In our syntax, an **Assume** statement will be used to record what must be true at the beginning of the program in order for it to work correctly. Similarly, a **Confirm** statement is used to record what should be true after the code is executed.

Assume $y \neq 0$**;**
$z := w/y;$
if $z \geq 0$ **then** abs := z
else abs := -z
endif;
Confirm abs = $|w/y|$;

This program computes a real quotient, so we must be sure that the divisor *y* is nonzero. The **Assume** clause accomplishes this. The **Confirm** statement claims that, upon execution of the code, the value of the variable *abs* will be the absolute value of the quotient *w/y*.

We can see that the **Assume** and **Confirm** clauses together serve to specify what the program does, by first screening out unsatisfactory input and finally by stating what will be true after program execution. In addition to providing formal specifications, these assertions permit verification by forming a basis for developing appropriate proof rules.

In the absolute value program, the constructs used are the **If then else** and the assignment statements, and so we need proof rules for these constructs and an explanation of what they mean. The form of typical proof rules is illustrated by the rule for **if then else** statements:

code: **Assume** B; code1; **Confirm** Q;
code; **Assume** $\neg$ B; code2; **Confirm** Q;
---
code; **If B then** code1 **else** code2; **endif**; **Confirm** Q;

The meaning of this rule (and of all future rules) is that the correctness of the bottom line can be deduced from the correctness of the top lines. The *code* at the

beginning of each of the above lines is a sequence of statements, the first of which is usually an **Assume** statement. Similarly, *code1* and *code2* also represent sequences of statements.

To illustrate this rule we apply it to the absolute value example, but first we need a basic understanding of the overall proof process. In general, we will have a proof rule to cover each different kind of statement in our programming language, and our proof construction process will involve the creation of a succession of lemmas. This process begins with the statement immediately preceding the final **Confirm** statement and progresses backward through the code, applying the appropriate rule at each point. More will be said about this order of rule application later.

In our example, the code preceding the **If then else** statement consists of an **Assume** statement and an assignment. Applying the **If then else** rule backwards yields two assertive programs which must then be proved correct:

(1) **Assume** $y \neq 0$**;**
z := w/y;
Assume $z \geq 0$**;**
abs := z;
Confirm abs = $|w/y|$;

(2) **Assume** $y \neq 0$**;**
z := w/y**;**
Assume $\neg(z \geq 0)$**;**
abs := -z;
Confirm abs = $|w/y|$;

We note that when we applied this rule, the **If then else** construct itself disappeared, having been replaced by two hypotheses, each containing fewer programming constructs than the original program. Since both these hypotheses contain assignment statements, we look next at the proof rule for assignments:

code; **Confirm** $Q[x \to \exp]$**;**

---

code; x := exp; **Confirm** Q;

As usual, the meaning of the rule is that in order to prove the bottom line, it is sufficient to prove the top line. The symbol "$\to$" can be read as "replaced by." This rule says that we omit the assignment statement and rewrite the **Confirm** clause *Q* replacing all the instances of *x* by the expression *exp*, which was to have been assigned to *x*.

Applying this rule to our absolute value proof, we get:

(1) **Assume** $y \neq 0$;
z := w/y;
Assume $z \geq 0$;
Confirm $z = |w/y|$;

(2) **Assume** $y \neq 0$;
z := w/y;
Assume $\neg(z \geq 0)$;
Confirm $-z = |w/y|$;

We now need a rule for **Assume** statements:

code; **Confirm** $P \Rightarrow Q$;

---

code; **Assume** P; **Confirm** Q;

Applying the rule for **Assume**, we obtain:

(1) **Assume** $y \neq 0$**;**
z := w/y;
Confirm $z \geq 0 \Rightarrow z = |w/y|$;

(2) **Assume** $y \neq 0$;
z := w/y;
Confirm $z < 0 \Rightarrow -z = |w/y|$;

Now we apply the assignment rule to each branch:

(1) **Assume** $y \neq 0$;
Confirm $w/y \geq 0 \Rightarrow w/y = |w/y|$;

(2) **Assume** $y \neq 0$**;**
Confirm $w/y < 0 \Rightarrow -w/y = |w/y|$**;**

Another application of the **Assume** rule yields:

(1) **Confirm** $y \neq 0 \Rightarrow \left(w/y \geq 0 \Rightarrow w/y = |w/y|\right)$;

(2) **Confirm** $y \neq 0 \Rightarrow \left(w/y < 0 \Rightarrow w/y = -|w/y|\right)$**;**

To complete the proof we need a rule for **Confirm**:

Q

Confirm Q;

Applying the **Confirm** rule produces the following mathematical propositions:

(1) $y \neq 0 \Rightarrow \left(w/y \geq 0 \Rightarrow w/y = |w/y|\right)$

(2) $y \neq 0 \Rightarrow \left(w/y < 0 \Rightarrow -w/y = |w/y|\right)$

As the example illustrates, every reverse rule application produces one or more new hypotheses, each of which has fewer programming constructs than the conclusion line. Ultimately, all the programming language syntax disappears, leaving hypothesis written strictly in the language of the underlying mathematical theory. In this case that theory happens to be real number theory, which allows us to conclude that both of these assertions are true from the definition of absolute value.

The part of the proof involving only the mathematical theory may strike the reader as particularly easy, and one may fear that only such obviously contrived examples as this will be so simple to verify. However, throughout this thesis, example after example will show that this phenomenon is not peculiar to this simple program, but rather is a common occurrence. In fact, we have not yet found any example in which the proof of program correctness requiresd more than simple use of mathematical definitions and straightforward applications of the appropriate theory. This is really no surprise if one stops to consider that, in order to write the correct code, the programmer must know at an intuitive level whatever theorems underlie the reasoning he is using for program development.

In the verification literature, associated with every loop is a clause called the "loop invariant." As the name suggests, the loop invariant is an assertion, which is true both before and after each iteration of the loop. The syntactic marker for the loop invariant is the keyword **Maintaining**. A simple example illustrating the **while** loop construct is:

Assume $n \geq 0$;
sum := 0;
i := 0;
Maintaining $i \leq n \wedge sum = \sum_{j=1}^{i} j$
while i < n **do**
    i := i + 1;
    sum := sum +1;
end;
Confirm $sum = \sum_{j=1}^{n} j$

Here the invariant states that *i* never exceeds *n* and that at the beginning or end of any iteration, the variable *sum* contains the total of the first *i* integers. This exactly describes what the loop is doing, namely computing a sequence of partial sums until it finally has the sum of the first *n* integers. For convenience in what follows, we will name this particular loop invariant "Sum_Inv." That is,

Sum_Inv = "$i \leq n \wedge sum = \sum_{j=1}^{i} j$."

In order to verify this program, we will need the proof rule for **while** statements:

code; **Confirm** Inv;
Assume Inv $\wedge$ B; body; **Confirm** Inv;
Assume Inv $\wedge \neg$ B; **Confirm** Q;
---
code; **Maintaining** Inv **while B do** body **end; Confirm** Q:

Here the first hypothesis is that the invariant *Inv* be true before the loop is executed. The second hypothesis requires that if *Inv* is true and the conditional *B* for the

loop is true and the body is executed, then *Inv* is true after execution, i.e., that *Inv* truly is an invariant. The third says that the truth of *Q* should follow from the truth of *Inv* and $\neg$ *B*, since they will both hold true when the loop terminates.

