CSC 530 Lecture Notes Week 8
Wrap Up of Denotational Semantics
Introduction to Axiomatic Semantics

I. Readings: papers 23-33 (many are abstracts only).

II. Tennent Wrap Up

A. You may want to check out remaining sections of Tennent chapter 13 (sections 13.4 - 13.8), but we'll not be covering them in detail. Here are some highlights:

1. 13.4 on continuations describes a mathematically sound, though strained formalism for handling gotos.
   a. The semantics of gotos is so far from what mathematical semantics can handle neatly that the going gets pretty sloggy here.
   b. Gotos really are an *inherently* operational concept that must be shoe-horned to fit the non-operational (denotational) formalism.
2. Sections 13.5 and 13.6 discuss some additional details of semantics for a real language, such as Pascal
   a. 13.5 covers simple but necessary semantic constraints such as identifier usage and scoping.
   b. 13.6 formally defines the data structure domains of Pascal
3. Section 13.7 is a nice summary.
4. Section 13.8.1 provides a nice segue into our coverage of verification semantics; more on this shortly.
5. Sections 13.8.2 and 13.8.3 discuss practical applications of denotational semantics.

B. So, in conclusion, is all the formalism worth it?

1. Once a working group shares a common understanding of the concise notation, things that are bulky to express in other ways come out very nice
2. E.g., expressing difference between static and dynamic binding is notationally trivial (problem 13.9).
3. Both the Tennent and Knuth style semantics provide *excellent* compiler specs.
   a. The problems you are working on in Assignment 3 are precisely the kind of problems that must be solved when writing a real compiler or interpreter.
   b. Formal semantics in some form are an indispensable tool for the serious compiler writer.

III. Relation of axiomatic semantics to attribute and denotational semantics

A. Knuth/Tennent semantics describe the meaning of a programming language in terms that amount to a language translator specification.

1. One might consider this the *internal* semantics of the language in the sense denotational semantics reveal the internal workings of the language at the level of what goes on inside a translator.
2. Knuth semantics do the same thing.

B. Verification-oriented semantics, as represented by the Floyd/Hoare style, describe the meaning of a programming language in terms that amount to a set of rules suitable for proving assertions about particular programs.

1. One might consider this the *external* semantics of the language in the sense that the semantic rules reveal nothing (or not much) about translator-level meaning.
2. Rather, the verification-oriented semantics specify a logical system which can be used to prove assertions about program behavior, as opposed to a set of functions that can be used to build a translator to perform program behavior.

C. The fundamental relationship between axiomatic and denotational is that the soundness of former is proved by appeal to the latter.

That is, for a axiomatic semantics to be sound, we must prove that the axiomatic proof system makes sense vis a vis the language it is to be used with.

1. This requires some form of internal (in the sense above) definition of the language.
2. Section 13.8.1 in Tennent discusses this relationship further.

IV. The two basic components of an axiomatic semantics.

A. A set of *proof rules*, that describe the logical behavior of each construct in a particular programming language.

1. E.g., the rule for an assignment statement describes the *logical* effect of assigning a value to a variable.
2. These proof rules constitute the axiomatic semantics of the programming language.
3. These semantics are comparable to a attribute or denotation semantic definition, but here the orientation is program proving rather than program translation or execution.

B. In conjunction with the proof rules, we must define an overall *verification strategy* that describes how a program proof is constructed using the proof rules applied to a particular program.

1. E.g., for a particular assignment statement in a particular program, the verification strategy defines how to apply the rule for assignment to deliver the meaning of the assignment statement.
2. The verification strategy is comparable to the evaluation strategy defined for an attribute or denotational semantic definition.
3. The key difference with axiomatic semantics versus the other two forms is is that applying the verification rules involves manipulation of boolean predicates rather than manipulation of data-valued variables.
   a. Evaluating an attributed parse tree or denotationally-defined function produces a data store, the contents of which represent the meaning of a program.
   b. In contrast, evaluating (i.e., verifying) an axiomatized program produces a set of boolean expressions -- called *verification conditions*, proof of which constitutes the meaning of the program.

