CSC 309 Lecture Notes Week 10
Introduction to Formal Program Verification

I. Two introductory definitions.

A. *Testing:* show that a program is correct for some (finite) set of inputs.

B. *Verification:* prove that a program is correct for all possible inputs.

II. Review of the problem with testing.

A. For a large system, testing cannot cover all possible cases.

B. Hence we can never be 100% sure that a system is correct.

C. For some systems, such as safety critical applications, this is simply not good enough

D. Enter program verification, the goal of which is to demonstrate in some convincing way that a system works *in general*, for all possible inputs.

III. Some practical applications of formal specification and verification.

A. Proof-carrying code.

1. There are potential problems with code sent between machines on a network, for example bots or agents that crawl around the web looking for things.
    1. Such code may want to run on a foreign host machine.
    2. The host would like to know if the code actually works properly.
2. Some proof-carrying code terminology:
    a. A *code producer* is the party who has code that wants to run on some foreign host machine.
    b. A *code consumer* is the host, seeking to verify the code produce gets up to no mischief.
    c. In particular, the code producer may violate *policies* of code consumer (see Figure 1).
3. To solve the problem:
    a. The code producer compiles *and proves* code.
    b. The proof is based on the formal policies defined by consumer, e.g., safety and data integrity policies.
    c. The producer sends code to consumer.
    d. The consumer checks that proof still holds; the checking is performed dynamically (see Figure 2).

B. Model checking.

1. It's certainly well known that large software exhibits complex behavior.
2. The idea of model checking is to prove properties of a model before it's implemented.
3. Noteworthy recent uses of model checking have been in avionics software, where correct operation is critically important.
4. One of the best examples is John Rushby's proof of a redundancy model, that behaves reliably in the face of so-called "Byzantine" failures.

C. Formalizing user mental models.

1. With model checking, complex software can get more reliable.
2. Problems still arise in human user errors.
3. E.g., modern aircraft systems are increasingly reliable.
    a. 70% of problems are human error.
    b. Cockpits are highly automated.
    c. Pilots can be surprised by system behavior.
4. Formal methods used for this problem:

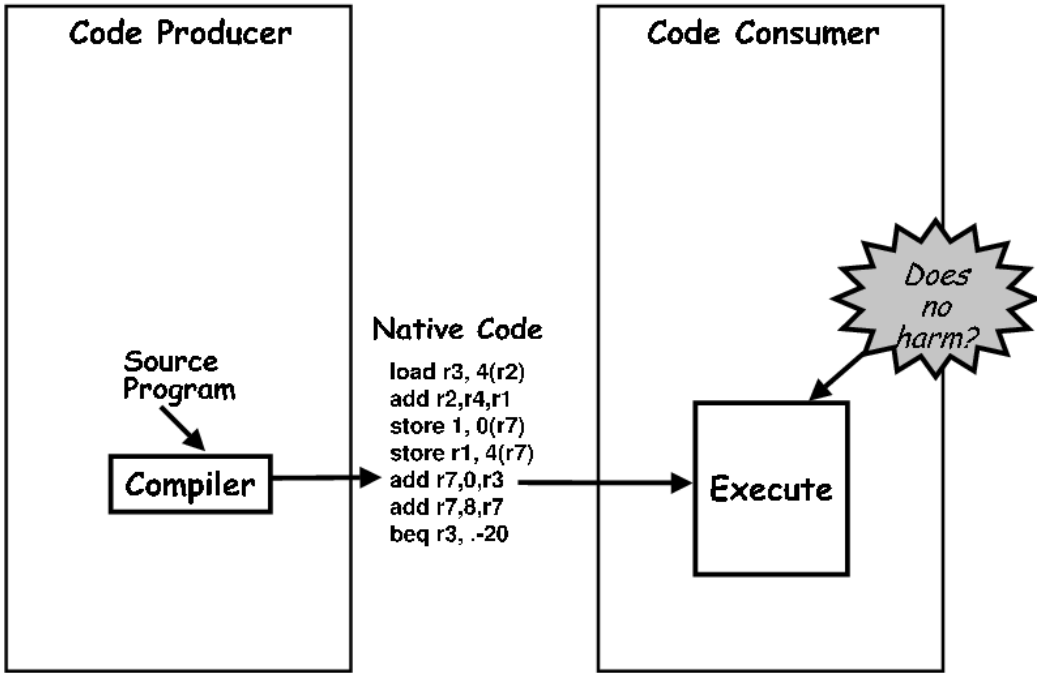


Figure 1: Code producers and consumers.