We will now use this rule in establishing the correctness of the summation program. As before, we will take for granted that the variables have been declared. Since the **while** loop is the last construct of this program, we apply that rule first,

obtaining three hypotheses:

(1) **Assume** $n \geq 0$; sum := 0; i := 0; **Confirm** Sum_Inv;

(2) **Assume** Sum_Inv $\wedge$ $i < n$; i := i + 1;
sum := sum + i; **Confirm** Sum_Inv;

(3) **Assume** Sum_Inv $\wedge$ $i \geq n$;
Confirm sum = $\sum_{j=1}^{n} j$;

To prove correctness of the program, we must apply the appropriate proof rules to each of these hypotheses. For the first, applying the assignment rule to *i* := 0 yields:

(1) **Assume** $n \geq 0$; sum := 0;
Confirm $0 \leq n \wedge \text{sum} = \sum_{j=1}^{0} j$;

Applying the assignment rule to *sum* := 0 leads to:

Assume $n \geq 0$;
Confirm $0 \leq n \wedge 0 = \sum_{j=1}^{0} j$;

Finally, using the rule for **Assume**, followed by the **Confirm** rule, we obtain:

$n \geq 0 \Rightarrow 0 \leq n \wedge 0 = \sum_{j=1}^{n} j$;

This follows from the definition of $\sum$.

In hypothesis (2), the body of the loop consists of two assignments, so we apply the assignment rule twice to obtain:

(2) **Assume** $i < n \wedge$ Sum_Inv;
Confirm $i + 1 \leq n \wedge \text{sum} + i + 1 = \sum_{j=1}^{i+1} j$;

Applying the **Assume** and **Confirm** rules, we get

$$i < n \wedge i \leq n \wedge sum = \sum_{j=1}^{i} j \Rightarrow$$

$$i + 1 \leq n \wedge sum + i + 1 = \sum_{j=1}^{i+1} j$$

This can be seen to be correct by adding $i + 1$ to both sides of the equation for *sum* in the hypothesis.

Hypothesis (3) expands out to:

(3) **Assume** $i \leq n \wedge sum = \sum_{j=1}^{i} j \wedge i \geq n$;

Confirm $sum = \sum_{j=1}^{n} j$;

Applying the rules for **Assume** and **Confirm**, we obtain:

$$i \leq n \wedge sum = \sum_{j=1}^{i} j \wedge i \geq n \Rightarrow$$

$$sum = \sum_{j=1}^{n} j$$

From $i \leq n$ and $i \geq n$, it follows that i = n, and we have the desired result.

To write the loop invariant, the programmer needed to know that the loop will have the total of the first *i* integers in *sum* after *i* iterations and that the highest value *i* will achieve is *n.* But, of course, had the programmer not known both of those facts implicity, he would not have been able to write this program. The point here is that loop invariants are not mysterious, nor do they requires deep mathematical insights, which most programmers are unlikely to have. Loop invariants are simply descriptions of what the loop does.

Just as a compiler must keep information about procedures so that it can do type checking of parameters and can find appropriate code into which to transfer control, the verifier must know certain information about procedures in order to be able to generate correctness proofs.

When a procedure call is made in a program, the verifier does not need to see the code for that procedure at all, but rather it must know what the code does, i.e. what its specifications are. So whenever a procedure is declared, the heading, which contains its specifications, is recorded in what we will call the program's "context."

To illustrate how procedures look, what the heading is, and where the specifications appear, we will present an example of a program fragment in which the number of permutations of *n* objects taken *r* at a time is computed. To facilitate the computation, a procedure for calculating factorials is used. The declaration of the factorial procedure is shown first:

```
Proc Find_Factorial (const n: integer, var fact: integer)
      requires n ≥ 0;
      ensures fact = n!;
      fact := 1;
      i := 0;
      Maintaining fact = i! ∧ i ≤ n
      while i < n do
        i := i + 1;
        fact := i * fact
      end
end;
```

The **requires** clause is a programmer supplied assertion which tells what restrictions must be met in order that, if the code is correct, the **ensures** clause will hold upon completion of the procedure body. Together, the **requires** and **ensures** clauses form the specifications of the procedure. The heading of the procedure consists of the first line and the specifications:

```
Proc Find_Factorial (const n: integer, var fact: integer);

      requires n ≥ 0;

      ensures fact = n!;
```

For any given block of code, the verifier will first look at the declarations and put appropriate information into the context, i.e., save information which will be needed in order to apply the proof rules. Then the verifier begins at the penultimate statement and proceeds toward the beginning of the program (usually an **Assume** statement), applying the appropriate rule as we have seen in the proceeding examples.