V. Overview of Floyd-style verification

A. The base programming language will be that of simple flowchart programs (SFPs) containing assignment, conditional, looping, and function-call constructs.

B. The axiomatic semantics are defined in terms of SFP constructs.

C. The Floyd-style the verification strategy is as follows:

1. We assert a logical formula at the beginning of the program that specifies what conditions we assume to be true before the program begins execution; call this the program *precondition* or *input predicate.*

2. We assert a logical formula at the end of the program that specifies what conditions obtain after the program has completed execution; this is the logical goal of the program, called the program *postcondition* or *output predicate.*
3. At the top of each program loop, we assert an *invariant condition* that specifies the logical behavior of the loop. (In practice, the derivation of invariant assertions is one of the more non-trivial aspects of the verification process.)
4. We verify that the input predicate implies the output predicate by applying the SFP proof rules using a technique called *backwards substitution*.
5. Example to follow shortly.

VI. Overview of Hoare-style verification

A. The base programming language is in textual form, such as Pascal or C.

B. The axiomatic semantics are defined in terms of PL constructs, in an syntax-directed manner.

C. The Hoare-style verification strategy is essentially the same as the Floyd-style.

1. There are clerical differences between the Floyd and Hoare styles of verification, given that one is based on graphical representation and the other on a textual representation.
2. Initially, the graphical representation is easier to follow.
3. In the graphical proof, we visualize the proof goal by annotating an SFP with formulae.
4. In the equivalent textual proof, we define the proof goal as a *Hoare triple* of the form

   precond {program} 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.

VII. The mechanism for applying proof rules in a Floyd- or Hoare-style proof.

A. The overall verification goal is to prove that the program precondition implies the postcondition *through* the program.

B. As is normal in mathematical proofs, we may work either direction on such a proof.

1. I.e., we may work forwards from the condition of the implication (the precondition) towards the conclusion (the postcondition).
2. Alternative, we may work backwards from the conclusion towards the condition.

C. Empirically, it's easier in program proofs to work backwards.

1. Accordingly, we will use a technique called backwards substitution.
2. Using this technique, we work our way from the postcondition, using the proof rules to "push formulas through" the program.
3. Since each proof rule defines an implication we can make about a particular program construct, we can apply these rules to work our way implicationwise through the program.
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 a termination condition may need to be proved.

D. Proof of termination is a separate, generally inductive proof, that verifies the program terminates on all inputs.

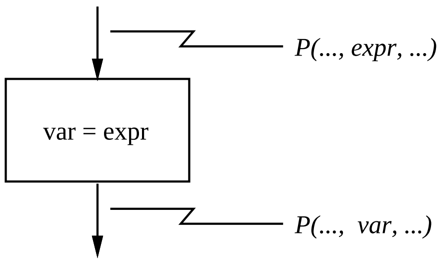
1. Without a termination proof, we achieve *partial correctness*.
2. With a termination proof, we achieve *total correctness*.
3. We'll deal only with partial correctness in these notes.

VIII. SFP proof rules

A. Flowcharts are a helpful representation for understanding formal verification.
   1. There's semantically special about the flowchart representation of programs vis a vis the text representation.
   2. They're in fact isomorphic.

