CSC 530 Lecture Notes Week 8

Wrap Up of Denotational Semantics

Introduction to Axiomatic Semantics

I. **Readings: papers 23-33.**

II. Tennent Wrap Up

A. Check out remaining sections of ch 13 (sections

B. Is all the formalism worth it?

III. Relation of axiomatic to attr and denotational semantics

A. Knuth/Tennent semantics amount to translator spec.

B. Verification-oriented semantics suitable for proving programs.

C. Soundness of axiomatic def appeals to denotational def.

IV. Basic components of axiomatic def

A. Set of *proof rules*

B. A *verification strategy*

V. Floyd-style verification

A. Base PL is SFPs

B. Semantics defined for SFP constructs.

C. Floyd-style verification strategy:

Floyd-style verification, cont'd

1. Assert *precondition*
2. Assert *postcondition*
3. Assert *invariant condition* for each loop.
4. Verify that precond implies post-cond via *backwards substitution*.

VI. Hoare-style verification

A. Base PL is textual.

B. Semantics defined syntax-directed.

C. Hoare-style strategy essentially same as Floyd, denoted with *Hoare triple* of the form

precond {program} postcond

VII. Applying proof rules

A. Goal to prove precond implies post-cond *through* the program.

B. May work either direction

C. Easier to work backwards, using back-wards substitution.

D. Proof of termination is separate

VIII. SFP proof rules

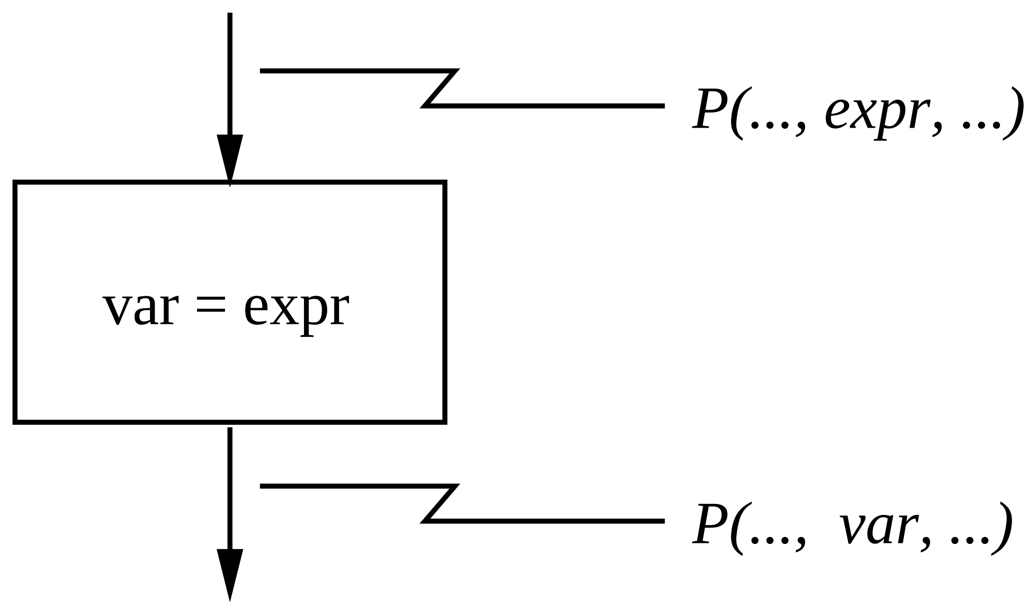
A. Flowcharts are helpful

B. We’ll examine basic constructs:

1. assignment
2. if-then-else
3. top-of-loop node
4. function call

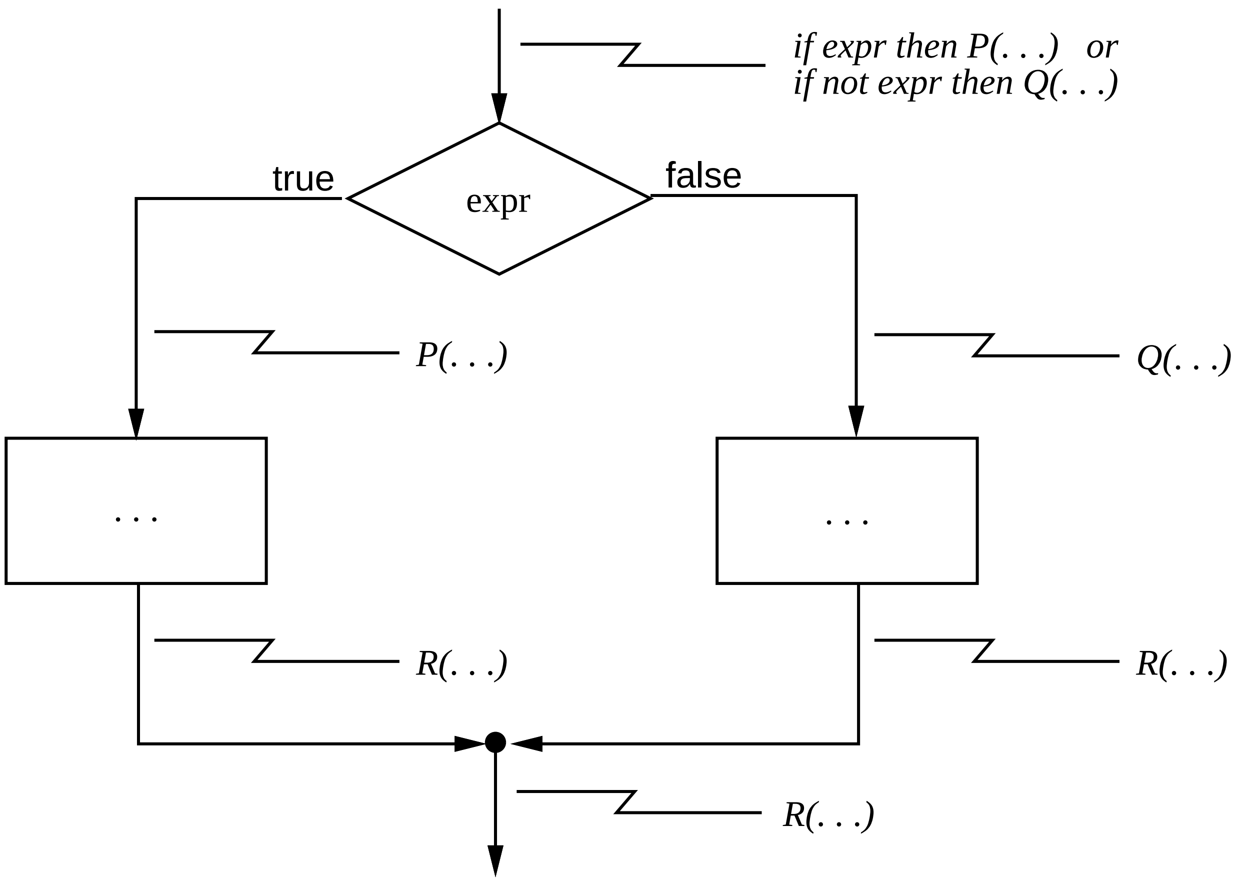
SFP proof rules, cont'd

C. Rule of assignment



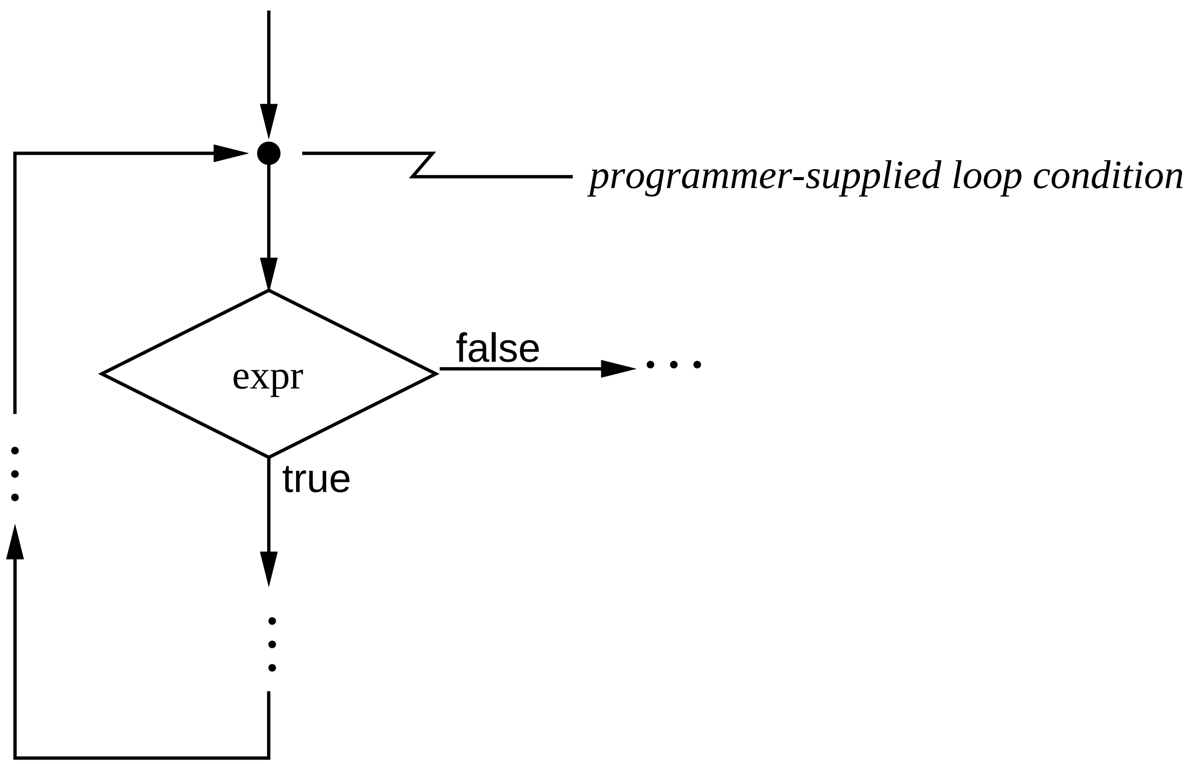