So, in this example, the verifier will have to put the heading of Find_Factorial into the correct context before applying the other rules necessary for proving correctness. Hence, if and when Find_Fact is called, the specifications telling what Find_Factorial does will be available. It is clear that we need two new proof rules, one to handle procedure declarations and one for procedure calls. The following is a simplified version of the procedure declaration rule:

C ∪ {p_heading} \ **Assume** pre; **Remember** x; body; **Confirm** post;
C ∪ {p_heading} \ code; **Confirm** Q;

C \ **Proc** p(**var** x: T); **requires** pre; **ensures** post; body; **end** code; **Confirm** Q;

Both hypotheses of the declaration rule indicate that the heading of the procedure being declared (but not its body) is to be placed in the context. This makes all the necessary information about the procedure available so that it can be called from any part of the program, including from within itself. The first hypothesis of the rule establishes that the procedure works as specified by requiring that, if the **requires** clause is met, then the **ensures** clause is true upon completion of the body of the procedure. The **requires** clause *pre* is called the precondition of the procedure and the **ensures** clause *post* is called the postcondition.

In applying the procedure declaration rule to Find_Factorial, we note that the first hypothesis to consider will be:

C $\cup$ {Find_Fact_Heading} \ **Assume** n $\geq$ 0; Find_Fact_Body;
Confirm fact = n!;

Here *C* stands for whatever context already exists at the point of declaration of Find_Factorial, and as the rule shows, the Find_Fact_Heading is added to that context by the verifier. The \ mark separates the context from the assertive program to be proved correct. Find_Fact_Body is an abbreviation to stand for the code in the body of the Find_Factorial procedure. We note that this hypothesis is in an already familiar form because it looks just like the preceding examples. Indeed, in order to establish the stated hypothesis, we proceed exactly as we did in the examples already given, and we will find that the rules we need are ones we have seen before.

Since the body of Find_Factorial ends with a **while** loop, it is the **while** rule that we apply first, thereby generating three hypotheses to check:

(1) **Assume** n $\geq$ 0; fact := 1; i := 0;
Confirm fact = i! $\wedge$ i $\leq$ n;
(2) **Assume** fact = i! $\wedge$ i $\leq$ n $\wedge$ i < n ;
i := i + 1; fact := i * fact;
Confirm fact = i! $\wedge$ i $\leq$ n ;
(3) **Assume** fact = i! $\wedge$ i $\leq$ n $\wedge$ i $\geq$ n ;
Confirm fact = n!;

For (1) we apply the assignment rule twice, obtaining:

(1) **Assume** n $\geq$ 0; **Confirm** 1 = 0! $\wedge$ 0 $\leq$ n ;

Applying the rules for **Assume** and **Confirm**, we get:

$$n \geq 0 \Rightarrow 1 = 0! \wedge 0 \leq n ;$$

The fact that 1 = 0! is a definition from number theory.

Hypothesis (2) also requires two applications of the assignment rule and use of the rules for **Assume** and **Confirm**. The result is:

fact = i! $\wedge$ i $\leq$ n $\wedge$ i < n $\Rightarrow$
(i + 1) * fact = (i + 1)! $\wedge$ i + 1 $\leq$ n

The first part of the conclusion can be proven by multiplying both sides of the equation fact = i! by i + 1. The other part of the conclusion is obvious since i < n.

For hypothesis (3) we need to verify:

(3) **Assume** fact = i! $\wedge$ i $\leq$ n $\wedge$ i $\geq$ n ;
Confirm fact = n!;

Applying the **assume** and **confirm** rules yields:

fact = i! $\wedge$ i $\leq$ n $\wedge$ i $\geq$ n $\Rightarrow$
fact = n!;

Since i $\leq$ n and i $\geq$ n, i = n. Hence fact = i! implies that fact = n!.

