12

Program Correctness

"Testing can show the presence of errors, but not their absence."
E. W. Dijkstra

CHAPTER OUTLINE

12.1 WHY CORRECTNESS?

Programming languages are powerful vehicles for designing and implementing complex software. Complex software systems are difficult to design well, and often the resulting system is full of errors. Much has been written about the need for better methodologies and tools for designing reliable software, and in recent years some of these tools have begun to show some promise.

It is appropriate in our study of modern programming languages to examine the question of language features that support the design of reliable software systems and how those features extend the expressive power of conventional languages. This chapter thus addresses the issue of program correctness from the important perspective of language features and programming paradigms.

A "correct" program is one that does exactly what its designers and users intend it to do – no more and no less. A "formally correct" program is one whose correctness can be proved mathematically, at least to a point that designers and users are convinced about its relative absence of errors.

For a program to be formally correct, there must be a way to specify precisely (mathematically) what the program is intended to do, for all possible values of its input. These so-called *specification languages* are based on mathematical logic, which we review in the next section. A programming language's specification language is based a concept called *axiomatic semantics*, which was first suggested by C.A.R. Hoare over three decades ago [Hoare 1969]. The use of axiomatic semantics for proving the correctness of small programs is introduced in the third section of this chapter.

Formally proving the correctness of a small program, of course, does not address the major problem facing software designers today. Modern software systems have millions of lines of code, representing thousands of semantic states and state transitions. This innate complexity requires that designers use robust tools for assuring that the system behaves properly in each of its states.

Until very recently, software modeling languages had been developed as separate tools, and were not fully integrated with popular compilers and languages used by real-world programmers. Instead, these languages, like the Universal Modeling Language (UML) [Booch 1998], provide a graphical tool that includes an Object Constraint Language (OCL) [Warmer 1998] for modeling properties of objects and their interrelationships in a software design. Because of their separation from the compiled code, these modeling languages have served mainly for software documentation and as artifacts for research in software methodology.

However, with the recent emergence of Eiffel [Meyer 1990], ESC/Java [Flanagan 2002], Spark/Ada [Barnes 2003], JML [Leavens 2004], and the notion of *design by contract* [Meyer 1997], this situation is changing rapidly. These new developments provide programmers with access to rigorous tools and verification techniques that are fully integrated with the runitime system itself. Design by contract is a formalism through which interactions between objects and their clients can be precisely described and dynamically checked. ESC/JAVA is a code-level language for annotating and statically checking a program for a wide variety of common errors.

The Java Modeling Language (JML) provides code level extensions to the Java language so that programs can include such formal specifications and their enforcement at run time. Spark/Ada is a proprietary system that provides similar extensions to the Ada language. To explore the impact of these developments on program correctness, we illustrate the use of JML and design by contract in the fourth section of this chapter.

Functional programs, because of their close approximation to mathematical

functions, provide a more direct vehicle for formal proof of program correctness. We discuss the application of proof techniques to functional programs using Haskell examples in the fifth section of this chapter.

12.2 *REVIEW OF LOGIC AND PROOF

Propositional logic provides the mathematical foundation for boolean expressions in programming languages. A *proposition* is formed according to the following rules:

- The constants *true* and *false* are propositions.
- The variables $p, q, r, \ldots$, which have values *true* or *false,* are propositions.
- The operators $\wedge, \vee, \Rightarrow, \Leftrightarrow$, and $\neg$, which denote conjunction, disjunction, implication, equivalence, and negation, respectively, are used to form more complex propositions. That is, if P and Q are propositions, then so are $P \wedge Q, P \vee Q, P \Rightarrow Q, P \Leftrightarrow Q$, and $\neg P$.

By convention, negation has highest precedence, followed by conjunction, disjunction, implication, and equivalence, in that order. Thus, the expression

$$p \vee q \wedge r \Rightarrow \neg s \vee t$$

is equivalent to

$$((p \vee (q \wedge r)) \Rightarrow ((\neg s) \vee t)).$$

Propositions provide symbolic representations for *logic expressions;* that is, statements that can be interpreted as either *true* or *false.* For example, if p represents the proposition "Mary speaks Russian" and q represents the proposition "Bob speaks Russian," then $p \wedge q$ represents the proposition "Mary and Bob both speak Russian," and $p \vee q$ represents "Either Mary or Bob (or both) speaks Russian." If, furthermore, r represents the proposition "Mary and Bob can communicate," then the expression $p \wedge q \Rightarrow r$ represents "If Mary and Bob both speak Russian, then they can communicate."

Predicates include all propositions such as the above, and also include variables in various domains (integers, reals, strings, lists, etc.), boolean-valued functions with these variables, and quantifiers. A *predicate* is a proposition in which some of the boolean variables are replaced by boolean-valued functions and quantified expressions.

A *boolean-valued function* is a function with one or more arguments that delivers *true* or *false* as a result. Here are some examples:

prime(n)—*true* if the integer value of n is a prime number; *false* otherwise.

$0 \leq x + y$—*true* if the real sum of x and y is nonnegative.

speaks(x, y)—*true* if person x speaks language y.

Table 12.1: Summary of Predicate Logic Notation

Notation	Meaning
true, false	Boolean (truth) constants
$p, q, \ldots$	Boolean variables
$p(x, y \ldots), \ q(x, y \ldots), \ \ldots$	Boolean functions
$\neg p$	Negation of p
$p \wedge q$	Conjunction of p and q
$p(x) \vee q(x)$	Disjunction of p and q
$p(x) \Rightarrow q(x)$	Implication: p implies q
$p(x) \Leftrightarrow q(x)$	Logical equivalence of p and q
$\forall x \ p(x)$	Universally quantified expression
$\exists x \ p(x)$	Existentially quantified expression
$p(x)$ is valid	Predicate $p(x)$ is *true* for every value of x
$p(x)$ is satisfiable	Predicate $p(x)$ is *true* for at least one value of x
$p(x)$ is a contradiction	Predicate $p(x)$ is *false* for every value of x

A predicate combines these kinds of functions using the operators of the propositional calculus and the quantifiers $\forall$ (meaning "for all") and $\exists$ (meaning "there exists"). Here are some examples:

$0 \leq x \wedge x \leq 1$—*true* if x is between 0 and 1, inclusive; otherwise *false*.

$speaks(x, Russian) \wedge speaks(y, Russian) \Rightarrow communicateswith(x, y)$—*true* if the fact that both x and y speak Russian implies that x communicates with y; otherwise *false*.

$\forall x(speaks(x, Russian))$—*true* if everyone on the planet speaks Russian; *false* otherwise.

$\exists x(speaks(x, Russian))$—*true* if at least one person on the planet speaks Russian; *false* otherwise.

$\forall x \exists y(speaks(x, y))$—*true* if every person on the planet speaks some language; *false* otherwise.

$\forall x(\neg literate(x) \Rightarrow (\neg writes(x) \wedge \neg \exists y(book(y) \wedge hasread(x, y))))$—*true* if every illiterate person x does not write and has not read a book.

Table 12.1 summarizes the meanings of the different kinds of expressions that can be used in propositional and predicate logic.

Predicates that are *true* for all possible values of their variables are called *valid*. For instance, $even(x) \vee odd(x)$ is valid, since all integers x are either even or odd. Predicates that are *false* for all possible values of their variables are called *contradictions*. For instance, $even(x) \wedge odd(x)$ is a contradiction, since no integer can be both even and odd.

Table 12.2: Properties of Predicates

Property	Meaning	
Commutativity	$p \vee q \Leftrightarrow q \vee p$	$p \wedge q \Leftrightarrow q \wedge p$
Associativity	$(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$	$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$
Distributivity	$p \vee q \wedge r \Leftrightarrow (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \Leftrightarrow p \wedge q \vee p \wedge r$
Idempotence	$p \vee p \Leftrightarrow p$	$p \wedge p \Leftrightarrow p$
Identity	$p \vee \neg p \Leftrightarrow true$	$p \wedge \neg p \Leftrightarrow false$
deMorgan	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
Implication	$p \Rightarrow q \Leftrightarrow \neg p \vee q$	
Quantification	$\neg\forall x\ p(x) \Leftrightarrow \exists x\ \neg p(x)$	$\neg\exists x\ p(x) \Leftrightarrow \forall x\ \neg p(x)$

Predicates that are *true* for some particular assignment of values to their variables are called *satisfiable.* For example, the predicate *speaks*(x, *Russian*) is satisfiable (but not valid) since presumably at least one person on the planet speaks Russian (but there are others who do not). Similarly, the predicate $y \geq 0 \wedge n \geq 0 \wedge z = x(y - n)$ is satisfiable but not valid since different selections of values for x, y, z, and n can be found that make this predicate either *true* or *false*.

Predicates have various algebraic properties, which are often useful when we are analyzing and transforming logic expressions. A summary of these properties is given in Table 12.2.

The commutative, associative, distributive, and idempotence properties have straightforward interpretations. The identity property simply says that either a proposition or its negation must always be *true,* but that both a proposition and its negation cannot simultaneously be *true*.

DeMorgan's property provides a convenient device for removing disjunction (or conjunction) from an expression without changing its meaning. For example, saying "it is not raining or snowing" is equivalent to saying "it is not raining and it is not snowing." Moreover, this property asserts the equivalence of "not both John and Mary are in school" and "either John or Mary is not in school."

Similarly, the implication and quantification properties provide vehicles for removing implications, universal, or existential quantifiers from an expression without changing its meaning. For example, "not every child can read" is equivalent to "there is at least one child who cannot read." Similarly, "There are no flies in my soup" is equivalent to "every fly is not in my soup."

12.2.1 Inference Rules and Direct Proof

An argument to be proved often takes the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q$, where the p's are the hypotheses and q is the conclusion.

A *direct proof* of such an argument is a sequence of valid predicates, each of which is either identical with an hypothesis or derivable from earlier predicates in the sequence using a *property* (Table 12.2) or an *inference rule.* The last

Table 12.3: Inference Rules for Predicates

Inference Rule	Meaning
Modus ponens	$p, p \Rightarrow q \ \vdash \ q$
Modus tollens	$p \Rightarrow q, \neg q \ \vdash \ \neg p$
Conjunction	$p, q \ \vdash \ p \wedge q$
Simplification	$p \wedge q \ \vdash \ p$
Addition	$p \ \vdash \ p \vee q$
Universal instantiation	$\forall x \ p(x) \ \vdash \ p(a)$
Existential instantiation	$\exists x \ p(x) \ \vdash \ p(a)$
Universal generalization	$p(x) \ \vdash \ \forall x \ p(x)$
Existential generalization	$p(a) \ \vdash \ \exists x \ p(x)$

predicate in the proof must be the argument's conclusion q. Each predicate in the sequence is accompanied by a "justification," which is a brief notation of what derivation rule and what prior steps were used to arrive at this predicate.