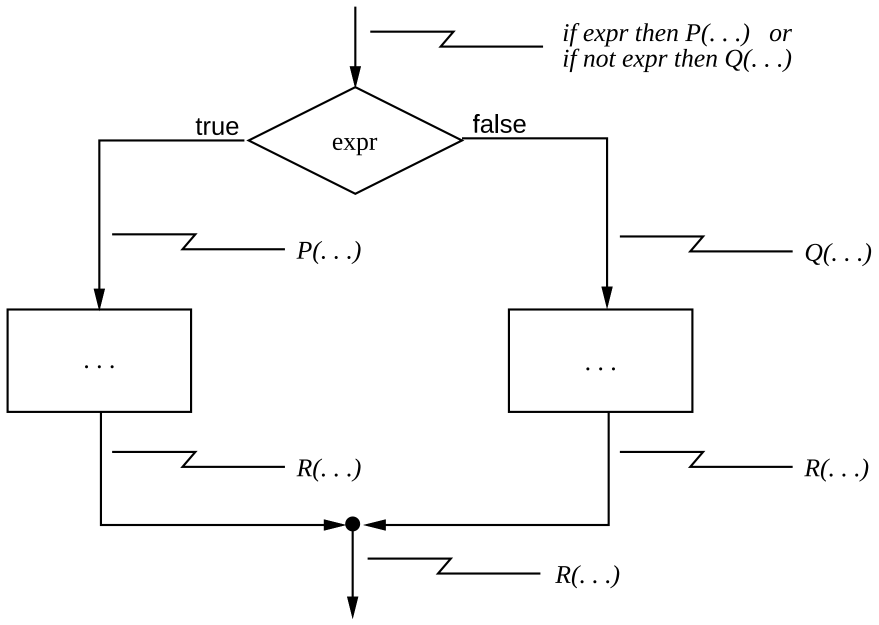
B. We'll examine proof rules for the following basic constructs:
   1. an assignment statement,
   2. an if-then-else statement
   3. a top-of-loop node that is used in conjunction with an if-then-else to form while loops in a flow chart.
   4. a function call
   5. As with our earlier work on operational and axiomatic semantics, these basics pretty well cover the fundamentals of a PL, from which advanced features can be derived.

C. The rule of assignment

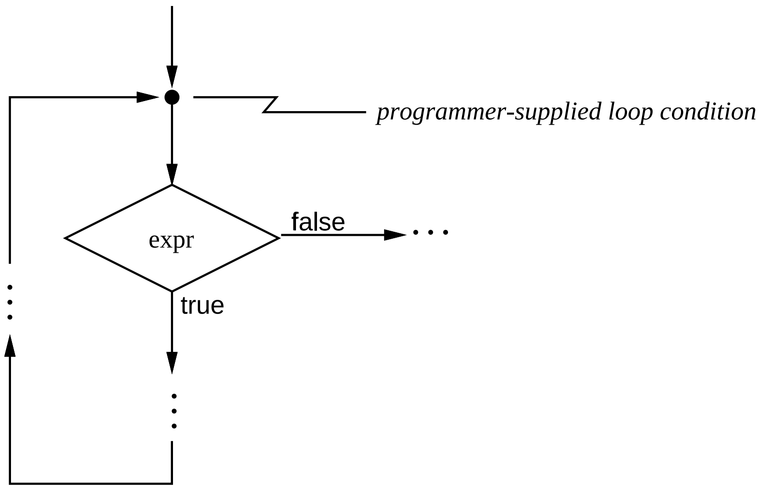


   1. The picture describes the meaning of assignment in terms of variable substitution.
   2. Specifically, the precondition for `var = expr` is derived from the postcondition by systematically substituting all occurrences of `var` in the postcondition with `expr` in the precondition.

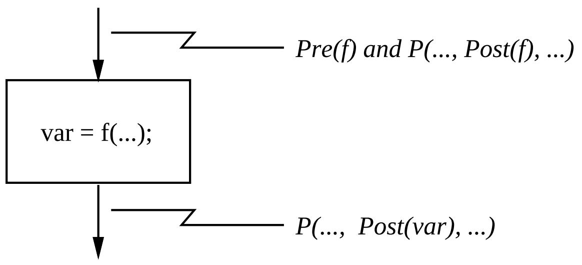
D. The rule of if-then-else



E. The rule for loops



F. The 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.

1. Intuitively, what we're doing is substituting the function precondition for the postcondition.
2. Recall in earlier discussions of formal specification we indicated that there are two methods to ensure that function preconditions are maintained:
   a. Specify explicit exceptions that are thrown by a function.
   b. Verify that a function will never be called if is precondition is false.
3. We're now in a position to see how to do the latter of these two methods.