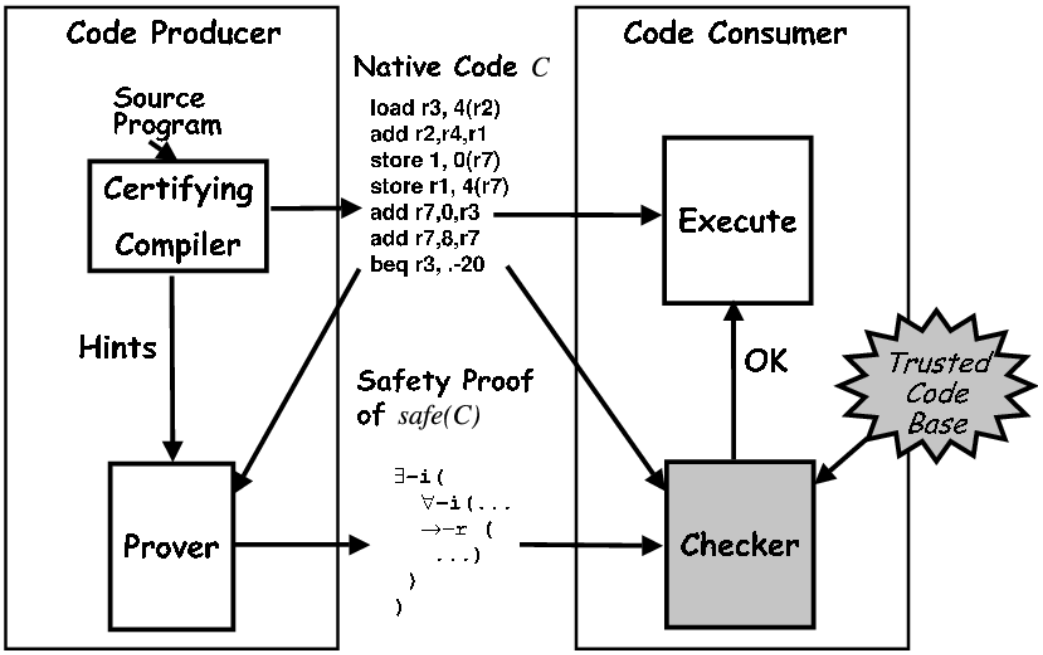


Figure 2: Proof-carrying code.

a. Cockpit control system formalized.
b. Pilot mental model formalized.
c. Model checking verifies consistency.
d. Inconsistencies help explain human failures and point to ways to improve the system.

5. This approach was successfully used to diagnose a real-life pilot error.
   a. It helped explain a (non-fatal) mishap that had otherwise gone undiagnosed.
   b. It pointed to two important improvements in the cockpit control model.

IV. A very simple example function.

```
/*
 * Compute factorial of x, for positive x, using an iterative technique.
 *
 * Precond: x >= 0
 *
 * Postcond: return == x!
 *
 */
int Factorial(int x) {
    int y;
    y = 1;
    while (x > 0) {
        x = x - 1;
        y = y * x;
    }
    return y;
}
```

Question: Is this correct?

V. Symbolic evaluation.

A. In the unit testing schemes that we've looked at in 309, all test inputs and outputs have been *concrete* values, as opposed to *symbolic* values.

B. To see what we mean by this, let's consider how we would test the factorial function with concrete values.

C. Table 1 shows a typical unit test plan for function `Factorial`.

D. To test, we feed these concrete values in and check out the results.

E. Two important questions even for this very simple function:

Test No.	Input	Expected Results
1	x = -1	ERROR (precondition violation)
2	x = 0	return = 1
3	x = 1	return = 1
4	x = 4	return = 24
5	x = 6	return = 120
6	x = 70	return > 10**100 (possible numeric overflow)

Table 1: Unit testing Factorial.

1. How do we know what the expected results are supposed to be? (Ans: an oracle.)
2. How do we know that it works *for all* inputs? (Ans: we'll have to prove it.)

F. One way to help answer these questions is to consider supplying a single *symbolic* value for the input, and analyzing a resulting symbolic output.

G. Here are the details (assuming that we've corrected bug by changing the order of the two loop statements):

1. For every expression that involves some input variable, we compute a symbolic result rather than a concrete result.
2. E.g., the symbolic computation of the statements:

```
y := 1;
y := y * x;
```

results in y having the *symbolic value* of *1 * x* which simplifies to just x.

3. Suppose we do some more symbolic computation:

```
x := x - 1;
y := y * x;
```

4. This results in a symbolic value of *x * (x - 1)* for y.
5. A bit more symbolic computation:

```
x := x - 1;
y := y * x;
```

results in y's symbolic value of *x * (x - 1) * ((x - 1) - 1)* which simplifies to *x * (x - 1) * (x - 2)*

6. The idea is that we treat input values as *symbols* inside expressions rather than as concrete values

H. What a symbolic evaluator would do for us in the factorial example is produce symbolic output results such as the following (on the *correct* version of the function):

y = 1
↓
y = 1 * x
↓
y = x * (x-1)
↓
y = x * (x-1) * (x-1-1)
↓
y = x * (x-1) * (x-2) * ((x-2)-1)
↓
.
. *after N times through the Factorial loop symbolically*
.
↓
y = x * (x-1) * ... * (x-N)

I. What's nice about this is that we don't need to worry about any concrete values at all, but we see an informative symbolic pattern developing that tells us by inspection if the function appears to be working.

J. It's also interesting to look at the erroneous case where the loop statements have been transposed (i.e., the way the function is defined originally above):

y = 1
↓
y = 1 * (x-1)
↓
y = (x-1) * ((x-1)-1)
↓
y = (x-1) * (x-2) * (x-3)

↓
.
.
.
↓
y = (x-1) * (x-2) * ... * (x-1-N)

K. Here the error manifests itself in a symbolic expression that's clearly wrong for computing factorial (i.e., it's off by an increment of 1).

L. We could do this kind of evaluation by hand, but it's much nicer if we have an augmented form of interpreter to do it for us. (Such symbolic interpreters in fact exist, and they aren't even that tough to build.) operations.)