Some of the key inference rules for predicates are summarized in Table 12.3. To interpret these rules, if the expression(s) on the left of $\vdash$ appear in a proof, they can be replaced later in the sequence by the expression on the right (but *not* vice versa). Below is a direct proof of the following argument:

> Every student likes crossword puzzles. Some students like ice cream. Therefore, some students like ice cream and crossword puzzles.

Suppose we assign the following names to the predicates in this problem:

$S(x)$ = "x is a student"
$C(x)$ = "x likes crossword puzzles"
$I(x)$ = "x likes ice cream"

Then the argument can be rewritten as:

$$\forall x(S(x) \rightarrow C(x)) \wedge \exists x(S(x) \wedge I(x)) \rightarrow \exists x(S(x) \wedge C(x) \wedge I(x))$$

Here is a direct proof of this argument:

1.	$\forall x(S(x) \rightarrow C(x))$	Hypothesis
2.	$\exists x(S(x) \wedge I(x))$	Hypothesis
3.	$S(a) \wedge I(a)$	2, Existential instantiation
4.	$S(a) \rightarrow C(a)$	1, Unversal instantiation
5.	$S(a)$	3, Simplification
6.	$C(a)$	4, 5, Modus ponens
7.	$S(a) \wedge C(a) \wedge I(a)$	3, 6, Addition
8.	$S(a) \wedge I(a) \wedge C(a)$	7, Commutativity
9.	$\exists x(S(x) \wedge I(x) \wedge C(x))$	8, Existential generalization

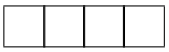
The notations in the right-hand column are justifications for the individual steps in the proof. Each justification includes line numbers of prior steps from which it is inferred by a property or inference rule from Table 12.2 or 12.3.

12.2.2 Induction Proof

This method of proof is very important in program correctness, as well as many other areas of computer science. An *induction proof* can be applied to any argument having the form $\forall n\ p(n)$. Here, the domain of n must be countable, as is the case for the integers or the strings of ASCII characters, for example. The strategy for an induction proof has two steps:[1]

1. (Basis step) Prove $p(1)$.
2. (Induction step) Assuming the hypothesis that $p(k)$ is valid for an arbitrary value of $k > 1$ in the domain of n, prove $p(k + 1)$.

Consider the following example. Suppose we want to prove by induction that the number of distinct sides in a row of n adjacent squares is $3n + 1$. Here, for example, is a row of 4 adjacent squares, having 13 adjacent sides:



Here is the inductive proof:

1. The basis step is simple, since 1 square has $3 \times 1 + 1 = 4$ sides (count 'em).
2. For the induction step, assume as our *induction hypothesis* that k squares have $3k + 1$ sides. Now we need to prove that this leads to the conclusion that $k + 1$ squares have $3(k + 1) + 1$ sides. But to construct a $k + 1$-square row, we simply add 3 sides to the k-square row. This leads to the conclusion that the number of sides in a $k + 1$-square row is $3k + 1 + 3 = 3(k + 1) + 1$, which completes the induction step.

12.3 AXIOMATIC SEMANTICS

While it is important for software designers to understand what a program does in all circumstances, it is also important to be able to confirm, or *prove* that the program does what it is *supposed* to do under all circumstances. That is, if someone presents a specification for what a program is supposed to do, the programmer should be able to prove to that person, beyond a reasonable doubt, that the program and this specification are formally in agreement with each other. When that is done, the program is said to be "correct."

For instance, suppose we want to prove that the C++Lite function `Max` in Figure 12.1 actually computes as its result the maximum value of any two arguments that correspond to its parameters `a` and `b`.

Calling this function one time will obtain an answer for a particular pair of arguments for `a` and `b`, such as 8 and 13. But each of the parameters `a` and `b`

[1] This strategy is often called "weak induction." The strategy of "strong induction" differs only in the assumption that it makes during the induction step. That is, with strong induction you can assume the hypothesis that $p(1), p(2), \ldots, p(k)$ are all valid for an arbitrary value of $k > 1$, in order to prove $p(k + 1)$.

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

Figure 12.1: A C++Lite Max Function

defines a wide range of integer values – something like 4 million of them. So to call this function 16 trillion times, each with a different pair of values for `a` and `b`, to prove its correctness would be an infeasible task.

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

12.3.1 Fundamental Concepts

Axiomatic semantics is based on the notion of an *assertion,* which is a predicate that describes the *state* of a program at any point during its execution. An assertion can define the meaning of a computation, as in for example "*the maximum of a and b*," without concern for how that computation is accomplished.

The code in Figure 12.1 is just one way of algorithmically expressing the maximum computation; even for a function this simple, there are other variations. No matter which variation is used, the following assertion Q can be used to describe the function *Max* declaratively:

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

That is, this predicate specifies the mathematical meaning of the function `Max(a, b)` for any integer values of `a` and `b`. It thus describes *what* should be the result, rather than *how* it should be computed. To prove that the program in Figure 12.1 actually computes $max(a, b)$, we must prove that the logical expression Q is valid for all values of `a` and `b`. In this formal verification exercise, Q is called a *postcondition* for the program `Max.`

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

In the case of `Max`, the postcondition Q can be satisfied for any pair of integer

values of `a` and `b`. This suggests the following precondition:

$$P = true$$

That is, for the program to be proved correct, no constraints or preconditions on the values of `a` and `b` are needed.[2]

One final consideration must be mentioned before we look at the details of correctness proofs themselves. That is, for *some* initial values of the variables that satisfy the program's precondition P, executing the program may *never* reach its last statement. This situation can occur when either of the following abnormal events occurs:

1. the program tries to compute a value that cannot be represented on the (virtual) machine where it is running, or

2. the program enters an infinite loop.

To illustrate the first event, suppose we call the C++ function in Figure 12.3 to compute the factorial of n for a large enough value of n. E.g., $n = 21$ gives $n! = 51090942171709440000$, which cannot be computed using 32- or 64-bit integers. An attempt to perform such a calculation would cause normal execution to interrupted by an `overflow_error` exception.[3]

In this section, we focus on proving program correctness only for those initial values of variables in which neither of these two abnormal events occurs and the program runs to completion. This constrained notion of correctness is called *partial correctness*. In a later section, we revisit the question of program correctness for cases where exceptions are raised at run time.

Recent research has developed tools and techniques by which exception handling can be incorporated into a program's formal specifications, thus allowing correctness to be established even when abnormal termination occurs. However, the second abnormal event noted above, where a program loops infinitely, cannot be covered automatically for the general case. That is assured by the unsolvability of the halting problem.[4]

Proofs of termination for a particular program and loop can often be constructed by the programmer. For instance, a C++/Java `for` loop that has explicit bounds and non-zero increment defines a finite sequence of values for the control variable. Thus, any such loop will always terminate. On the other

[2]Such a weak precondition is not always appropriate. For instance, if we were trying to prove the correctness of a function `Sqrt(x)` that computes the `float` square root of the `float` value of `x`, an appropriate precondition would be $P = x \geq 0$. We will return to this particular example later in the chapter.

[3]The Java virtual machine, curiously, does not include integer overflow among its exceptions, although it does include division by zero. Thus, the computation of 21! by the Java program in Figure 12.4 gives an incorrect result of -1195114496, and no run-time exception is raised! Haskell, however, does this calculation correctly for every value of n, since it supports arithmetic for arbitrarily large integers.

[4]This well-known result from the theory of computation confirms that no program can be written which can determine whether any other arbitrary program halts for all possible inputs.

$\{true\}$

```
if (a >= b)
  m = a;
else
  m = b;
```

$\{m = max(a, b)\}$

Figure 12.2: The Goal for Proving the Correctness of `Max(a, b)`

hand, proof of termination for a `while` loop is often not possible, since the `test` condition for continuing the loop might not submit to formal analysis. For example, termination of the loop `while (p(x)) s` reverts to the question of whether or not `p(x)` ever becomes `false`, which is sometimes not provable.

These considerations notwithstanding, we can prove the (partial) correctness of a program by placing its precondition in front of its first statement and its postcondition after its last statement, and then systematically deriving a series of valid predicates as we simulate the execution of the program's code one instruction at a time. For any statement or series of statements s, the predicate

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

formally represents the idea that s is partially correct with respect to the precondition P and the postcondition Q. This expression is called a *Hoare triple* and asserts "execution of statements s, beginning in a state that satisfies P, results in a state that satisfies Q."[5]

To prove the partial correctness of our example program, we need to show the validity of the Hoare triple in Figure 12.2. We do this by deriving intermediate Hoare triples $\{P\}\ s\ \{Q\}$ that are valid for the individual statements s in the program, beginning with the last statement and the program's postcondition. This process continues until we have derived a Hoare triple like the one in Figure 12.2, which completes the correctness proof.

How are these intermediate Hoare triples derived? That is done by using rules of inference that characterize what we know about the behavior of the different types of statements in the language. Programs in C++Lite-like languages have four different types of statements: *Assignment*s, *Block*s (sequences), *Conditional*s, and *Loop*s. Each statement type has an *inference rule* which defines the meaning of that statement type in terms of the pre- and postconditions that it satisfies. The rules for C++Lite statement types are shown in Table 12.4.

As for the notation in Table 12.4, we note first that all five of these rules are of the form $p \vdash q$, which is similar to that used in the previous section's discussion of the predicate calculus. Second, we note that the comma (,) in rules of the form $p_1, p_2 \vdash q$ denotes conjunction. Thus, this form should be read, "if p_1 and p_2 are valid then q is valid."