We have seen that an application of the proof rule for procedure declarations causes the context to be enriched with the procedure heading. Such an application also establishes that if the parameters to the procedure meet the **requires** clause, then upon completion of the procedure body, the **ensures** clause is met.

In case the post condition refers to old values of one of the parameters, we need the **Remember** rule:

$\mathcal{C}$ \ code; **Confirm** RP[@s⇝s, @t⇝t];

$\mathcal{C}$ \ code; **Remember** s, t; **Confirm** RP⌿ s, @s, t, @t, u, v, ··· ⌿;

Next we will see how procedure calls work by examining a call to Find_Factorial. The following program fragment computes the number of permutations of *n* objects taken *r* at a time:

```
C \ Assume n ≥ r ≥ 0;
  Find_Factorial(n, nfact);
  d := n – r;
```

Find_Factorial(d, dfact);
Perm := nfact/dfact;
Confirm Perm = ${}_{n}P_{r}$;

To show correctness of this program fragment, we will need to apply the assignment rule first, followed by two applications of the call rule with an assignment between the two calls.

The following is a simple version of the proof rule for procedure calls:

C \ code; **Confirm** pre[x $\rightarrow$ a]
C \ code; **Confirm** $\forall$?a: T, post[@x $\rightarrow$ a, x $\rightarrow$?a] $\Rightarrow$ Q[a $\rightarrow$?a]

C \ code; p(a); **Confirm** Q

Since the declaration rule establishes that if the **requires** clause is satisfied, then the **ensures** clause is met upon completion of the procedure body, the first hypothesis of the **call** rule checks that the **requires**clause *pre* holds when the actual parameter *a* is substituted for the formal parameter *x*. The second hypothesis asks that the **ensures** clause *post* imply *Q*. The @ sign in front of the variable *x* refers to the value of *x* at the beginning of the procedure. This distinguishes the old value of *x* from the current value. The extra complication here is the introduction of a new variable ?*a* to stand for the value of the actual parameter *a* after the procedure *p* has modified it.

The **call** rule is simpler to understand in the version presented above, but in normal usage we don't want to break out two separate hypotheses when a single, slightly lengthier one will do. So we ordinarily combine this rule in a single hypothesis rule:

C \ code; **Confirm** pre[x $\rightarrow$ a] $\wedge$
    $\forall$?a: T, post[@x $\rightarrow$ a, x $\rightarrow$?a] $\Rightarrow$ Q[a $\rightarrow$?a]

C\ code; p(a); **Confirm** Q

With this rule available to us, we are ready to establish correctness of our program fragment. Since the last executable statement in our fragment is an assignment, we apply the assignment rule first, obtaining:

C \ **Assume** $n \geq r \geq 0$;
  Find_Factorial(n, nfact);
  d := n – r;
  Find_Factorial(d, dfact);
  Confirm nfact/dfact = $_{n}P_{r}$;

Next we need the call rule;

C \ **Assume** $n \geq r \geq 0$;
  Find_Factorial(n, nfact);
  d := n – r;
  Confirm $d \geq 0 \wedge \forall ?dfact : integer, ?dfact! \Rightarrow$
  nfact/?dfact = $_{n}P_{r}$;

Applying the assignment rule results in:

C \ **Assume** $n \geq r \geq 0$;
  Find_Factorial(n, nfact);
  Confirm $n - r \geq 0 \wedge \forall ?dfact : integer, ?dfact = (n - r)! \Rightarrow$
  nfact/?dfact = $_{n}P_{r}$;

Another application of the call rule yields:

C \ **Assume** $n \geq r \geq 0$;
  Confirm $n \geq 0 \wedge \forall ?nfact, ?nfact = n! \Rightarrow$
  $n - r \geq 0 \wedge \forall ?dfact : integer, ?dfact = (n - r)! \Rightarrow$
  ?nfact/?dfact = $_{n}P_{r}$;