VI. Moving on to formal verification

A. While symbolic evaluation is more general than concrete testing, it still involves some informal human analysis to verify that the output of a program is correct.

B. What we want is a statement of mathematical certitude that a program is correct *for all* inputs -- i.e., we want to *mathematically prove* program correctness.

C. That is, we seek to prove that a program *meets* its specification fully.

D. The general steps to perform formal program verification are the following:

1. We will need to make some formal connection between a program written in a programming language and the language of formal mathematics.
   a. We do this by defining *mathematical meaning* for a particular programming language.
   b. Specifically, the meaning is given by a set of rules for how each statement in the language behaves mathematically.
   c. For example, we can define an assignment statement in terms of mathematical variable substitution; a programming language if-then-else is defined pretty directly as mathematical if-then-else.
   d. In these notes, we will give a few such rules, and see in general what it takes to define programming language formally.
   e. We call such a set of rules a *mathematical* (or sometimes *axiomatic*) semantics for a programming language.
2. After we define the mathematical language rules, we need a general procedure that tells us how to assign meaning to a particular program.
   a. This procedure is rather like symbolic evaluation, in that we walk through a program producing symbolic rather than concrete results.
   b. Coming up, we will look at specific verification procedure known as ''backwards substitution''

E. Given the language rules and verification procedure, we need to state formal pre and post conditions for any function that we want to verify formally.
   a. These are precisely the kind of pre/post conds that we did for the specs.
   b. As it turns out, we'll need just a bit more than function pre/post conds -- we also need conditions for each *loop* within a function.
   c. These loop conditions are called *loop invariants*, and they can be generated with the help of a symbolic evaluator.

F. After the preconditions, postconditions, and loop invariants are established, we apply our rule-based verification procedure to the desired function, the result being the mathematical verification of the proposition:

precond $\supset$ postcond, *through the function*

i.e., the precond mathematically implies the post cond when the mathematical rules of the function are obeyed.

G. To make it clear that the rules of the function body must be obeyed in order for the implication to be true,

program verifiers have invented a new notation

precond {function body} postcond

which means that if the precond is true before the function body is (mathematically) executed, then the postcond must be true after the execution is complete.

H. This notational form is called a "Hoare triple", named after C.A.R. Hoare, one of the acknowledged founders of formal verification, among other computer science principles.

I. A final step in many formal verifications is to prove the *termination condition*, that says under which conditions the function will actual finish its computation; more on this later.

J. We will look at a set of verification rules for a Pascal-class language, and an actual verification of the Factorial example.

VII. Simple Flowchart Programs (SFPs)

A. When we looked at path testing, it was convenient to think of programs in graphical flowchart form.

B. Flowcharts are also a helpful representation for understanding formal verification.

1. There's nothing particularly special about the flowchart representation of programs.
2. It's just a graphical view of programs that's isomorphic to a textual version written in some textual programming language.
3. For our intro to formal verification, we'll use a notation that covers just three of the most basic program constructs.
4. With some additional work (that we don't do here), these simple basic constructs can be generalized to cover all of the constructs of a high-level programming language.

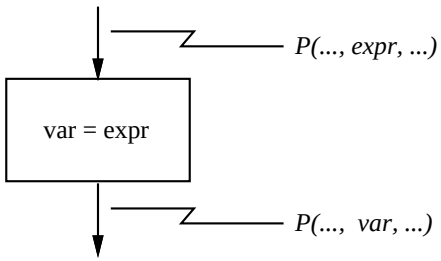
C. The basic constructs are:

1. an assignment statement,
2. an if-then-else statement
3. a top-of-loop node that is used in conjunction with an if-then-else to form while loops in a flow chart.
4. a function call

D. Graphical versions of these rules follow.

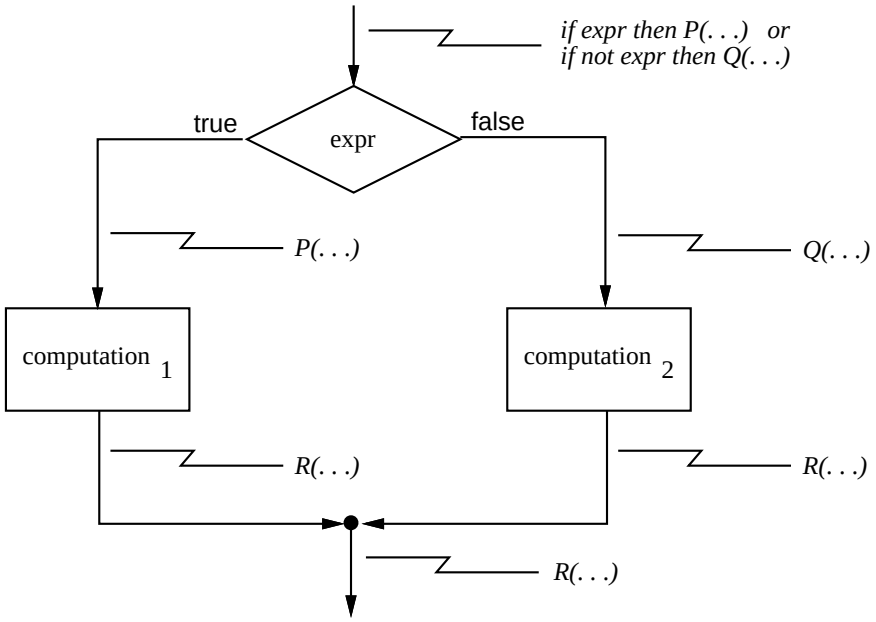
VIII. The semantic rules for SFPs

A. The rule of assignment



1. The picture describes the meaning of assignment in terms of variable substitution.
2. Specifically, the precondition for `var = expr` is derived from the postcondition by systematically substituting all occurrences of `var` in the postcondition with `expr` in the precondition.