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

Table 12.4: Inference Rules for Different Types of C++Lite Statements

Statement Type (s)	**Inference Rule**
1. *Assignment* s.target = s.source;	$true \vdash \{Q[s.\text{target} \leftarrow \text{s.source}]\}\ s\ \{Q\}$
2. *Block* (sequence) s_1 ; s_2	$\{P\}\ s_1\ \{R\}, \{R\}\ s_2\ \{Q\} \vdash \{P\}\ s_1\ ;\ s_2\ \{Q\}$
3. *Conditional* if (s.test) s.thenpart else s.elsepart	$\{s.\text{test} \wedge P\}\ s.\text{thenpart}\ \{Q\}$, $\{\neg s.\text{test} \wedge P\}\ s.\text{elsepart}\ \{Q\} \vdash \{P\}\ s\ \{Q\}$
4. *Loop* while (s.test) s.body	$\{s.\text{test} \wedge R\}\ s.\text{body}\ \{R\} \vdash \{R\}\ s\ \{\neg s.\text{test} \wedge R\}$
5. *Rule of consequence*	$P \Rightarrow P', \{P'\}\ s\ \{Q'\}, Q' \Rightarrow Q \vdash \{P\}\ s\ \{Q\}$

The *Assignment* inference rule has *true* as its premise, guaranteeing that we can always derive the conclusion. The notation $Q[t \leftarrow s]$ means "the predicate that results from replacing all occurrences of t in Q by s." For instance, if $\{Q\} = \{x = 1 \wedge y = 4\}$ then $\{Q[x \leftarrow 1]\} = \{1 = 1 \wedge y = 4\}$. Applied to an assignment, rule 1 guarantees that the following Hoare triple is valid:

$\{a = max(a, b)\}$
  m = a ;
$\{m = max(a, b)\}$

That is, the assignment rule allows us to reason backwards through the assignment `m = a;` and derive a new precondition $\{a = max(a, b)\}$ for that assignment which is the result of replacing all instances of m in the postcondition by a.

Inference rule 5, the *rule of consequence*, allows us to perform arithmetic and logical simplification in a predicate during the proof process. In particular, we can strengthen a statement's precondition (i.e., replace P by P' when $P \Rightarrow P'$) in order to match it better with the postcondition of the previous statement during the proof process. Similarly, we can weaken a postcondition (i.e., replace Q by Q' when $Q' \Rightarrow Q$) in order to match it better with the precondition of the next statement. In our example, we can perform precondition strengthening by identifying the assertion $\{P'\} = \{a = max(a, b)\}$, which is implied mathematically by $\{P\} = \{a \geq b\}$, so we can utilize rule 5 as follows:

$$a \geq b \Rightarrow a = max(a, b)\ ,\ \{a = max(a, b)\}\ \texttt{m = a;}\ \{\text{m} = max(a, b)\} \vdash$$
$$\{a \geq b\}\ \texttt{m = a;}\ \{\text{m} = max(a, b)\}$$

to derive the Hoare triple:

$\{a \geq b\}$
  m = a ;
$\{m = max(a, b)\}$

The rule of consequence also suggests that any one of several alternative preconditions might be derived from a given Hoare triple, using various properties that we know from the mathematical and logical domains of the variables that are in play. That precondition which is the *least* restrictive on the variables in play is called the *weakest precondition.* For instance, the precondition $\{a \geq b\}$ is the weakest precondition for the assignment `m = a`; and its postcondition $\{m = max(a, b)\}$. Finding weakest preconditions is important because it enables simplification of the proof at various stages.

A strategy for proving the partial correctness of the rest of the program in Figure 12.1 works systematically from the postcondition backwards through the if, and then through the two assignment statements in the then- and else-part toward a derivation of the precondition for that program. If that strategy is successful, the program is said to be *correct* with respect to its given pre- and postconditions. Let's finish the proof of this program.

We use rules 1 and 5 again with the postcondition on the assignment in the else part of the if statement, to obtain:

$\{a \leq b\}$
  m = b;
$\{m = max(a, b)\}$

Since $a \leq b$ is implied by $a < b \wedge true$ (using rule 5 again), we can apply rule 3 to this conditional statement and establish the following inference:

$\{a \geq b \wedge true\}$ `m = a;` $\{m = max(a, b)\}$,
$\{a < b \wedge true\}$ `m = b;` $\{m = max(a, b)\}$ $\vdash$
$\{true\}$ `if (a>=b) m = a; else m = b;` $\{m = max(a, b)\}$

Thus, we have proven the correctness of the entire program in Figure 12.1 by deriving the Hoare triple in Figure 11.2 using the inference rules of program behavior. In the next section, we consider the issue of correctness for programs that contain loops.

12.3.2 Correctness of Programs with Loops

The (partial) correctness of a loop depends not only on logically connecting the pre- and postconditions of its Hoare triple with the rest of the program, but also on the correctness of each iteration of the loop itself. For that purpose, we introduce the idea of a *loop invariant* and use induction to assist with the proof.

To illustrate these ideas, suppose we want to prove that the C/C++ function `Factorial` in Figure 12.3 actually computes as its result $n!$, for any integer n where $n \geq 1$, assuming the function terminates normally. By $n!$ we mean the product $1 \times 2 \times \cdots \times n$.

The precondition P for `Factorial` is $1 \leq n$, while the postcondition is $f = n!$. In general, a program involving a loop uses rule 4 of Table 12.4 to break the code into three parts, as shown in Figure 12.4. There, P is the program's precondition, Q is its postcondition, and R is known as the *loop invariant.*

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

Figure 12.3: A C/C++ Factorial Function

$\{P\}$
initialization
$\{R\}$
while (test) {
  loop body
}
$\{\neg test \wedge R\}$
finalization
$\{Q\}$

Figure 12.4: Hoare Triples for a Program with a Loop

A loop invariant is an assertion that remains *true* before and after every iteration of the loop. In general, there is no algorithmic way to derive a loop invariant from the program's pre- and postconditions.[6] Thus, it is necessary for the programmer to supply an invariant for every loop in a program if we are to prove correctness for the whole program.

For the *Factorial* function given in Figure 12.3, the loop invariant is $R = \{1 \leq i \wedge i \leq n \wedge (f = i!)\}$. Thus, the *Factorial* program with its pre- and postconditions and loop invariant can be rewritten as:

$\{1 \leq n\}$
```
  f = 1;
  i = 1;
```
$\{1 \leq i \wedge i \leq n \wedge f = i!\}$
```
  while (i < n) {
     i = i + 1;
     f = f * i;
  }
```
$\{i \geq n \wedge 1 \leq i \wedge i \leq n \wedge f = i!\}$
```
  ;
```
$\{f = n!\}$

[6] Finding a loop invariant is often tricky. Interested readers are encouraged to find additional sources (e.g., [Gries 1981]) that develop this very interesting topic in more detail.

This step reduces the problem of proving the correctness of the original program to the three smaller problems:

(1) proving the initialization part;

(2) proving (inductively) that the premise R of rule 4 is valid for all iterations of the loop; and

(3) proving the finalization part.

These subproblems may be proved in any convenient order.

The third subproblem is easiest, since it involves only the *Skip* statement. Since that statement does nothing, its precondition must directly imply its post-condition. This can be shown by repeated applications of rule 5 and using our algebraic skills:

$$\begin{aligned} i \geq n \wedge 1 \leq i \wedge i \leq n \wedge f = i! &\Rightarrow \\ (i = n) \wedge f = i! &\Rightarrow \\ f = n! & \end{aligned}$$

That is, since $i \geq n$ and $i \leq n$, it follows that $i = n$.

A strategy for solving the first subproblem uses rule 2 to break a *Block* into its individual components and then find the linking assertion $\{R'\}$:

$\{1 \leq n\}$
```
  f = 1;
```
$\{R'\}$
```
  i = 1;
```
$\{1 \leq i \wedge i \leq n \wedge f = i!\}$

The linking assertion R' can be found by using rule 1 with the second assignment, so that $R' = \{1 \leq 1 \wedge 1 \leq n \wedge f = 1!\}$. So now we can insert this expression for R' and apply rule 1 to the first assignment:

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

obtaining $\{1 \leq 1 \wedge 1 \leq n \wedge 1 = 1!\}$, which simplifies to $1 \leq n$. Thus, we have proved the validity of the *Block* by showing the validity of:

$$\begin{array}{rlll} \{1 \leq n\} & \mathrm{f} = 1; & \{1 \leq 1 \wedge 1 \leq n \wedge f = 1!\}, & \\ \{1 \leq 1 \wedge 1 \leq n \wedge f = 1!\} & \mathrm{i} = 1; & \{1 \leq i \wedge i \leq n \wedge f = i!\} & \vdash \\ \{1 \leq n\} & \mathrm{f} = 1; \mathrm{i} = 1; & \{1 \leq i \wedge i \leq n \wedge f = i!\} & \end{array}$$

Solving the second subproblem requires that we validate rule 4 for our invariant R and every iteration of the loop. So we must validate:

$\{s.\text{test} \wedge R\} s.\text{body} \{R\} \vdash \{R\} s \{\neg s.\text{test} \wedge R\}$, where `s` is a loop statement.

To do this for our particular loop test $i < n$, invariant R, and loop body, we need to show the validity of the following Hoare triple:

$\{i < n \wedge 1 \leq i \wedge i \leq n \wedge f = i!\}$
```
  i = i + 1;
  f = f * i;
```
$\{1 \leq i \wedge i \leq n \wedge f = i!\}$

Let us use rule 2 again to derive a linking assertion *R'* between the two statements in this *Block*:

$\{i < n \wedge 1 \leq i \wedge i \leq n \wedge f = i!\}$
```
  i = i + 1;
```
$\{R'\}$
```
  f = f * i;
```
$\{1 \leq i \wedge i \leq n \wedge f = i!\}$

Applying rule 1 to the second assignment gives:

$$R' = 1 \leq i \wedge i \leq n \wedge f \times i = i!$$

Applying rule 1 to the first assignment and R' gives:

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

Now, we need to show that:

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

in which case 5 can be used to complete our proof. To do this, we use the following principle from logic:

$$p \Rightarrow q, p \Rightarrow r, p \Rightarrow s \ \vdash p \Rightarrow q \wedge r \wedge s$$

and proceed by proving each term of this consequent separately.

First, we must show:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge f = i! \Rightarrow 1 \leq i+1$$

This is valid, since $1 \leq i$ is a term in the antecedent and $i \leq i+1$ is always valid, algebraically.

Second, we must show:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge f = i! \Rightarrow i+1 \leq n,$$

This is also valid, since $i < n$ is in the antecedent and it follows algebraically that $i+1 \leq n$.

Third, we must show:

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

We can safely divide both sides of $f \times (i+1) = (i+1)!$ by $i+1$, since $i \geq 1$, resulting in:

$$i < n \wedge 1 \leq i \wedge i \leq n \wedge f = i! \Rightarrow f = i!$$

This is valid, since its consequent appears as a term in its antecedent.

This last step amounts to an induction proof, in which we show both: (1) the basis step in which $R(1)$ is established, and (2) the induction step in which $R(i) \Rightarrow R(i+1)$ is established for invariant *R(i)* over all $i = \{i, \ldots n\}$. Since loops have indeterminate length, the invariant R is expressed as a function $R(i)$ on the number of iterations i that have taken place. The basis step, in which $R(1)$ is valid, corresponds to the validity of R before the first iteration.