Upon applying the rules for **Assume** and **Confirm**, we get an implication:

$n \geq r \geq 0 \Rightarrow n \geq 0 \wedge \forall ?nfact, ?nfact = n! \Rightarrow$
$(n - r \geq 0 \wedge \forall ?dfact : integer, ?dfact = (n - r)! \Rightarrow$
?nfact/?dfact = $_{n}P_{r}$;

Since $n \geq r \geq 0, n \geq 0$. Since ?nfact = n! and ?dfact = (n – r)!, by a definition in combinatorics, ?nfact/?dfact = ${}_nP_r$, since ${}_nP_r$ = n!/(n – r)! Hence our program fragment is correct.

The two examples we have just presented performed calculations on integers in a programming style, which is typical for manipulating small objects, i.e., a constant parameter was passed to a procedure which performed some calculations and then assigned the resulting value to a variable parameter. This is typical and reasonable because small objects take up little space, and psychologically it seems reasonable to have a new variable to represent the result of the operation while the original variable retains its value. In the case of *Find_Factorial,* the constant parameter *n* remained fixed while the variable parameter *fact* had a new value after the computation was completed. Our simplified procedure declaration rule was sufficient for proving the correctness of procedures written in this style.

However, this programming style is not desirable when programming with large objects, or even small ones in some cases. For example, in sorting the elements of an array, a programmer would not want to create a new array each time a permutation of the given elements is made. In order to consume both time and space, it is preferable to modify the existing array. This means that there must be a way to specify what modification has been made, and this requires us to reference values of variables both before and after the change takes place. To accommodate this need we have introduced the use of the symbol "@" to be placed in front of a variable *var_name* to indicate that its value at a given point in the program must be remembered as *@var_name*.

To illustrate the use of this idea, consider a trivial procedure which increments a given integer by a constant value. Here, even though it may not require a lot of space to create a new integer to hold the incremented value, psychologically we expect the given variable to have its old value destroyed and replaced by its incremented value. This procedure might be part of an integer package:

```
Procedure Increment(var n: integer, const c: integer);
  requires min_int ≤ n + c ≤ max_int;
  ensures n = @n + c;
  n := n + c;
end;
```

The **requires** clause makes sure that the result is within the bounds permitted for integers. The **ensures** clause states that the new value of *n* will be the old value of *n* incremented by *c*.

Verification of procedures whose clauses refer to old values requires us to supplement our declaration rule as follows:

```
  C ∪ {p_heading} \ Remember; Assume pre; body; Confirm post;
            code; Confirm Q;
-------------------------------------------------------------------
  C \ Proc p(var x, const y); requires pre; ensures post; code; Confirm Q;
```

The new keyword, which appears here indicates that values of all variables are to be saved as of this point in the program. The proof rule for **Remember** is:

```
  C \ code; Confirm Shift(Q);
  ------------------------------------------------
  C \ code; Remember; Confirm Q;
```

where Shift(Q) is defined by:

Shift(x) = x
Shift(c) = c for any constant c
Shift(@x) = x
Shift(f(e1,e2)) = f(Shift(e1), Shift(e2))

The Shift function removes one-pound sign from any given variable if one is there.

At this point, we can now make some general observations about how our proof system works. Although the programmer must supply certain clauses specifying program behavior, the generation of intermediate assertions is mechanical. An automated verifier can determine syntactically what rule to apply at each step of the proof. Moreover, when all the pertinent rules have bee applied, the statement which remain are ones involving only the mathematical theory for the given program, and the proof reasoning can then be completed using only traditional mathematical reasoning.

An automatic verifier will perform three major tasks: (1) generation of assertions by applying the rules, (2) simplification of these assertions, probably as the verifier proceeds through the code, rather than all at once, and (3) application of axioms and theorems of the underlying mathematical theories in an attempt to prove the final assertion.