B. The rule of if-then-else



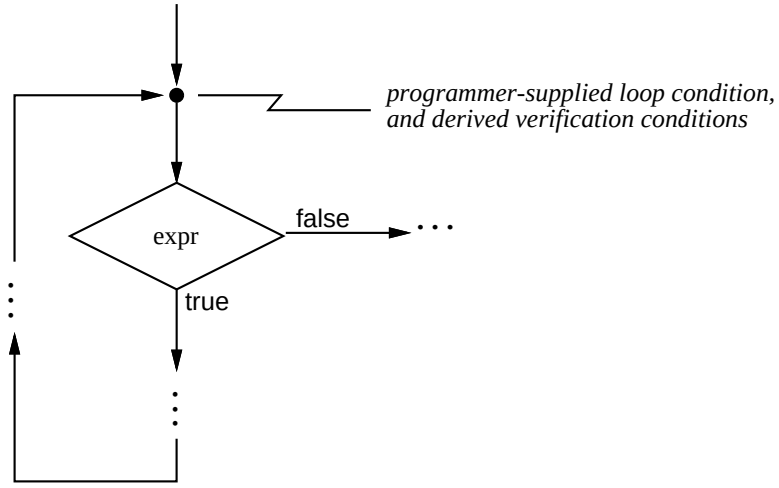
1. The picture shows an if-then-else program statement defined mathematically in terms of logical implication.
2. Here we use a "syntactically sugared" form of logical implication, i.e.,

   if X then Y

   is equivalent to

   $X \supset Y$
3. In the picture, the predicates *P(...)* and *Q(...)* are derived from predicate *R(...)* by applying proof rules for computation$_1$ and computation$_2$, respectively.

C. The rule for loops



1. The proof rule for a loop requires that the programmer supply a loop condition, also known as a loop

invariant, for each loop in the program.

2. These invariants are in addition to the pre- and postconditions for each function.
3. A loop invariant is a condition that is true throughout the execution of the loop body.
4. The invariant must be stated in terms of all the variables that are used and modified within the loop body, including the loop test.

IX. Application of semantic rules

A. Recall from above that the overall goal of a program verification is to prove

precond {function body} postcond

B. That is, the precond implies the postcond *through* the function body.

C. The semantic rules of the programming language provide the means to mechanically *push predicates through* a program in order to prove the desired implication goal.

X. The backwards substitution technique

A. To actually carry out the verification process, we do a kind of symbolic evaluation on an SFP.

B. But, we are evaluating with predicates rather than program variable values.

C. In theory, we can evaluate in either a forward or backward direction.

1. In practice, however, backwards evaluation is easier.
2. This is due to the way the rules are constructed and the fact that we are trying to prove an implication from precond to postcond.

D. Here are the steps to carry out a verification, given a SFP program and the semantic rules for the SFP language:

1. Annotate the program with pre and post conditions.
2. At each loop node, provide an additional predicate called the *loop invariant* (more on this below).
3. Take the overall program postcondition and *push it through* the program using the semantic rules.
4. At every point that a "pushed-through" predicate "runs into" a supplied predicate, we have a *verification condition (VC)* that must be proved.
5. After all VCs are proved, the program proof is complete, except for a termination condition may need to be proved.
6. We do not deal with proof of termination in these notes.

XI. Before we tackle verification of the Factorial example, let's see how the preceding verification rules can be used to prove that 2+2=4 (a clearly stunning result).

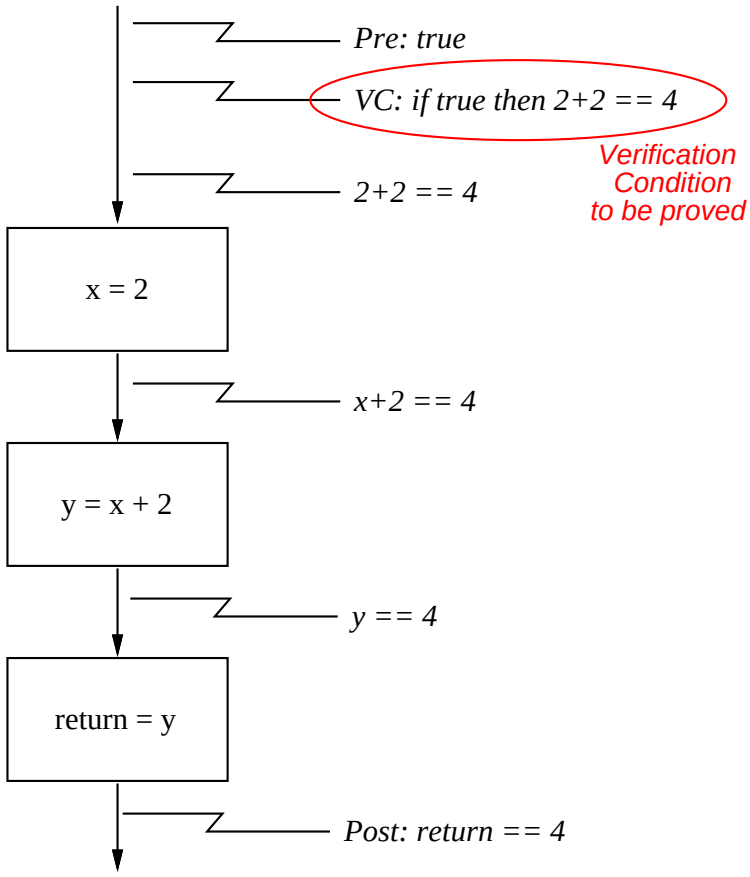
A. Here's the program:

```
int Duh() {
    /*
     * Add 2 to 2 and return the result.
     *
     * precondition: ;
     * postcondition: return == 4;
     *
     */

    int x,y;
    x = 2;
```

```
    y = x + 2;
    return y;
}
```