This concludes our proof of the (partial) correctness of the *Factorial* function in Figure 12.3. Note that our proof does not address correctness when the calculation of $n!$ cannot be completed because too large a value for n was passed. We return to this important issues in a later section.

12.3.3 Perspectives on Formal Methods

Axiomatic semantics and the corresponding techniques for proving the correctness of imperative programs were developed in the late 1960s and early 1970s. At that time, many expected that most programs would routinely be proven correct, and that software products would become more reliable in general. Given the current state of the software industry today, it is clear that these expectations have come nowhere near to being fulfilled.

To further advance this discussion, the emergence of a field called *formal methods* in software design has emerged during the last twenty years. This field attempts to develop and apply correctness tools and techniques to two different phases of the software development process – software requirements analysis and software validation (testing). Tools like the Universal Modeling Language (UML) and the Java Modeling Language (JML), for example, have emerged to help designers specify more formally the behavior of components in large systems. Techniques like *design by contract* [Meyer 1990] have been proposed to provide a basis upon which software components can be validated with a higher degree of reliability than the various testing techniques of the past.

Within this setting, the utility and importance of correctness proofs in software design has continued to be a subject of heated debate, especially throughout the most recent decade. Many software engineers reject the use of formal methods for software validation [DeMillo 1979], arguing that it is too complex and time-consuming a process for most programmers to master. Instead they suggest that more elaborate testing methods be used to convince designers and users that the software runs correctly most of the time.

A counter-argument to this view was made many years ago by Dijkstra [1972], who simply recognized that testing could only prove the presence of bugs, never their absence. For example, a simple program that inputs two 32-bit integers, computes some function, and outputs a 32-bit integer has 2^{64} possible inputs (approximately 10^{20}), so that even if one could test and verify

(!) 100 million test cases per second, a complete test of this simple program would take approximately 10^5 years. Imagine how long it would take to perform a complete test of a complex program using such exhaustive testing methods.

Of course, correctness proofs are also time-consuming and difficult. Most programmers and software engineers do not have the mathematical training to incorporate such formal verification methods into their design process. However, there have been many software products designed with the careful use of formal methods, both in the verification phase and (most importantly, perhaps) in the design phase. The outcomes reported for these projects have been impressive – the resulting software exhibits a far higher standard of reliability and robustness than software designed using traditional techniques. However, some argue that this outcome is achieved at a price – increased design time and development cost overall, as compared with traditional testing methods.

Given these tradeoffs, it is fair to ask whether there is a middle ground between complex and time-consuming formal verification of software, such as the one described above, and traditional testing methods which have helped produce highly unreliable software? We believe so.

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

Second, the methods that define the behaviors of a class in an object-oriented program are often quite small in size. Informal proofs of correctness for such methods are routinely possible, although they are not often practiced. One reason for this is that many programmers, who are mostly untrained in the use of formal logic, cannot even state the pre- and postconditions for the methods they write.

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

As an example of how such formalism could be put to better use, consider Sun's `javadoc` documentation for the various `String` methods in JDK 1.5.[7] There, the defining comment for the `substring` method:

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

reads:

> "Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex-1`. Thus the length of the substring is `endIndex-beginIndex`. Throws `IndexOutOfBoundsException` if the `beginIndex` is negative, or `endIndex` is larger than the length of this String object, or `beginIndex` is larger than `endIndex`."

How would an implementor carry out a correctness proof for `substring`, given such a vague specification? Wouldn't this specification need to be trans-

[7] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html#substring(int)

$\{0 \leq beginIndex < endIndex \leq n \wedge s = s_0 s_1 \ldots s_{n-1}\}$
`s.substring(beginIndex, endIndex);`
$\{s = s_0 s_1 \ldots s_{n-1} \wedge result = s_{beginIndex} s_{beginIndex+1} \ldots s_{endIndex-1}\}$

Figure 12.5: Formal specification of the Java `substring` Method.

lated into a logical representation so that following questions can be answered more clearly and precisely?

> What are the valid values for `beginIndex` and `endIndex`?
> For a given string $s = s_0 s_1 \ldots s_{n-1}$, what result is normally returned?
> What happens in the abnormal case, when either index is not valid?

A programmer interested in producing even an informal proof of an implementation of `substring` would at least require a more formal description of this method's pre- and postconditions. Figure 12.5 shows one such description (which omits, for simplicity, the abnormal case). Unlike the informal description, this description formally specifies the acceptable values of *beginIndex*, *endIndex*, and the length n of the string s for which `substring` is well defined, as well as the exact nature of the result itself.

In the next section, we discuss recent improvements in language design and software methodology that are helping developers address these kinds of problems more effectively.

12.3.4 Formal Methods Tools: JML

During the last several years, new tools and modeling techniques have been developed to assist software developers in making specifications more rigorous and designs more reliable. One recently developed tool is called the *Java Modeling Language* (JML for short), which is fully implemented and adaptable to a variety of software design and verification activities. A promising modeling technique is called *design by contract* [Meyer 1997], which provides an operational framework within which object-oriented programs can be reliably designed.

These two work together. That is, JML provides a language for incorporating and checking formal specifications in Java programs, while design by contract provides the operational guidelines within which specifications can be used to ensure system integrity when classes interact with each other.

In this section, we introduce the features of JML as they apply to the formal specification and verification of an individual function, such as the *Factorial* function that we specified and verified by hand in the previous section. We also show how JML allows us to specify run-time exceptions, providing a more robust vehicle than the pure Hoare triples in a real computational setting where exceptions actually occur. Consider the JML-annotated version of the *Factorial* function shown in Figure 12.6.

This version differs from the C/C++ program in Figure 12.3 in only one significant way. That is, the function `Factorial` is annotated by two stylized

```
public class myFactorial {

/*@ requires 1 <= n;
    ensures \result == (\product int i; 1<=i && i<=n; i);
  @*/
static int Factorial (int n) {
  int f = 1;
  int i = 1;
/*@ loop_invariant i <= n &&
        f == (\product int j; 1 <= j && j <= i; j);
  @*/
  while (i < n) {
    i = i + 1;
    f = f * i;
  }
  return f;
}
public static void main(String[] args) {
   int n = Integer.parseInt(args[0]);
   System.out.println("Factorial of " + n +
                      " = " + Factorial(n));
}
}
```

Figure 12.6: A JML-annotated Java Version of Factorial

Table 12.5: Summary of JML Expressions

JML Expression	**Meaning**
`requires p ;`	`p` is a precondition for the call
`ensures p ;`	`p` is a postcondition for the call
`signals (E e) p;`	When exception type `E` is raised by the call, then `p` is a postcondition
`loop_invariant p;`	`p` is a loop invariant
`invariant p ;`	`p` is a class invariant (see next section)
`\result` $==$ `e`	`e` is the result returned by the call
`\old(v)`	the value of `v` at entry to the call
`(\product int x ; p(x); e(x))`	$\prod_{x \in \mathtt{p(x)}} \mathtt{e(x)}$; i.e., the product of `e(x)`
`(\sum int x ; p(x); e(x))`	$\sum_{x \in \mathtt{p(x)}} \mathtt{e(x)}$; i.e., the sum of `e(x)`
`(\min int x ; p(x); e(x))`	$\min_{x \in \mathtt{p(x)}} \mathtt{e(x)}$; i.e., the minimum of `e(x)`
`(\max int x ; p(x); e(x))`	$\max_{x \in \mathtt{p(x)}} \mathtt{e(x)}$; i.e., the maximum of `e(x)`
`(\forall type x ; p(x) ; q(x))`	$\forall \mathtt{x} \in p(x) : \mathtt{q(x)}$
`(\exists type x ; p(x) ; q(x))`	$\exists \mathtt{x} \in p(x) : \mathtt{q(x)}$
`p ==> q`	$p \Rightarrow q$
`p <== q`	$q \Rightarrow p$
`p <==> q`	$p \Leftrightarrow q$
`p <=!=> q`	$\neg(p \Leftrightarrow q)$

comments (written `/*@...@*/`), one containing `requires` and `ensures` and the other beginning `loop_invariant`. The first comment is the JML encoding for the pre- and postconditions P and Q that are used to form a Hoare triple out of the `Factorial` function in preparation for its correctness proof. The second comment is the JML encoding of the assertion R that represents the loop invariant in that proof.

Each one of the `requires`, `ensures`, and `loop_invariant` clauses has a Java-style boolean expression as its main element. Variables mentioned in these clauses, like `n`, are the ordinary variables and parameters that are visible to the `Factorial` function's code itself. Additional names mentioned in these clauses are of two types, local variables (like `i` and `j` in this example) and JML reserved words (like `\result` and `\product` in this example). *The important caveat for JML clauses like these is that their execution must have no side effect on the state of the computation.*

In JML, the reserved word `\result` uniquely identifies the result returned by a non-void function. The reserved word `\product` is a mathematical quantifier (a summary of the key JML quantifiers and operators appears in Table 12.5), and carries the same meaning as Π in mathematical expressions. The rest of the `ensures` clause defines the limits on the controlling variable `i` and the expression that is the subject of the calculation of the product. Thus, the JML expression `(\product int i; 1<=i && i<=n; i)` is equivalent to the mathematical expression $\prod_{i=1}^{n} i$.

Two significant points need to be made about the JML `requires`, `ensures`, and `loop_invariant` clauses in relation to our verification exercise using the pre-and postconditions P and Q and loop invariant R in the previous section. First, these three clauses are integrated with the Java compiler and runtime system, enabling their specifications to be checked for syntax, type, and run-time errors. Second, however, these three clauses do not by themselves provide a platform upon which the code will be automatically proved correct by an anonymous agent.[8]

So we can conclude that JML provides a formal language for defining the preconditions, postconditions and loop invariants of an executable function, and automatically checking at run time that a call to the function will:

1. satisfy the precondition P,

2. satisfy each loop invariant R during execution of the function, and

3. return a result that satisfies the postcondition Q.

While this is not formal verification *per se*, JML does provide a robust basis for integrating a program's specifications with its code. This occurs because JML specifications can be actively compiled and interpreted with each compile and run of the program.[9]

Here are the results of four different runs for the program in Figure 12.4 that illustrate various possible outcomes.

```
% jmlrac myFactorial 3
Factorial of 3 = 6
% jmlrac myFactorial -5
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError:
   by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 3, character 15 when
        'n' is -5
        at myFactorial.checkPre$Factorial$myFactorial(myFactorial.java:240)
        at myFactorial.Factorial(myFactorial.java:382)
        at myFactorial.main(myFactorial.java:24)
% jmlrac myFactorial 21
Factorial of 21 = -1195114496
% jmlrac myFactorial 32
Factorial of 32 = -2147483648
```

[8] However, research efforts have developed tools that work from JML specifications to perform various levels of formal verification. Examples of these tools are ESC/JAVA2[Flanagan 2002] and LOOP [vandenBerg 2000].