SFP proof rules, cont'd

D. Rule of if-then-else



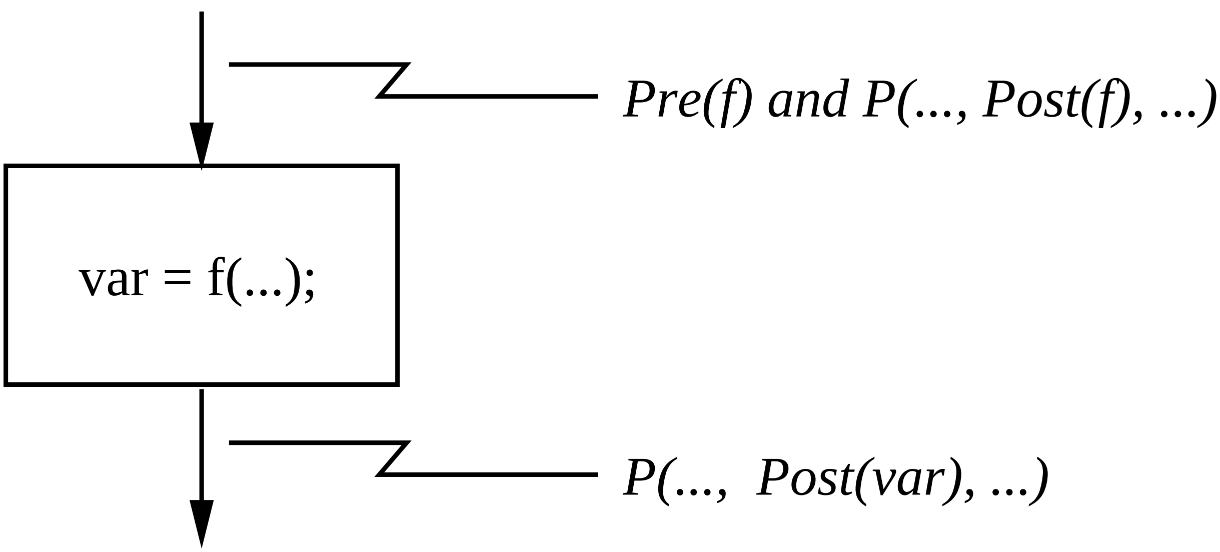
SFP proof rules, cont'd

E. Rule for loops



SFP proof rules, cont'd

F. The rule for function calls:



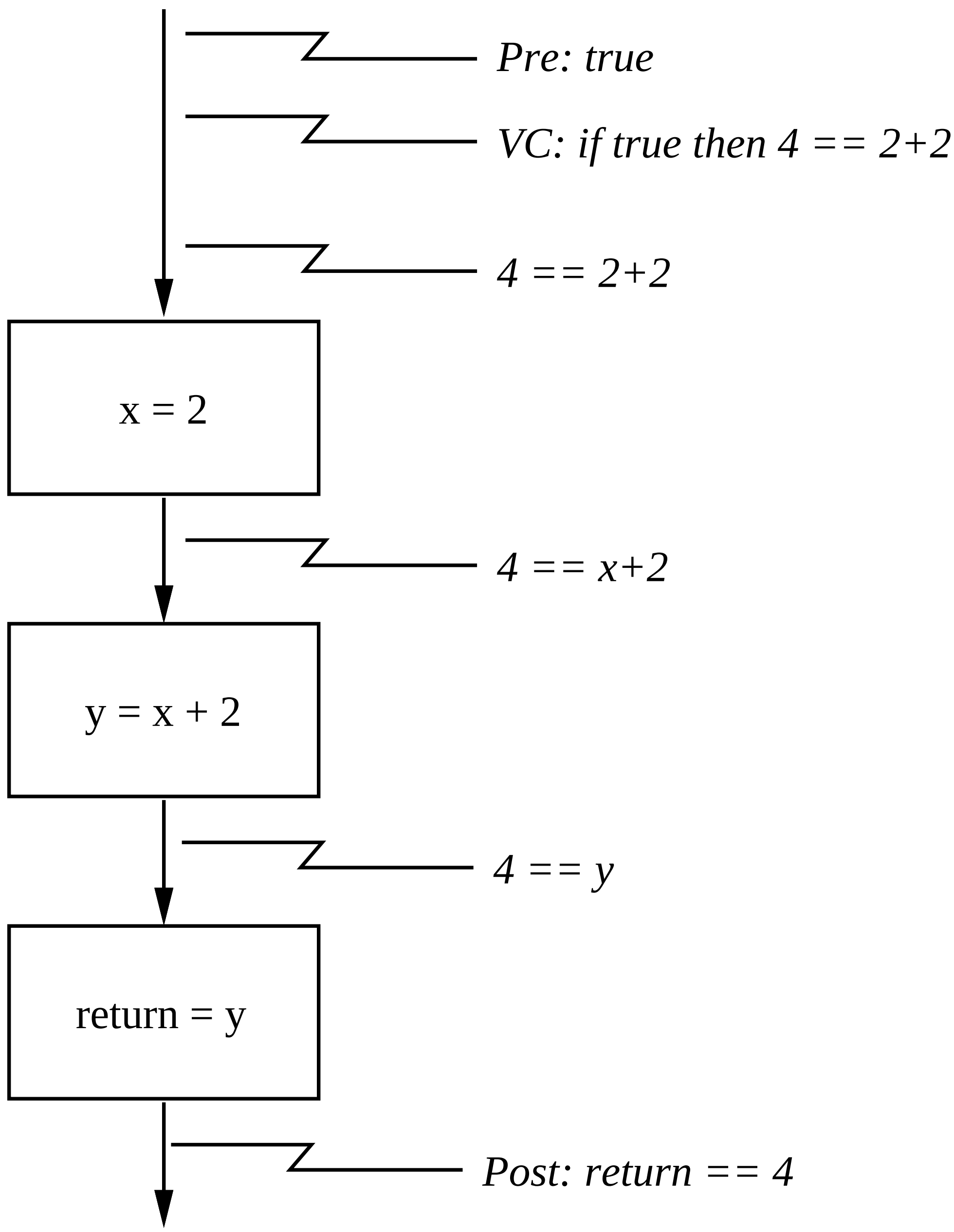
IX. A stunning result

A. Here's the program:

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

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

Stunning result, cont'd

B. Here's the SFP:


Pre: true
VC: if true then 4 == 2+2
4 == 2+2
x = 2
4 == x+2
y = x + 2
4 == y
return = y
Post: return == 4