B. Here's the SFP:



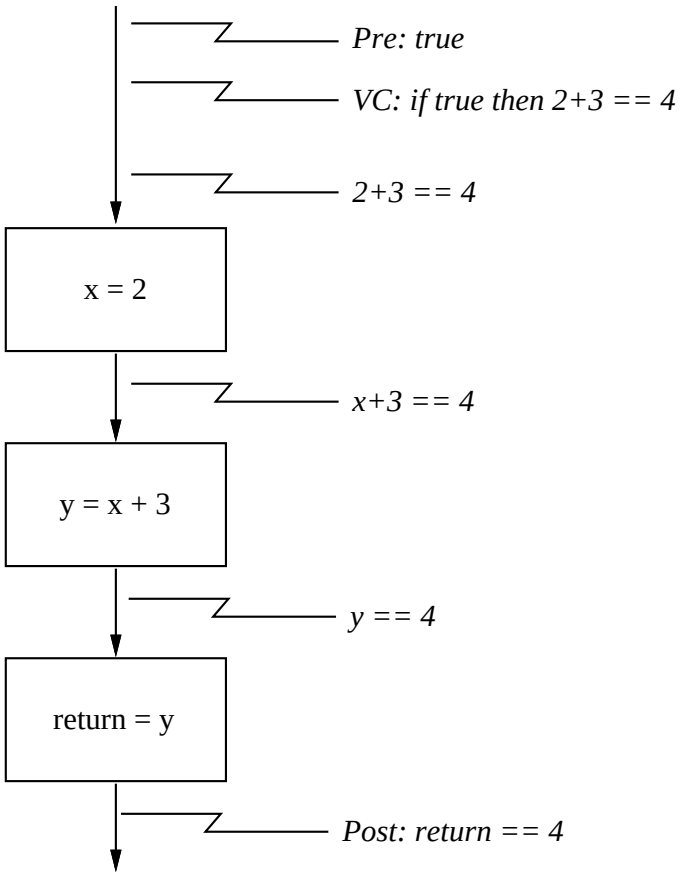
XII. The example above concluded with the startling result that a program correctly adds 2+2 to get 4.

A. Let's try to prove the following implementation:

```
int ReallyDuh() {
/*
 * Add 2 to 3 and return the result.
 * precondition: ;
 * postcondition: return == 4;
 */

    int x,y;
    x = 2;
    y = x + 3;
    return = y;
}
```

B. Here's the proof attempt



C. What happens here is that we are left with the VC

true ⊃ 4 == 2 + 3 ==>
true ⊃ false

which is false.

D. In general, proofs will go wrong at the VC nearest the statement in which the error occurs.

XIII. The basic ground rules of implication proofs

A. You may recall from your discrete math class the following truth table for logical implication:

p	**q**	**p ⊃ q**
0	0	1
0	1	1
1	0	0
1	1	1

B. That is, the logical implication $p \supset q$ is only false if p is true and q is false.

C. Now, in a formal program verification, we assume that the p in the implication formula is true, since it represents the precondition.

D. Hence, the basic way that a VC will fail to be proved is if q in the implementation is false (as was the case in the attempt to prove 2 + 3 == 4).

XIV. Now let's try a proof of the Factorial example.

A. Here's the (correct) function definition:

```
int Factorial(int N) {
/*
 * Compute factorial of x, for positive x, using an iterative technique.
 *
 * Precond: N >= 0
 *
 * Postcond: return == N!
 *
 */
    int x,y;     /* Temporary computation vars */

    x = N;
    y = 1;
    while (x > 0) {
        y = y * x;
        x = x - 1;
    }
    return y;
}
```

B. Note that this is slightly different than the version presented at the beginning of the notes.

1. Here the formal parameter has been changed from `x` to `N`, and `x` is now a local variable.
2. We'll discuss why this change was done a little later.

C. Figure 3 outlines the proof; VC proof details follow shortly.

XV. Logical derivation of loop invariant "y * x! = N!"

A. At the top of loops, we ask ourselves what relationship should exist between program variables throughout the loop. I.e., what relationship should x, y, and N have to one another each time through at the top of the loop?

B. Looking at it another way, we want to characterize the *meaning* of the loop in terms of program variables.

C. Since the meaning of whole program is y = N!, the meaning of the loop is something like "y *approximates* N!" But how?