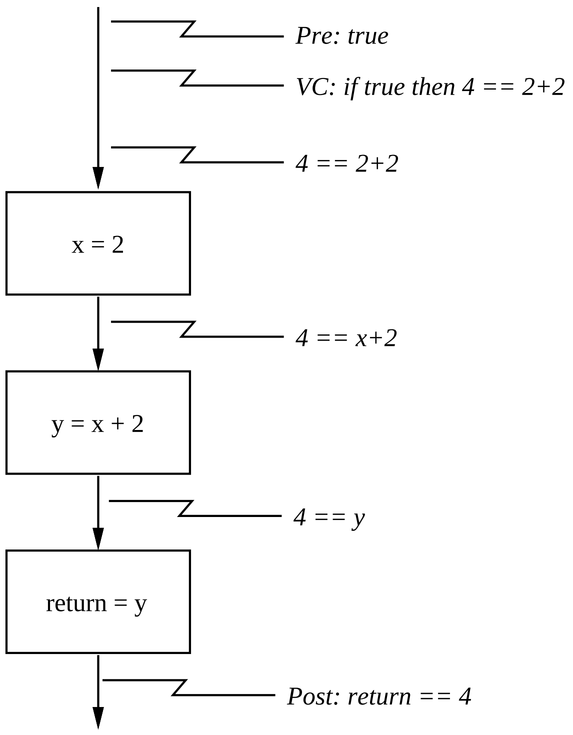
IX. Before we tackle a serious example, let's see how the preceding verification rules can be used to prove that 2+2=4 (a clearly stunning result).

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:



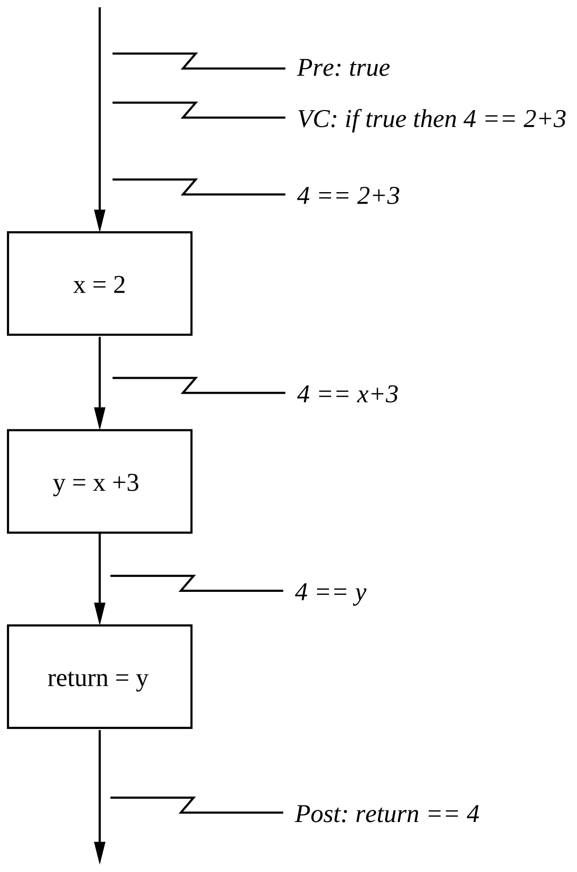
X. The example above concluded with the startling result that a program correctly adds 2+2 to get 4.

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.

XI. 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	$p \supset q$
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).

XII. Now let’s try a proof of a simple factorial example

A. Here’s the 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. Figure 1 outlines Floyd-style proof.

C. Figure 2 outlines the equivalent Hoare-style proof.

XIII. Logical derivation of inductive assertion ‘‘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 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.