[9] To compile a Java program with JML specifications embedded, the command line `%jmlc -Q myProgram.java` is used. To run such a program, the command line `%jmlrac myProgram` is used. Interested readers should visit the Web site http://www.jmlspecs.org for a free download of the JML software and other documentation.

Table 12.6: Some of the Predefined JML Exceptions

JML Exception	**Meaning**
`JMLEntryPreconditionError`	A method call's parameters do not satisfy the method's `requires` clause.
`JMLNormalPostconditionError`	A method call exits normally, but its result does not satisfy the method's `ensures` clause.
`JMLExceptionalPostconditionError`	A method call exits abnormally, raising an exception defined by the method's `signals` clause.
`JMLLoopInvariantError`	Some execution of a loop's body does not satisfy its `loop_invariant` clause.
`JMLInvariantError`	Some call to a method or constructor does not leave the object in a state that satisfies the `invariant` clause.

In the first run, the program executes normally, with only the result and no additional errors reported. For the second run, an attempt to compute the factorial of -5 is met with a `JMLEntryPreConditionError`, which says that the call to the method `myFactorial.Factorial` violates that method's precondition `1 <= n`; it reports that the actual argument `'n' is -5`. Since this event is an instance of Java exception handling, a trace of the method calls that are active for this JML error is also provided.

The third and fourth runs show some of the vulnerability of the specifications to idiosynchrasies in Java itself. Since Java has no `ArithmeticOverflow` exception, the calculation of any `int` value that exceeds $2^{31} - 1 = 2147483647$ will give an incorrect result.[10] The largest `int` value of n for which $n! \leq 2^{31} - 1$ is 12.

Looking at the results of the third and fourth runs, we can now understand how no error was reported. That is, the `while` loop gives the same spurious result as that calculated by the JML run-time check that was specified by the postcondition. Thus, two equally incorrect answers create the illusion that all is well with this function for the arguments 21 and 32. In the next section, we revisit the handling of run-time exceptions using JML specifications.

The exception `JMLEntryPreConditionError` is just one of several types of exceptions that can occur when running JML-annotated programs. A brief description of this and other key JML exceptions is given in Table 12.6.

However, suppose the `while` loop in Figure 12.4 were changed ever so slightly

[10]For Java type `long`, the maximum value is $2^{63} - 1 = 9223372036854775807$, and for type `BigInteger` the maximum value is unlimited. So practical applications that compute factorials will likely use `BigInteger` values in order to rule out possibilities for overflow. We have avoided using the `BigInteger` class here because to do so would have introduced an enormous amount of extra baggage into the Java code, making our discussion of formal specifications almost unreadable. For this reason, we will stick with the simple type `int`.

to introduce an error in calculating the factorial. For instance, suppose we replace the line `while (i < n)` by the line `while (i <= n)`. This would raise the following error when the loop invariant is checked:

```
% jmlrac myFactorial 3
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLLoopInvariantError: LOOP INVARIANT:
   by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 9, character 24 when
        'n' is 3
        at myFactorial.internal$Factorial(myFactorial.java:102)
        at myFactorial.Factorial(myFactorial.java:575)
        at myFactorial.main(myFactorial.java:211)
```

Here, the message `'n' is 3` indicates that the resulting value of `n` does not satisfy the loop invariant specified in Figure 12.6. The information given in this message doesn't really tell the whole story. That is, it would have been useful to see the value of `i` and the loop invariant, as well as `n`, since it is that value which causes the invariant to become `False`.[11]

If we had not included the loop invariant in the program at all, the following exception would have been raised by the erroneous line `while (i <= n)`:

```
% jmlrac myFactorial 3
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLNormalPostconditionError:
   by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 4, character 23 when
   'n' is 3
   '\result' is 24
   at myFactorial.checkPost$Factorial$myFactorial(myFactorial.java:321)
   at myFactorial.Factorial(myFactorial.java:392)
   at myFactorial.main(myFactorial.java:24)
```

This message signals disagreement between the `ensures` clause and the result actually returned by the function, as reported by the line `'\result' is 24`. Here again, the programmer learns only that there is either something wrong with the specification or something wrong with the program code (or both).

A disagreement between a loop's invariant and its code could very well signal an error in the invariant and not the code. For example, if we insert an incorrect invariant into Figure 12.4, such as mistakenly writing `j <= i` as `j < i`, a loop invariant exception is raised again:

```
% jmlrac myFactorial 3
Exception in thread "main"
```

[11] In fairness, we must emphasize that JML is a work in progress at this writing. Perhaps in a later version, JML will report the values of all variables in such expressions when such an error is raised.

```
   org.jmlspecs.jmlrac.runtime.JMLLoopInvariantError: LOOP INVARIANT:
   by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 9, character 24 when
        'n' is 3
        at myFactorial.internal$Factorial(myFactorial.java:101)
        at myFactorial.Factorial(myFactorial.java:573)
        at myFactorial.main(myFactorial.java:209)
[dhcp-53-152:~/desktop/pl/correctness] allen%
```

But this time, it is the invariant that needs to be corrected and not the code.

Another benefit of run-time pre- and post-condition checking is that the programmer can slide a different implementation of a function into the program, and then test it using the same pre- and postconditions. For example, suppose we decide to implement `Factorial` recursively rather than iteratively, with the following code:

```
static int Factorial (int n) {
  if (n < 2) return n;
  else return n*Factorial(n-1);
}
```

Both of the JML `requires` and `ensures` clauses remain intact while we compile and run this version; thus its satisfaction of the preconditions and postconditions can be immediately tested.

JML Exception Handling

Formal methods for program correctness should support the specification of conditions under which exceptions occur. To that end, JML provides a `signals` clause:

```
signals (exception) expression ;
```

that can appear together with a function's `requires` and `ensures` clauses. When that `exception` occurs, the `expression` is checked; if that `expression` is not true, the `exception` is displayed and the program is interrupted. Figure 12.6 shows a variant of the `Factorial` function that incorporates these ideas:

Now when we run this program to compute the factorial of a number that will cause arithmetic overflow, an exception is raised:

```
% jmlrac myFactorial 13
Exception in thread "main" java.lang.ArithmeticException
        at myFactorial.internal$Factorial(myFactorial.java:9)
        at myFactorial.Factorial(myFactorial.java:610)
        at myFactorial.main(myFactorial.java:213)
```

Observant readers will notice that `signals` clauses can be avoided in many cases simply by writing stronger preconditions – ones that so constrain the input to the call that the exception cannot occur. For instance, in the `Factorial`

```
/*@ requires 1 <= n;
    ensures \result == (\product int i; 1<=i && i<=n; i);
    signals (ArithmeticException) n > 12;
  @*/
static int Factorial (int n) {
  if (n > 12) throw new ArithmeticException();
  else {
    int f = 1;
    int i = 1;
  /*@ loop_invariant i <= n &&
          f == (\product int j; 1 <= j && j <= i; j);
    @*/
    while (i < n) {
      i = i + 1;
      f = f * i;
    }
    return f;
  }
}
```

Figure 12.7: Adding Exception handling to a JML specification

function, we can just as easily replace the `signals` clause by the following enhancement to the `requires` clause:

```
requires 1 <= n && n < 13;
```

Now the call `Factorial(13)` will raise a `JMLEntryPrecondition` error rather than a Java `ArithmeticException` error.

In more complex software design situations, the need for JML specifications to signal exceptions explicitly is more compelling than in this simple example. For example, consider the task of defining complete JML specifications for all the classes and methods in the Java class library. Among these methods is the `substring` method, whose informal and formal specifications were discussed earlier in this chapter (see Figure 12.4).

Below is a JML-style rendition of those formal specifications, with `signals` clauses added to describe actions when the preconditions are not met by the call.

```
/*@ requires 0 <= beginIndex && beginIndex < endIndex &&
        endIndex <= s.length();
    ensures \result ==
        "s[beginIndex] s[beginIndex+1]... s[endIndex-1]";
    signals (StringIndexOutOfBoundsException)
        0 > beginIndex || beginIndex >= endIndex ||
        endIndex > s.length();
  @*/
```

```
s.substring(intbeginIndex, intendIndex)
    throws StringIndexOutOfBoundsException
```

This specification is a slight abbreviation of the full JML specifications for `substring` that define the entire Java class library. In particular, the line `"s[beginIndex] s[beginIndex+1]...s[endIndex-1]"` is our informal algebraic specification for the value of the string object that is returned by `substring`. In fact, that line appears in the JML specification as:

```
this.stringSeq.subsequence(beginIndex,endIndex);
```

Here, `stringSeq` is a JML class that defines strings as sequences of characters, and `subsequence` is a method in that class. Interested readers should consult `http://www.jmlspecs.org` for more details about these conventions.

The next section considers the use of formal methods and correctness concepts within an object-oriented framework.

12.4 CORRECTNESS OF OBJECT-ORIENTED PROGRAMS

Object-oriented programs are collections of classes. Each class defines a kind of object and a set of features (methods) that can transform that kind of object. When implementing an object-oriented program, the programmer needs to have standards, or tools by which s/he can formally verify that the entire program is doing what it is intended to do. In addition to the formalization of individual functions, as described in the foregoing section, object-oriented programs provide two additional tools by which programmers can ensure correctness.

First, each interaction in which an object in one class is accessed or modified by a method call from a so-called *client* class, must be guided by certain "rules of engagement." These rules ensure that the client provides the called method with appropriate values for its parameters, and that the called method returns a result to the client that is consistent with the purpose of the method. These rules of engagement are called a *contract* between the class and the client. When all the interactions among classes in a software design follows these rules, the software is said to have been *designed by contract.*

Second, whenever an object in a class is transformed, it must maintain the same internal consistency, or set of properties that identifies it as a member of that particular class, that it had when it was created. This internal consistency can be formally characterized as the *class invariant.*

We define and illustrate the use of design by contract and the class invariant in the next two sections.

12.4.1 Design By Contract

The methodology of design by contract was developed by Bertrand Meyer [Meyer 1990]. Design by contract is a formal framework that establishes so-

called obligations and benefits (responsibilities and rights) between each supplier/client pair that occur during the life of a software product. This framework helps software designers to document constraints that appear at the interface between interacting classes, and to effectively assign blame when things go wrong.

