove the correctness of the following program to compute the power a^b of two natural numbers a and b, where $a > 0$ and $b \geq 0$:

$\{(a > 0) \wedge (b \geq 0)\}$
$i := 0;$
$p := 1;$
while $i < b$ **do**
  $p := p * a;$
  $i := i + 1$
od
$\{p = a^b\}$

We'll use the loop invariant $P = (p = a^i) \wedge (i \leq b)$ for the while statement. To keep things straight, we'll insert $\{P\}$ as the precondition for the while loop and

$\{(a > 0) \wedge (b \geq 0)\}$
$i := 0;$
$p := 1;$
$\{P\} = \{(p = a^i) \wedge (i \leq b)\}$
while $i < b$ **do**
  $p := p * a;$
  $i := i + 1$
od
$\{P \wedge \neg C\} = \{(p = a^i) \wedge (i \leq b) \wedge \neg (i < b)\}$
$\{p = a^b\}$

We'll start by proving that $P \wedge \neg C \rightarrow (p = a^b)$.

1.	$(p = a^i) \wedge (i \leq b) \wedge \neg (i < b)$	P
2.	$p = a^i$	1, Simp
3.	$(i \leq b) \wedge \neg (i < b)$	1, Simp
4.	$i = b$	3, T
5.	$p = a^b$	2, 4, EE
6.	$(p = a^i) \wedge (i \leq b) \wedge \neg (i < b) \rightarrow (p = a^b)$	1, 5, CP

Next, we'll prove the correctness of $\{P\}$ **while** $i < b$ **do** S $\{P \wedge \neg (i < b)\}$. The while inference rule tells us to prove the correctness of $\{P \wedge (i < b)\}$ S $\{P\}$.

7.	$\{(p = a^{i+1}) \wedge (i + 1 \leq b)\}$ $i := i + 1$ $\{(p = a^i) \wedge (i \leq b)\}$	AA
8.	$\{(p * a = a^{i+1}) \wedge (i + 1 \leq b)\}$ $p := p * a$ $\{(p = a^{i+1}) \wedge (i + 1 \leq b)\}$	AA
9.	$(p = a^i) \wedge (i \leq b) \wedge (i < b)$	P
10.	$p = a^i$	9, Simp
11.	$i < b$	9, Simp
12.	$b < i + 1$	P for IP
13.	$(i < b) \wedge (b < i + 1)$	11, 12, Conj
14.	false	13, T (for integers i and b)
15.	$i + 1 \leq b$	11, 14, IP
16.	$a = a$	EA
17.	$p * a = a^{i+1}$	10, 16, EE
18.	$(p * a = a^{i+1}) \wedge (i + 1 \leq b)$	15, 17, Conj
19.	$P \wedge (i < b) \rightarrow (p * a = a^{i+1}) \wedge (i + 1 \leq b)$	9, 18, CP
20.	$\{P \wedge (i < b)\}$ $p := p * a;$ $i := i + 1$ $\{P\}$	7, 8, 19, Conseq, Comp
21.	$\{P\}$ **while** $i < b$ **do** $p := p * a;$ $i := i + 1$ **od** $\{P \wedge \neg (i < b)$	20, While

Now let's work on the two assignment statements that begin the program. So we'll prove the correctness of $\{(a > 0) \wedge (b \geq 0)\}\ i := 0;\ p := 1\ \{P\}$.

22.	$\{(1 = a^i) \wedge (i \leq b)\}\ p := 1\ \{(p = a^i) \wedge (i \leq b)\}$	AA
23.	$\{(1 = a^0) \wedge (0 \leq b)\}\ i := 0\ \{(1 = a^i) \wedge (i \leq b)\}$	AA
24.	$(a > 0) \wedge (b \geq 0)$	P
25.	$a > 0$	24, Simp
26.	$b \geq 0$	24, Simp
27.	$1 = a^0$	25, T
28.	$(1 = a^0) \wedge (0 \leq b)$	26, 27, Conj
29.	$(a > 0) \wedge (b \geq 0) \rightarrow (1 = a^0) \wedge (0 \leq b)$	24, 28, CP
30.	$\{(a > 0) \wedge (b \geq 0)\}\ i := 0;\ p := 1\ \{P\}$	22, 23, 29, Comp, Conseq

The proof is finished by using the Composition and Consequence rules:

	QED	30, 21, 6, Comp, Conseq.

end example

8.2.2 Array Assignment

Since arrays are fundamental structures in imperative languages, we'll modify our theory so that we can handle assignment statements like $a[i] := t$. In other words, we want to be able to construct a precondition for the following partial wff:

$$\{\ \}\ a[i] := t\ \{Q\}.$$

What do we do? We might try to work backward, as with AA, and replace all occurrences of $a[i]$ in Q by t. Let's try it and see what happens. Let $Q(a[i]/t)$ denote the wff obtained from Q by replacing all occurrences of $a[i]$ by t. We'll call the following statement the "attempted" array assignment axiom:

Attempted AAA: **(8.16)**

$$\{Q\,(a\,[i]\,/t)\}\,a\,[i] := t\,\{Q\}\,.$$

Since we're calling (8.16) the Attempted AAA, let's see whether we can find something wrong with it. For example, suppose we have the following wff, where the letter i is a variable:

$$\{\text{true}\}\ a[i] := 4\ \{a[i] = 4\}.$$