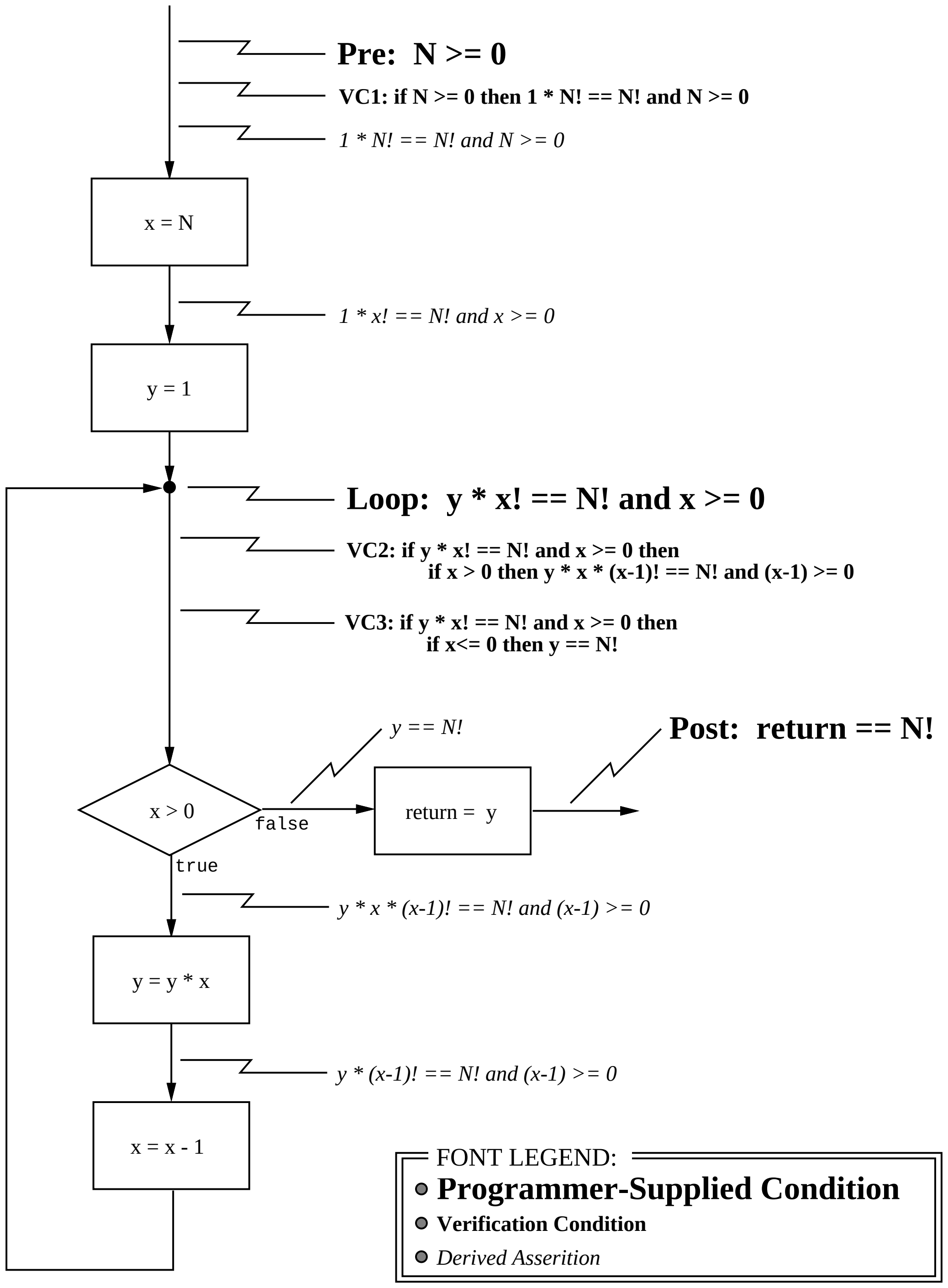

Pre: N >= 0
VC1: if N >= 0 then 1 * N! == N! and N >= 0
1 * N! == N! and N >= 0
x = N
1 * x! == N! and x >= 0
y = 1
Loop: y * x! == N! and x >= 0
VC2: if y * x! == N! and x >= 0 then if x > 0 then y * x * (x-1)! == N! and (x-1) >= 0
VC3: if y * x! == N! and x >= 0 then if x<= 0 then y == N!
y == N!
Post: return == N!
x > 0
false
return = y
true
y * x * (x-1)! == N! and (x-1) >= 0
y = y * x
y * (x-1)! == N! and (x-1) >= 0
x = x - 1
FONT LEGEND:
Programmer-Supplied Condition
Verification Condition
Derived Asserition

Figure 1: Floyd-style factorial proof.

Step	**Logic**	**Rule**
1	$1*x! = N! \wedge x \geq 0$ { $y = 1$ } $y*x! = N! \wedge x \geq 0$	Assmnt
2	$1*N! = N! \wedge N \geq 0$ { $x = N$ } $1*x! = N! \wedge x \geq 0$	Assmnt
3	$1*N! = N! \wedge N \geq 0$ { $x = N; y = 1$} $y*x! = N! \wedge > \geq 0$	Comp(2,1)
4	$N \geq 0 \supset 1*N! = N! \wedge N \geq 0$	VC1