The structure of the design by contract framework is a 2x2 matrix that identifies obligations and benefits in one dimension and client and supplier classes in the other, as shown in Table 12.6. Typically, the client is about to call a public method that is implemented within the supplier class. At that time, the client is obligated to pass parameters that satisfy the method's precondition. If that is done, the client can be confident that a result will be computed and work will be accomplished. If, on the other hand, the client's parameters do not satisfy the method's precondition, no confidence can be had for the veracity of the result. If this event causes an error, then blame for the error is squarely in the lap of the client.

For the same call, the supplier has the obligation to compute a result that satisfies the method's postcondition, and to leave the object in a state in which the so-called *class invariant* satisfied (see below for a discussion of the class invariant). The supplier benefits from the contract by not having to include code that explicitly checks that the argument satisfies the precondition. If an error occurs and/or the postcondition is not met, the blame is squarely in the lap of the supplier.

Let's look at a simple example of how the design by contract framework plays out for the factorial computation discussed in the foregoing section.

	Obligations	**Benefits**
Client	pass $n > 0$	receive $n!$ computed
Supplier	compute $n!$	can assume that $n > 0$

If either the client or the supplier fails to meet its obligations, it becomes responsible for any error that occurs during the call. If a precondition error occurs, the client is responsible for the error. If a postcondition error occurs, the supplier is responsible.

If neither type of error occurs during the call, then both the client and the supplier receive appropriate benefits. Specifically, the client can be confident that the value $n!$ is returned correctly, and the supplier need not explicitly check for $n > 0$ within its own code.

Table 12.7: The Design by Contract Framework

	Obligations	**Benefits**
Client	satisfy precondition	result is computed
Supplier	satisfy postcondition	simpler coding

12.4.2 The Class Invariant

A class invariant is a tool for ensuring that all objects in the class retain their integrity throughout their lifetime, no matter what methods are applied to them. In discussing class invariants, we follow the approach of [Meyer 1997]. We illustrate this approach with the formalization of the `MyStack` class that was originally introduced in Chapter 7 (Figure 7.7).

A *class invariant* is a Boolean-valued expression that specifies the conditions under which an object in that class remains well-defined. This expression describes the *internal state* of the object using the class's public and private instance variables. An expression *INV* is a correct class invariant for class *C* if it meets the following two conditions:

- Every call to a constructor *C* with arguments that satisfy *C*'s precondition, creates a new object with a state that satisfies *INV*.
- Every call to a public method *M* with arguments that satisfy *M*'s precondition, leaves the object in a state that satisfies *INV*.

Thus, the class invariant must become *true* when the object is created by a constructor, and it must remain *true* after any public method in the class is called. During the execution of the code inside a call, the class invariant may be temporarily broken; however, such a condition must be repaired by the time the call is completed.

For example, consider the class `MyStack` in Figure 7.7, which we have recreated and expanded in Figure 12.9. This new version has a class invariant, an additional private instance variable `n`, three new public methods, and appropriate pre- and postconditions added to all methods. The new private instance variable `n` is a count of the number of elements in the stack, and the instance variable `theStack` is a reference to the topmost element in the stack.

For the time being, let us concentrate on the specification of the class invariant, which has the following general form in JML:

```
public invariant expression ;
```

The invariant is checked automatically by JML each time a constructor or method call is entered or exited, and a `JMLInvariantError` exception is raised whenever the invariant's `expression` is not *true*.

Consider the JML specification for the invariant for the class *myStack* in Figure 12.9:

```
/*@ public model Node S;
    private represents S <- theStack;
    public invariant S == null || n == this.size();
  @*/
private /*@ spec_public @*/ Node theStack = null;
private /*@ spec_public @*/ int n = 0;
```

Here, we use a so-called *model variable* S, which is known only to the JML specifications and has no run-time functionality within the Java program. The

purpose of a model variable is to facilitate writing JML specifications that are free from the details of the class implementation.

Our use of the model variable S frees the JML specifications from explicitly mentioning the variable `theStack` which is an implementation-dependent name. The clause that begins `private represents` identifies the relationship between the model variable and the actual variable that it represents.

The other JML clause that appears here is the `spec_public` clause. This clause allows a `private` variable to be treated as a public variable by the JML specifications themselves. Thus, a `private` variable's value can be accessed whenever it is mentioned by a JML specification, while it remains unaccessible to any client of the class at run time.

12.4.3 Example: Correctness of a Stack Application

As suggested above, any method in a class can be augmented with pre- and postconditions that constrain the range of values that its arguments and results can have. Consider, for example, the method `pop` that appears in the class `MyStack` in Figure 7.7:

```
public int pop( ) {
    int result = theStack.val;
    theStack = theStack.next;
    return result;
}
```

What would be an appropriate precondition for this method? That is, what would be required in order for the `pop` method to be able to complete its task successfully? Minimally, we would not want a `pop` method ever to be called when the stack is empty. That is, the references in the above code to `theStack.val` and `theStack.next` are meaningful only if the value of `theStack` is not `null` at the time of the call. So a good precondition for `pop` would require that $n > 0$.

What would we like to see for a postcondition? Informally, we know that the top element of the stack is removed and returned by this operation, so that the resulting stack might become empty. Beyond that, we can expect that the variable `n` will be decremented by 1 to signify removal of the top element.

These pre- and postconditions are formalized by adding JML specifications to `pop` as shown in Figure 12.8. There, we see a use of the special JML function `\old` to specify the *a priori* value of `theStack` and `n` upon entry to the `pop` method. Thus, the final value of a variable can be specified as a function of its value at the beginning of the call.

The JML specification `/*@ pure @*/` is used for any method that has no non-local side-effects and is provably non-looping. That is, pure methods override, in some sense, the notion of partial correctness discussed in an earlier section. In order for a method to be used in the JML specification of the class invariant, it must be a `pure` method. This is the case, for instance, for the methods `push`, `pop`, and `top` in the `MyStack` class.

Using these conventions, a complete set of specifications for the methods in the `MyStack` class are shown in Figure 12.9. This includes pre- and postconditions for the `push`, `pop`, `top`, `isEmpty`, and `size` methods, the class invariant, and the use of the model variable S throughout.

Testing the Contract

Annotating the `MyStack` class with pre- and postconditions and a class invariant provides an executable environment in which the contract between the class and its clients can be continuously tested. Moreover, these annotations provide a mechanism for assigning blame when the contract is broken by the class or its client.

To illustrate this testing activity, we wrote the simple driver program shown in Figure 12.10 that can exercise the methods of the `MyStack` class. The following command was used to run the program.

```
% jmlrac myStackTest 3 4 5 6
Stack top = 6
Is Stack empty? false
Stack size = 3
Stack contents =
6
5
4
Is Stack empty now? true
```

The first parameter counts the number of values to be pushed onto the stack, and the remaining parameters provide those values. The normal output produced by this program follows the command.

In order to exercise various aspects of the contract between the class and its client, we then ran three different tests.

```
/*@ requires n > 0;
    ensures \result==\old(S).val &&
            S==\old(S).next && n==\old(n)-1;
  @*/
  public /*@ pure @*/ int pop( ) {
      int result = theStack.val;
      theStack = theStack.next;
      n = n-1;
      return result;
  }
```

Figure 12.8: Stack `pop` Method with Specifications Added

```
public class MyStack {
    private class Node {
        /*@ spec_public @*/ int  val;
        /*@ spec_public @*/ Node next;
        Node(int v, Node n) {
            val = v; next = n;
        }
    }
    /*@ public model Node S;
        private represents S <- theStack;
        public invariant n == this.size();
    @*/
    private /*@ spec_public @*/ Node theStack = null;
    private /*@ spec_public @*/ int n = 0;

    /*@ requires n > 0;
        ensures \result==\old(S).val && S==\old(S).next;
    @*/
    public /*@ pure @*/ int pop( ) {
        int result = theStack.val;
        theStack = theStack.next;
        n = n-1;
        return result;
    }
    //@ ensures S.next==\old(S) && S.val==v;
    public /*@ pure @*/ void push(int v) {
        theStack = new Node(v, theStack);
        n = n+1;
    }
    /*@ requires n > 0;
        ensures \result==S.val && S == \old(S);
    @*/
    public /*@ pure @*/ int top() {
        return theStack.val;
    }
    //@ ensures \result == (S == null);
    public /*@ pure @*/ boolean isEmpty() {
        return theStack == null;
    }
    //@ ensures \result == n;
    public /*@ pure @*/ int size() {
        int count;
        Node p = theStack;
        for (count=0; p!=null; count++)
            p = p.next;
        return count;
    }
}
```

Figure 12.9: A Fully Specified Stack Class using JML

```
public class myStackTest {
public static void main(String[] args) {
   MyStack s = new MyStack();
   int val;
   int n = Integer.parseInt(args[0]);
   for (int i=1; i<=n; i++)
      s.push(Integer.parseInt(args[i]));
   System.out.println("Stack top = " + s.top());
   System.out.println("Is Stack empty? " + s.isEmpty());
   System.out.println("Stack size = " + s.size());
   System.out.println("Stack contents = ");
   for (int i=1; i<=n; i++) {
      System.out.println(s.top());
      s.pop();
   }
   System.out.println("Is Stack empty now? " + s.isEmpty());
}
}
```

Figure 12.10: A Driver Program for Testing the JML-Specified MyStack Class.

The first test, whose results are shown below, illustrates what happens when the `top` method erroneously removes the top element as well as returning it – i.e., it incorrectly acts like a `pop`.

```
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLNormalPostconditionError:
   by method MyStack.top regarding specifications at
File "MyStack.java", line 31, character 26 when
        '\old(S)' is MyStack$Node@5ff48b
        '\result' is 5
        'this' is MyStack@affc70
        at MyStack.checkPost$top$MyStack(MyStack.java:999)
        at MyStack.top(MyStack.java:1078)
        at myStackTest.main(MyStackTest.java:15)
```

To trigger this error, we added two extra lines to the `top` method in Figure 12.9 so that its body looked like that of the `pop` method. Since `top`'s result now failed to satisfy its postcondition `S == \old(S)`, a `JMLNormalPostconditionError` was raised and the values of `\old(S)` and `\result` were reported by JML. With this information, blame for the error could be assigned to to the `top` method rather than to its caller.