D. Putting things a bit more precisely,

$$y \; \mathbf{R} \; f(x) = N!$$

for some relation R. And it looks like **R** is multiplication, i.e.,

$$y * f(X) = N!$$

E. So what is f(x)? I.e., how much shy of N! is y at some arbitrary point k through the loop? It looks like y is growing by a multiplicative factor of x each time through, so at point k we have

$$y = x * (x-1) * (x-2) * ... * (x-k) * (x-k-1) * ... * 1 = N!$$

F. I.e.,

$$y * x! = N!$$

G. This kind of reasoning is typical of that used to derive loop assertions.

H. An alternative to puzzling it out with abstract reasoning is to use symbolic evaluation as an aid in deriving loop assertions, which topic will look at shortly.

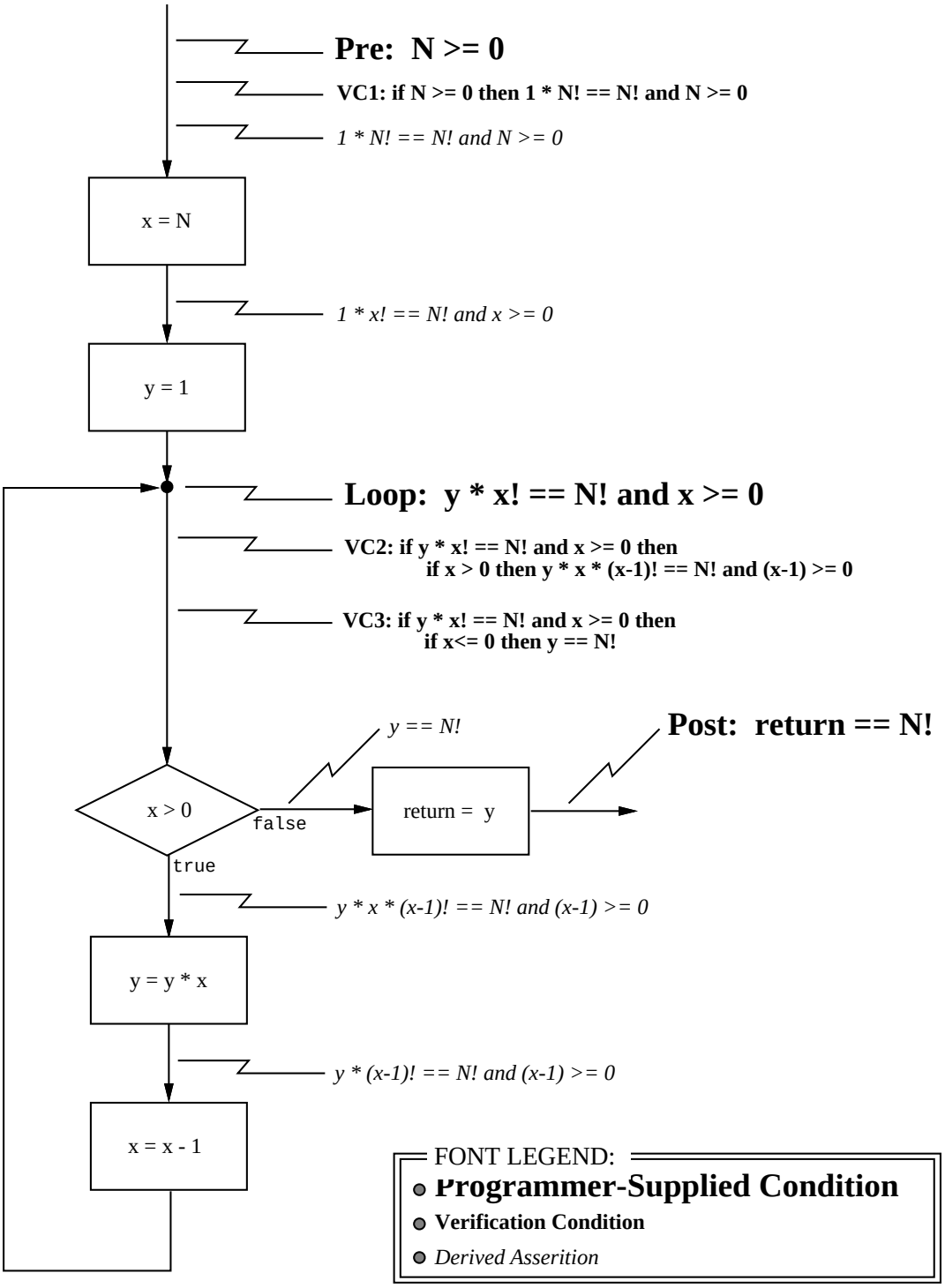


Figure 3: Factorial proof outline.

XVI. Further tips on doing the proofs

A. Generally, the proofs of verification conditions are not that difficult.

B. If the program is correct, then the proofs generally involve simple algebraic formula reduction.

C. A discrete Math book (e.g., from CSC 141) contains rules for logical formula manipulation.

D. In addition, here are some rules for reducing "if-then-else" style formulas:

1. if t then P1 else P2 <=> t ⊃ P1 and not t ⊃ P2
2. if t then t => true
3. if t then if t then P1 else P2 => if t then P1 else P2
4. if t then t and P => if t then P
5. if t1 then if t2 then P => if t1 and t2 then P
6. t and (if t then P) => P *(modus ponens)*
7. t and (if t then P1 else P2) => if not t then P2
8. x≥n and x≤n => x==n
9. x>n and x<n => false