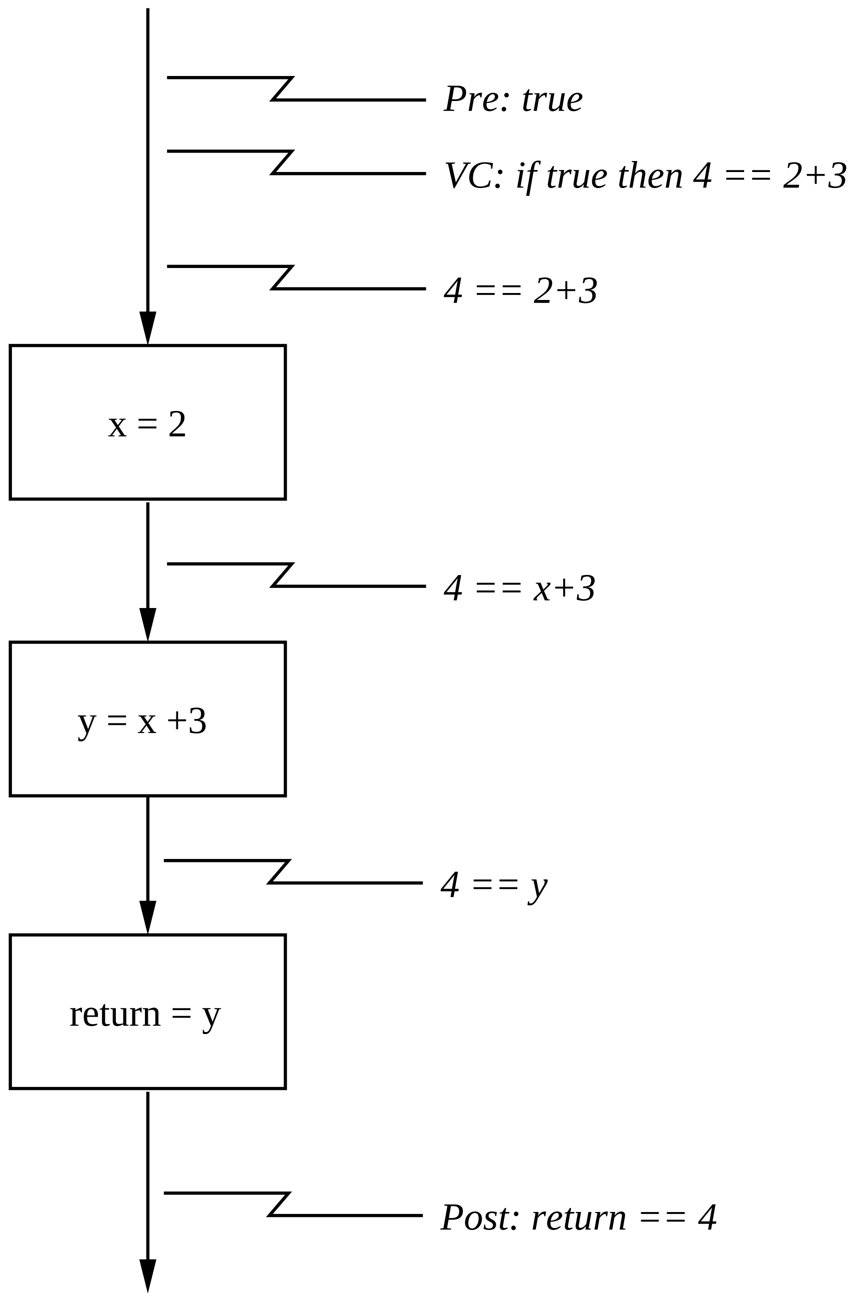
X. A stunned result

A. Let's try to prove

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

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

Stunned result, cont’d

B. Here’s the proof attempt


Pre: true
VC: if true then 4 == 2+3
4 == 2+3
x = 2
4 == x+3
y = x +3
4 == y
return = y
Post: return == 4

Stunned result, cont'd

C. We are left with the VC

$$\text{true} \supset 4 == 2 + 3 \quad ==>$$
$$\text{true} \supset \text{false}$$

which is false.