For the second test, we excluded the line `n=n-1;` from the `pop` method shown in Figure 12.9, thus creating a situation in which the method's postcondition was satisfied but the class invariant was violated. Below is the outcome.

```
Stack top = 6
Is Stack empty? false
Stack size = 3
Stack contents =
6
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLInvariantError:
   by method MyStack.pop@post<File "MyStack.java", line 16, character 17>
   regarding specifications at
File "MyStack.java", line 11, character 30 when
        'this' is MyStack@9664a1
        at MyStack.checkInv$instance$MyStack(MyStack.java:102)
        at MyStack.pop(MyStack.java:525)
        at myStackTest.main(MyStackTest.java:21)
```

Here, we see that the program began to run normally, but when the first call to the `pop` method was completed, the JML test of the class invariant `n==this.size();` failed. This caused the `JMLInvariantError` to be displayed, along with some useful tracing information. In order to check satisfaction of this invariant, JML had to call the method `this.size()`, which recomputes the size of the stack independently by traversing its linked list representation.

The third test was created by adding a new final line `s.pop();` to the driver program so that it would try to pop an element from an empty stack. Here is the result.

```
Stack top = 6
Is Stack empty? false
Stack size = 3
Stack contents =
6
5
4
Is Stack empty now? true
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError:
   by method MyStack.pop regarding specifications at
File "MyStack.java", line 16, character 21 when
        'this' is MyStack@9664a1
        at MyStack.checkPre$pop$MyStack(MyStack.java:330)
        at MyStack.pop(MyStack.java:479)
        at myStackTest.main(MyStacktest.java:24)
```

This run reports a `JMLEntryPreconditionError`, indicating that the precondition `n > 0` for `pop` was violated. Thus, the blame for this error lies with the caller rather than with the `pop` method. The presence of an actively-checked precondition protects the method itself from having to engage in "defensive programming," which would be necessary if the precondition were not there.

Correctness of the MyStack Class

What about the correctness of the `MyStack` class? Annotating it with pre- and postconditions and a class invariant, and then testing the contract with a driver program, surely doesn't guarantee correctness in a formal sense.

Informally, a class is *correct* if, for every object in the class and every constructor or method call that satisfies its precondition, completion of the call satisfies its postcondition and leaves the object's instance variables in a state that satisfies the class invariant. This idea assumes that no constructor or method call will result in an infinite loop, and thus it is a statement about partial correctness.

Let's try for a more formalized definition of class correctness,[12] using the notation of Hoare triples that we introduced at the beginning of the chapter. Let R denote a class's invariant, and P_i and Q_i denote the precondition and postcondition for its i^{th} constructor C_i or method M_i. Then we can say that a class is *correct* with respect to its assertions if both:

1. For every set of valid arguments x to every constructor C_i, $\{P_i(x)\}C_i.body\{Q_i(x) \wedge INV\}$, and

2. For every set of valid arguments x to every method M_i, $\{P_i(x) \wedge INV\}M_i.body\{Q_i(x) \wedge INV\}$.

Rule 1 says, in effect, that execution of any constructor for an object in the class should establish the validity of the class invariant. Rule 2 says that execution of any method call in which the class invariant is valid at the outset should preserve the validity of the invariant upon completion of the call. Thus, this definition requires us to prove the correctness of every constructor and public method individually. Each such proof is conducted in a way similar to that developed for the `Factorial` method in Section 12.3.2.

Let us illustrate these ideas by developing some of the proof that our linked list implementation of the `MyStack` class is formally correct.

First, we note that the default class constructor `MyStack()` establishes the instance variables' values `theStack==null` and `n==0`. The class invariant *INV* itself is stated in terms of the public method `size`. Verification of the body of the `size` method with this invariant established should therefore satisfy its postcondition and preserve the invariant. Formally, we want to prove:

$$\{theStack = null \wedge n = 0\}\text{size.body}\{theStack = null \wedge n = 0 \wedge \backslash result = n\}$$

Informally, we note that the local variables `p` and `count` for `size` are initialized at `null` and 0, respectively. So the loop is not executed at all and the result 0 is returned, establishing the validity of the above Hoare triple. (A formal proof would have more rigorously applied all the steps of the proof method described in Section 12.3.2; we have short-circuited that process here in order to keep our main focus on class verification.)

[12] This formalization is adapted from [Meyer 1997].

Now we need to verify all the methods in the class. Let's illustrate this by verifying the `pop` method. That is, we need to establish the validity of the following Hoare triple:

$\{n > 0 \wedge n = this.size()\}$
pop.body
$\{ \backslash result = \backslash old(S).val \wedge S = \backslash old(S).next \wedge n = this.size()\}$

The precondition `n > 0` states that the stack cannot be empty; that is equivalent to requiring that `S` not be `null`.

The series of assignments in `pop.body` lead directly to the validity of the postcondition. That is, the first element is removed from the linked list and the resulting list is returned. More precisely, note that the postcondition for `pop` specifies its effect in terms of the model variable S, which is a surrogate for the instance variable `theStack`. Since `\old(S)` identifies the value of `S` at the beginning of the call to `pop`, the whole expression $S = \backslash old(S).next$ asserts that the resulting stack S is identical to the input stack with the first element removed. Concurrently, the value of instance variable `n` is decremented, so as to preserve the validity of the invariant $n = this.size$.

Finally, the correctness of the `MyStack` class depends implicitly on the assumption for correctness of the `Node` class. In a formal verification setting, the `Node` class would need to be formally specified and proved correct as well.

Final Observations

We have sketched enough of the formal correctness procedures for a reasonably complex class to suggest that proof of correctness of any substantially large program is tedious (at best), and maybe totally useless in practice. What are the prospects for the effective use of formal methods in software design?

First, there are many who believe that other software design techniques may have more promise for solving the current software crisis than formal methods. For instance, the so-called capability maturity model (CMM) [SEI 2004] focuses on the refinement of software management processes as the key to improving software quality.

Second, it is also true that formal methods have been effectively used to verify components of safety-critical software products. For instance, a secure certification authority for smart cards was developed by Praxis Critical Systems [Hall 2002], using formal methods to establish correctness of the system's critical security properties. There are many other examples of the effective use of formal methods in the practice of software design.

Third, the cost of developing a system that is provably correct is high, relative to the cost of using traditional test-and-debug methods. Moreover, most programmers are not well trained in the use of mathematical logic to reason about their programs. They would need to be trained in logic and the use of advanced software tools that assist with formal verification, such as the LOOP tool discussed earlier in the chapter.

In summary, we conclude that the community of interest in developing better formal methods for software design has gained substantial momentum in the recent past. Surely the use of formal methods by itself is no panacea for the software crisis, but it does provide a level or rigor for the software design process that is badly needed. For that reason alone, we expect that more programming language tools like JML, ESC/JAVA, and LOOP will continue to evolve and make their impact on the software design process in the future.

12.5 CORRECTNESS OF FUNCTIONAL PROGRAMS

This section addresses the question of program correctness from the point of view of functional programming. We revisit the question of what makes a program correct for the special case when it is written in a pure functional program – one that is state-less and relies instead on functional composition and recursion as a foundation for its semantics.

The first section below illustrates this process by making a strong connection between a recursive function and an inductive proof of its correctness. The second section provides three additional examples, paying particular attention to the use of structural induction – that is, an induction on data structures like lists and strings, rather than on the integers.

12.5.1 Recursion and Induction

When considering the question of correctness for programs written in a functional language, such as Haskell, we find ourselves in a very different place. First, absent the notion of program state and assignment in pure functional programs, we need not write Hoare triples to keep track of the state transformations as we would with programs written in imperative and object-oriented languages.

Instead, functional programs are written as collections of functions that are well grounded in the mathematics of functions and recurrence relations. This allows us to base correctness proofs for Haskell functions on the well-worn technique of mathematical induction, rather than direct proofs that rely on reasoning about state transformations at every step. Overall, the verification of functional programs is a much more straightforward process than the verification of imperative and object-oriented programs.

For a simple example, consider the Haskell function that computes the factorial of a nonnegative integer `n`:

```
>   fact n
>       | n == 1  = 1              -- fact.1  (basis step)
>       | n > 1   = n*fact(n-1)    -- fact.2  (induction step)
```

Suppose we want to prove that this function computes the product of the first n nonnegative integers, given n. That is, we want to prove that:

$$fact(1) = 1$$
$$fact(n) = 1 \times 2 \times \ldots \times (n-1) \times n \quad \text{when } n > 1$$

For an inductive proof, recall that we need to show both of the following:

1. (Basis step) That the function computes the correct result for $n = 1$.
2. (Induction step) Assuming the hypothesis that the function computes the correct result for some integer $n = k-1$, we can conclude that the function computes the correct result for the next integer $n = k$.

Since the function `fact` is recursively defined, its guarded commands naturally delineate the basis step from the induction step, as indicated by the comments on the right. So the basis step is handled by the first line of the function definition and the induction step is handled by the second.

The function definition satisfies the basis step by observation. That is, when $n = 1$ we have `fact(1) = 1`, using the line annotated `fact.1`.

For the induction step, assume that $n > 1$ and $fact(n-1) = 1 \times 2 \times \ldots \times (n-1)$. Then correctness is established for `fact(n)` by using the line annotated `fact.2` and the hypothesis, followed by an algebraic simplification:

```
fact(n) = n*fact(n-1)
        = n*(1 * 2 * ...  * (n-1))
        = 1 * 2 * ...  * (n-1) * n
```

From this particular example, readers should notice the relative ease with which the correctness of a program in a functional language can be proved in contrast with that of its counterpart in an imperative language. The latter's bulky Hoare triples and direct proof techniques are replaced by a straightforward induction process in which the function's definition directly mirrors the proof.

12.5.2 Examples of Structural Induction

An induction strategy can be used to prove properties of Haskell functions that operate on lists and strings. Induction on list-processing and string-processing functions is often called "structural induction" because it simplifies the structure (size) of a list or string as it defines the hypothesis and shows the validity of the induction step.

This section provides examples of induction proofs for various Haskell functions involving list concatenation, reversal, and length. Because a Haskell string is a list of characters, these proofs apply to strings as well as lists.