XVII. A closer look at the factorial verification conditions (VC's)

A. According to the proof strategy outlined earlier, we are obligated to prove each verification condition.

B. For factorial, VC1 is trivial.

C. Proof of factorial VC2:

if (y*x! == N! and x>=0) then if (x>0) then y*x*(x-1)! == N! and (x-1)>=0 =>
if (y*x! == N! and x>=0) then if (x>0) y*x! == N! and x>=1 =>
if (y*x! == N! and x>=0) then if (x>0) y*x! == N! =>
if (y*x! == N! and x>=0) then y*x! == N! and x>0 =>
true

D. Proof of factorial VC3:

if (y*x! == N and x>=0) then if (x<=0) then y==N! =>
if (y*x! == N! and x==0) then y==N! =>
if (y*0! == N!) then y==N! =>
if (y*1 == N!) then y==N! =>
true

XVIII. Looking at some possible errors in factorial and how they would manifest in the verification.

A. Suppose we transpose the two loop body statements (''x = x-1'' and ''y = y*x''), as was the case in the original `Factorial` function presented above?

B. The ultimate result is we'll get the following erroneous VC3:

y * x! = N! and x≥0 and x>0 ⊃ y * (x-1) * (x-1)! = N! and x-1 ≥ 0 ==>

y * x! = N! and x>0 ⊃ y * (x-1) * (x-1)! = N!

no go

C. Suppose we have ''x ≥ 0'' in the test (instead of x strictly greater 0); we'll get the following:

y * x! = N! and x≥0 and ¬ (x≥0) ⊃ y = N! ==>

y * x! = N! and x≥0 and x<0 ⊃ y = N!

no go

XIX. Automatic generation of loop invariants via symbolic evaluation

A. A mechanical technique for generating loop assertions is to apply the idea of symbolic evaluation discussed above.

B. Starting with the output predicate, symbolically cranking through the factorial loop looks like this:

y = N!
↓
y * x = N!
↓
y * x * (x-1) = N!
↓
y * x * (x-1) * (x-2) = N!
↓
y * x * (x-1) * (x-2) * (x-3) = N!
↓
.
.
.
↓
y * x * (x-1) * ... * (x-N) = N!

C. By inspecting the result of this symbolic evaluation, we notice that the general relationship that remains true during loop execution is y * x! = N!.

D. This, then, is the *loop invariant* that remains true through the execution of the loop.

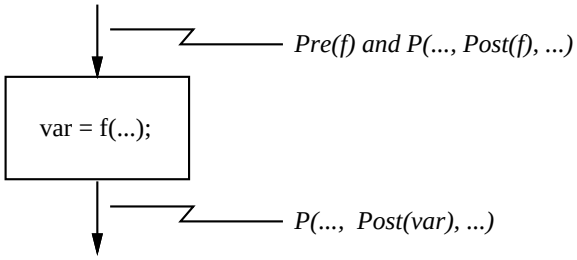
E. It's also interesting to look at the erroneous case where the loop statements have been transposed:

y = N!
↓
y * (x-1) = N!
↓
y * (x-1) * (x-2) = N!
↓
y * (x-1) * (x-2) * (x-3) = N!
↓
.
.
.
↓
y * (x-1) * (x-2) * ... * (x-N) = N!

F. In the erroneous case, the symbolic evaluation will lead us to derive the wrong loop assertion.

G. This will ultimately cause the verification to fail (if we don't notice that the assertion is clearly wrong before we attempt the verification).

XX. The verification rule for function calls:



where *Post(var)* is the postcondition of function f in which *var* appears, and *Post(f)* is the postcondition of f with appropriate local variable substitution.

A. Intuitively, what we're doing is substituting the function precondition for the postcondition.

B. Recall in earlier discussions of formal specification we indicated that there are two methods to ensure that function preconditions are maintained:
   1. Specify explicit exceptions that are thrown by a function.
   2. Verify that a function will never be called if is precondition is false.

C. We're now in a position to see how to do the latter of these two methods.

XXI. Example verification that function `Factorial` is never called with a false precond.

A. Consider the SFP in Figure 4 that calls `fact` in a verifiably correct way.

B. Table 2 shows the details of the proof, top-down.
   1. The proof in the table refers to the predicates **P***n* in Figure 4.
   2. For each proof step there is a brief explanation of how that step is proved.
   3. The gist of the proof is that the proof rule for function call substitutes the Boolean expression *"pre implies post"* as the predicate value for the call.
      a. This substitution is valid if the function itself has been proved, which in this case it has.
      b. Hence, the proof in Figure 4 and Table 2 is guaranteeing that the call to the function never happens when the function's precondition is false, which in this case is done by a direct conditional guard around the call.
      c. Other less direct logic could be used to ensure that the function is never called with a false precond.
      d. The point is, it must always be possible to prove that some code is in place to prevent the call with a false precondition.
      e. If no such logic exists in the program, the proof won't be possible

XXII. Partial versus total correctness

A. The preceding verification methodology demonstrates *partial* program correctness.

B. Partial correctness means that the program is correct, *if and only it terminates*

C. To achieve *total* correctness, we must supply an additional proof of program termination.

D. Such proofs generally involve an induction on one or more program variables.

E. Further discussion of total correctness is beyond the scope of these notes.

XXIII. Verification and programming style

A. In order to make a program verifiable using the simple rules we've discussed thus far, certain stylistic rules must be obeyed.