5	$N \geq 0$ { $x = N; y = 1$} $y*x! = N! \wedge x \geq 0$	Conseq(4,3)
6	$y*(x-1)! = N! \wedge x-1 \geq 0$ { $x = x-1$ } $y*x! = N! \wedge x \geq 0$	Assmnt
7	$y*x*(x-1)! = N! \wedge x-1 \geq 0$ {$y = y*x$ } L_6	Assmnt
8	L_7 { $y = y*x; x = x-1$ } R_6	Comp(7,6)
9	$R_6 \wedge x > 0 \supset L_7$	VC2
10	$R_6 \wedge x > 0$ {S_8} R_6	Conseq(9,8)
11	R_6 { while $(x > 0)$ { S_8 } } $R_6 \wedge \neg x > 0$	Iter(10)
12	$N \geq 0$ { S_5 ; S_{11} } $R_6 \wedge \neg x > 0$	Comp(5,11)
13	$R_6 \wedge \neg x > 0 \supset y = N!$	VC3
14	$N \geq 0$ { $x = N; y = 1;$ while $x > 0$ { $y = y*x; x = x-1;$ } } $y = N!$	Conseq(12,13)

where L_i stands for the left part of the ith Hoare triple, R_i stands for the right part of the ith triple, and S_i stands for the statment (middle) part of the ith triple.

Figure 2: Hoare-style factorial proof.

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.