List Reversal and Concatenation

Consider the following functions defined for list concatenation and reversal (these mirror the standard Haskell functions `++` and `reverse`, respectively):

```
>  cat [] ys     = ys                 -- cat.1
>  cat (x:xs) ys = x : (cat xs ys)    -- cat.2
>
>  rev []     = []                    -- rev.1
>  rev (x:xs) = cat (rev (xs)) [x]    -- rev.2
```

Suppose we want to prove the following property about the relationship between these two functions:

```
rev (cat xs ys) = cat (rev ys) (rev xs)
```

For instance, if the two lists (strings) are "hello" and "world," then the following is true:

```
rev (cat "hello " "world")
     = cat (rev "world")  (rev "hello ")
     = "dlrow olleh"
```

To prove this property by induction, we begin with the basis step and use the definitions of these two functions. So we first need to show that:

```
rev ([] ++ ys) = rev (ys) ++ rev ([])
```

Using various lines in the definitions of these functions, we prove this by substitution as follows (justifications for each step are shown on the right):

```
rev (cat [] ys)} = rev (ys)                        (from cat.1)
                 = cat (rev (ys) [])               (from rev.2)
                 = cat (rev (ys) rev [])           (from rev.1)
```

The induction hypothesis for this proof is written by stating the conclusion for any two lists xs and ys.

```
rev (cat xs ys) = cat reverse(ys) reverse(xs)
```

Now the induction step can be completed by showing how a slightly longer (by 1 element) list x:xs obeys the same rule, as follows:

```
rev (cat (x:xs) ys) = cat (rev ys) (rev (x:xs))
```

Here, we transform the left-hand side of this expression using our hypothesis and various lines in the definitions of the functions `rev` and `cat`, to achieve the following:

```
rev (cat (x:xs) ys) = rev (x : (cat xs ys))  (from cat.2)
     = rev (cat (cat xs ys) [x])             (from rev.2)
     = cat (cat (rev ys) (rev xs)) [x]       (from our hypothesis)
     = cat (rev ys) (cat (rev xs) [x])       (associativity of cat)
     = cat (rev ys) (rev (x:xs))             (from rev.2)
```

Finally, notice that the fourth line in this derivation assumes associativity for the operator `cat`, which can be separately proved by induction. This is left as an exercise.

List Length and Concatenation

Consider the following Haskell function, which explicitly computes the length of a list. Because this is predefined in Haskell as `length`, we redefine it here with a slightly different name. Again, the comments on the right will be used in proofs about the properties of this function.

```
> len []     = 0                      -- len.1
> len (x:xs) = 1 + (length xs)        -- len.2
```

For this function, the first line defines the length, 0, of an empty list and the second shows how to compute the length of a list based on the known length of a list slightly smaller than it. For example,

```
len [1,3,4,7]
   = 1 + len [3,4,7]
   = 1 + (1 + len [4,7])
   = 1 + (1 + (1 + len [7]))
   = 1 + (1 + (1 + (1 + len [])))
   = 1 + (1 + (1 + (1 + 0)))
   = 4
```

The first four calls use the second line of the `len` function, while the fifth call uses the first line.

Here is an inductive proof that the length of two concatenated strings is identical to the sum of their individual lengths. I.e.:

```
len (cat xs ys) = len xs + len ys
```

Notice in this proof that a familiar pattern is used: the basis step uses the first line in the recursive definition, and the induction step in the proof uses the second line. This proof provides another example of structural induction.

For the basis step, we need to show that:

```
len (cat [] ys) = len [] + len ys
```

This is done by the following two lines:

```
len (cat [] ys) = len ys              by cat.1
        = 0 + len ys                  by arithmetic
        = len [] + len ys             by len.1
```

For the inductive step, we assume the hypothesis is true for arbitrary strings xs and ys

```
len (cat xs ys) = len xs + len ys for some lists xs and ys.
```

Now let's see what happens when we add an additional character to the first string.

```
len (cat x:xs ys) = len x: (cat xs ys)  by cat.2
        = 1 + len (cat xs ys)           by len.2
        = 1 + len xs + len ys           by hypothesis
        = len x:xs + len ys             by len.2
```

This completes the proof.

As these examples illustrate, Haskell provides especially strong support for correctness proofs in complex software systems. While, unfortunately, not a large number of software systems are implemented in Haskell, those that are enjoy a generally high level of reliability.

However, functional languages like Haskell are being considered more and more seriously by software designers as vehicles for defining precise specifications for software prototypes. Conventional languages like C++ and Ada have been inadequate for this purpose [Hudak 1994].

EXERCISES

1. Suggest a different way to write the function `Max(a, b)` in Figure 12.1 without changing the meaning of the function.

2. Below is a Hoare triple that includes a C++Lite program fragment to compute the product `z` of two integers `x` and `y`.

   $\{y \geq 0\}$

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

   $\{z = x \times y\}$

   (a) What inference rules in Table 3.1 and additional knowledge about algebra can be used to infer that the precondition in this Hoare triple is equivalent to the assertion $\{y \geq 0 \wedge 0 = x(y - y)\}$?

   (b) Using the assignment inference rule, complete the following Hoare triple for the first two statements in this program fragment:

   $\{y \geq 0 \wedge 0 = x(y - y)\}$

   ```
   z = 0;
   n = y;
   ```

   $\{y \geq 0 \wedge \qquad\qquad\qquad\qquad \}$

(c) Explain how the following can be an invariant for the while loop in this program fragment.

$\{y \geq 0 \wedge n \geq 0 \wedge z = x(y - n)\}$

That is, why is this assertion *true* before execution of the first statement in the loop, and why must it be *true* before execution of every successive repetition of the loop?

(d) Show that this invariant holds for a single pass through the loop's statements.

(e) Using the inference rule for loops, show how the invariant is resolved to an assertion that implies the validity of the postcondition for the entire program.

3. The following C++ function approximates the square root of `x` to within a small error `epsilon`, using Newton's method.

```
float mySqrt (float x, float epsilon) {
    float a, result;
    a = 4.0;
    x = 1.0;
    while (x*x > a+epsilon || x*x < a-epsilon)
        x = (x + a/x)/2.0;
        result = x;
}
```

(a) Describe a precondition and a postcondition, P and Q, that would serve as appropriate formal specifications for this function.

(b) Describe a loop invariant that would serve to describe the loop in this function.

(c) Are there any special circumstances under which a call to this function will not terminate or satisfy its postcondition? Explain.

(d) Prove the (partial) correctness of this function.

4. Suppose the function in the previous exercise were part of a Java class that supported mathematical functions. Describe a contract that would be appropriate for any client of that class with regard to their use of the `mySqrt` function. Give JML `requires` and `ensures` clauses that will document this contract.

5. Write a Java function that computes the sum of a series of integers stored as a linked list (similar to the class `MyStack` and `Node` discussed in this chapter. Write pre- and postconditions for this function, and then develop a proof of its correctness using the inference rules in Table 3.1.

6. In the spirit of designing software from formal specifications, find a precise English-language definition at the Java Web site for the method `indexOf(String)` in the class `java.lang.String`.

(a) Translate that definition to a formal pre- and postcondition.

(b) Now translate your specification into JML `requires` and `ensures` clauses.

7. Give a recursive C/C++ implementation of the function *Factorial* in Figure 12.3. Prove the partial correctness of your recursive implementation for all values of $n > 0$. *Note*: to prove the correctness of a recursive function, induction must be used. That is, the base case and recursive call in the function definition correspond with the basis step and induction step in the proof.

8. A program has *total correctness* if it (completes its execution and) satisfies its postcondition for *all* input values specified in its precondition. Suppose we altered the function `Factorial` in Figure 12.3 so that its argument and result types are `long` rather than `int`.

   (a) Experimentally determine the largest value of `n` for which your altered version of `Factorial` will deliver a result. What happens when it does not?

   (b) Refine the precondition for this version of *Factorial* so that its correctness proof becomes a proof of total correctness.

   (c) How is the correctness proof itself altered by these changes, if at all? Explain.

9. Alter the JML version of the `Factorial` function definition in Figure 11.6 so that its argument and result types are `long` rather than `int`. Add exception generating capabilities to this function so that it raises an `ArithmeticError` exception whenever the factorial cannot be correctly computed. Finally, add a JML `signals` clause to the specification that covers this event.

10. Reimplement the `Factorial` function so that it returns a value of type `BigInteger`. In what ways is this implementation superior to the version presented in Figure 12.6?

11. Reimplement the `Factorial` function in Haskell. In what ways is this implementation superior to the Java variations in Figure 12.6 and the previous question? In what ways is it inferior?

12. Give an induction proof for the correctness of your Haskell implementation of the `Factorial` function in the previous exercise. For this, you should rely on the mathematical definition of factorial.

13. Discuss the tradeoffs that exist between the choices of refining the precondition and adding a `signals` clause when specifying a function's response to an input value for which it cannot compute a meaningful result. E.g., these choices are illustrated in the foregoing two exercises.

14. Consider the correctness of the class `MyStack` in Section 12.4.

    (a) The method `size` was verified in consort with initialization of an object using the class constructor. Write an appropriate Hoare triple and then verify the method `size` for all other states that the object may have.

    (b) Write an appropriate Hoare triple and then verify the method `push`.

15. Prove by induction that the Haskell operator ++ is associative, using its recursive definition named `cat` that is given in this chapter. That is, show that for all lists xs, ys, and zs:

    ```
    cat (cat xs ys) zs = cat xs (cat ys zs)
    ```

    I.e., this is equivalent to `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`.

16. Given the definition of the Haskell function `len` in this chapter, prove by induction the following:

    ```
    len (reverse xs) = len xs
    ```

17. Consider the following (correct, but inefficient) Haskell implementation of the familiar Fibonacci function:

    ```
    > fibSlow n
    >   | n == 0  = 1                  -- fib.1
    >   | n == 1  = 1                  -- fib.2
    >   | otherwise   = fibSlow(n-1) + fibSlow(n-2)   -- fib.3
    ```

    The correctness of this function can be proved quickly, since it is a direct transcription of the familiar mathematical definition below, and since the Haskell type Integer is an infinite set:

    $$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \quad \text{if } n \geq 2 \end{aligned}$$

    Give an induction proof of correctness for `fibSlow`.

18. As sugggested by its name, the effeciency of the `fibSlow` function in the previous exercise is suspect.

    (a) Try running `fibSlow(25)` and then `fibSlow(50)` on your system and see how long these computations take. What causes this inefficiency?

(b) An alternative definition of the Fibonacci calculation can be made in the following way. Define a function `fibPair` that generates a 2-element pair that contains the nth Fibonacci number and its successor. Define another function `fibNext` that generates the next such tuple from the current one. Then the Fibonacci function itself, optimistically named `fibFast`, can be defined by selecting the first member of the nth `fibPair`. In Haskell, this is written as follows:

```
> fibPair n
>    | n == 0   = (1,1)
>    | n > 0    = fibNext(fibPair(n-1))
> fibNext (m,n) = (n,m+n)
> fibFast n = fst(fibPair(n))
```

Try running the function `fibFast` to compute the 25th and 50th Fibonacci numbers. It should be considerably more efficient than `fibSlow`. Explain.

(c) Prove by induction that $\forall n \geq 0 : \mathit{fibFast}(n) = \mathit{fibSlow}(n)$.