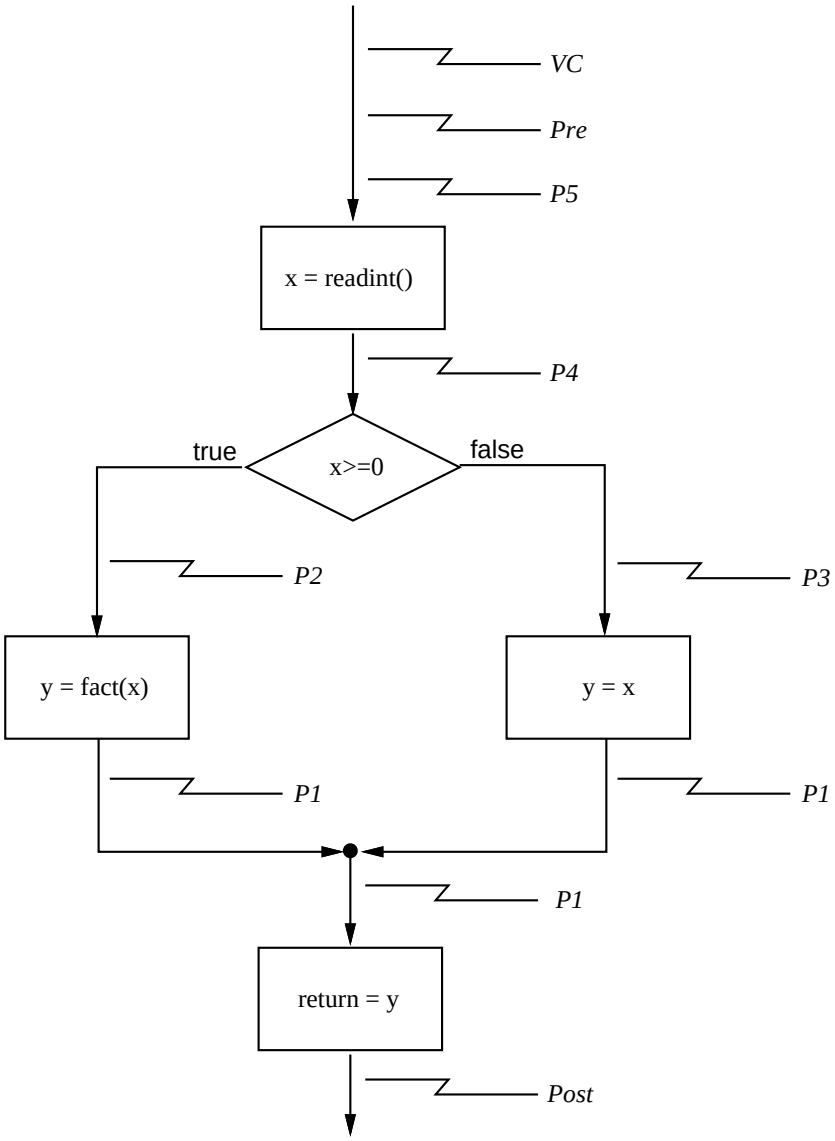


Figure 4: Factorial call proof outline.

Label	Predicate	Proof Step

VC:	`true => forall (x: integer)`
	`  if (x>=0) then x!==x! else x==x`
	`=>`
	`true`
		Rule of verification condition generation
		
		Induction on x

Pre:	`true`	Given
P5:	`forall (x: integer)` `  if (x>=0) then x!==x! else x==x`	Rule of readint

P4:	`if (x>=0) then`
	`  if (x>=0) then x!==x! else x!==x`
	`else`
	`  if (x>=0) then y==x! else x==x`
	`=>`
	`if (x>=0) then x!==x! else x==x`
		Rule of if-then-else
		P2
		
		P3
		
		Simplification

P3:	`if (x>=0) then y==x! else x==x`	Rule of assignment
P2:	`if (x>=0) then x!==x! else x!==x`	Rule of function call
P1:	`if (x>=0) then y==x! else y==x`	Rule of assignment
Post:	`if (x>=0) then return==x! else return==x`	Given

Table 2: Sample proof that Factorial call does not violate precondition.

B. Here is a summary of rules we've assumed thus far
   1. Functions cannot have side effects.
   2. Input parameters cannot be modified in the body of a function. (This is why we added the input N to the implementation of the `Factorial` function earlier.)
   3. A restricted set of control flow constructs is used, i.e., only those constructs for which proof rules exist.

XXIV. Some critical questions about formal program verification.

A. Can it scale up? Fisher says yes, with appropriate tools.

B. Why hasn't it caught on yet? Fisher says for the same reasons that formal specification hasn't caught on yet.

C. When will it catch on? Fisher says:
   1. when software engineers receive adequate training in formal methods, and
   2. when production quality tools become available, and
   3. when software users and customers get sufficiently sick of crappy products and/or when there are sufficient market pressure, including law suits, to force demand for better software.

D. Verification tools include:
   1. formal specification languages
   2. automatic assertion generators
   3. automatic theorem provers

E. Such tools have been developed and studied widely in the research community and have been used by a few commercial developers, notably Boeing.

F. A good example of current work in the area of verification-related tools is the KeY project from the University of Karlsruhe, and others -- http://www.key-project.org/

XXV. An optimistic conclusion -- it *will* happen, when some or all of the above conditions are met.