XIV. 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 245) 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

XV. 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
```

XVI. 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 $x \geq 0$ and $x>0 \supset y * (x-1) * (x-1)! = N!$ and $x-1 \geq 0$ ==>

$y * x! = N!$ and $x>0 \supset y * (x-1) * (x-1)! = N!$ (oops)

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

$y * x! = N!$ and $x \geq 0$ and $\neg (x \geq 0) \supset y = N!$ ==>

$y * x! = N!$ and $x \geq 0$ and $x<0 \supset y = N!$

XVII. Automatic inductive assertion generation via symbolic evaluation

A. A mechanical technique for generating loop assertions is to apply the idea of symbolic evaluation, which means to evaluating a program with symbolic rather than actual data values.

B. For example, starting with the output predicate symbolically evaluating the factorial loop looks like this:

y = N!
↓
y = N!
↓
y * x = N!
↓
y * (x-1) = N!
↓
y * x * (x-1) = N!
↓
y * (x-1) * (x-1-1) = N!
↓
y * x * (x-1) * (x-2) = 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. It’s also interesting to look at the erroneous case where the loop statements have been transposed:

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

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

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

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

A. Consider the SFP in Figure 3 that calls fact in a verifiably correct way.

B. Table 1 shows the details of the proof, top-down.

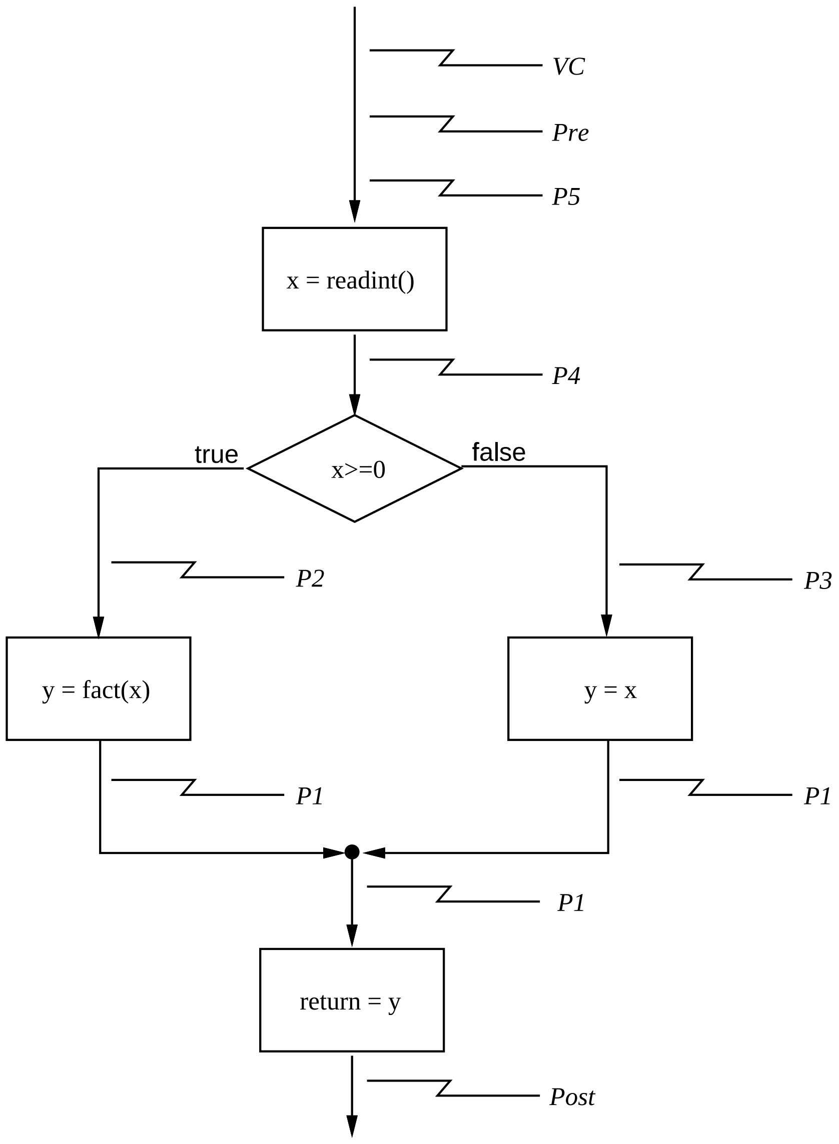


Figure 3: Factorial call proof outline.

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

5:	`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 1: Proof that Factorial call does not violate precondition.

XIX. 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.

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 must be used, i.e., only those constructs for which proof rules exist.

XX. Some critical questions about formal program verification.

A. Question: Can it scale up?
Answer: Yes, with appropriate tools.

B. Question: Why hasn't it caught on (yet)?
Answer: For a variety of reasons, not the least of which are *cultural* (cf. the Perlis paper).

C. Question: Will it ever catch on?
Yes, when
1. software engineers receive adequate training in formal methods, and
2. software users and customers get sufficiently sick of crappy products, and
3. production quality tools become available, such as the following:
   a. formal specification languages

   b. automatic assertion generators
   c. automatic theorem provers

D. Such tools have been developed and studied widely in the research community and at a handful of commercial locations.