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

XI. Implication proofs

A. Recall truth table for logical implication.

B. $p \supset q$ is only false if p is true and q is false.

C. In a program verification, we assume p is true.

D. Hence, VC will fail to be proved is if q is false.

XII. Proof of Factorial example.

A. The definition:

```
int Factorial(int N) {
/*
 * Compute factorial of x,
 * for positive x, using
 * an iterative technique.
 *
 * pre: N >= 0
 *
 * post: return == N!
 *
 */
```

Proof of Factorial, cont'd

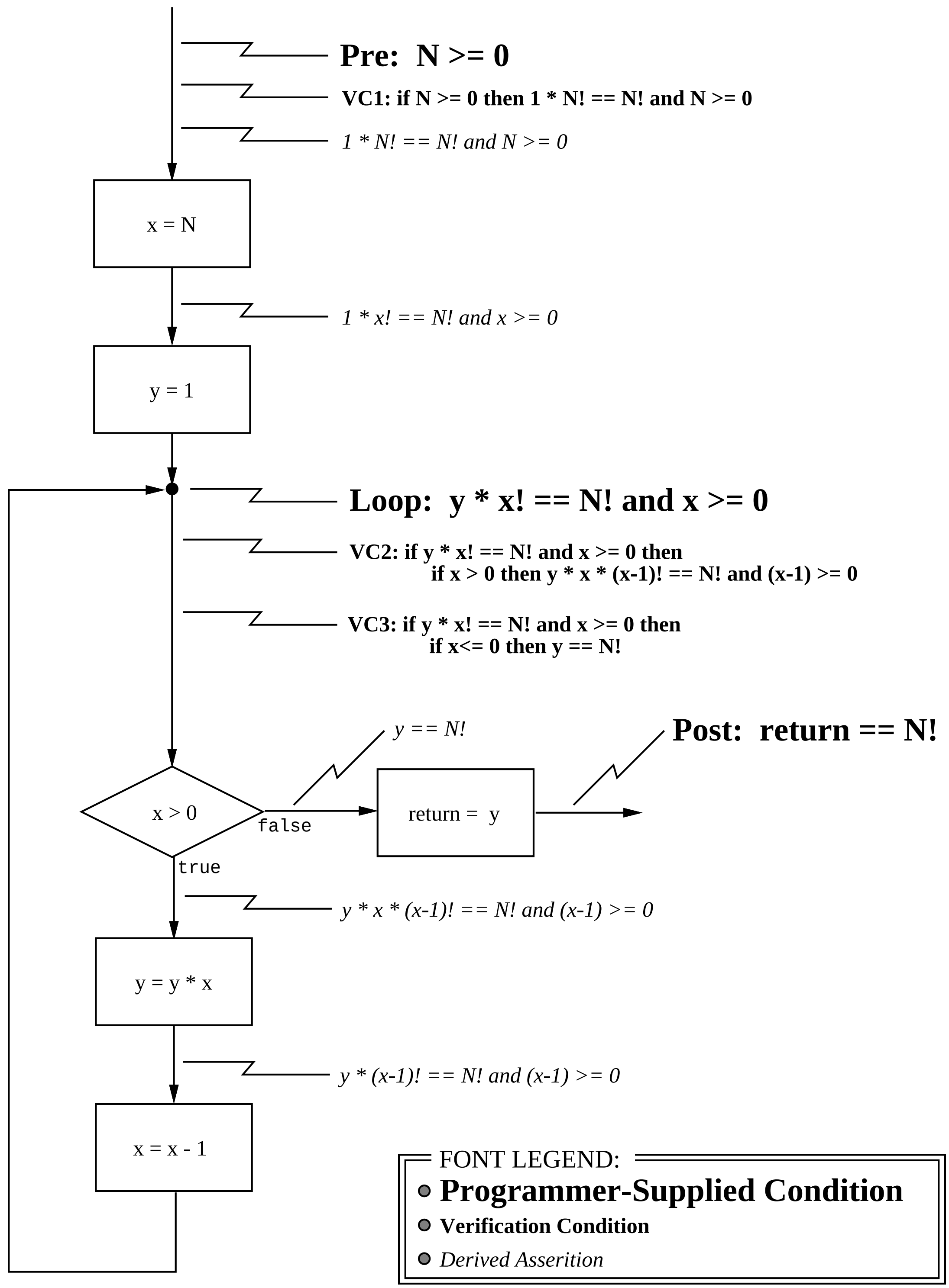
```
        int x,y; /* Temp vars */

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

Proof of Factorial, cont'd

B. Figure 1 outlines Floyd-style proof

C. Figure 2 outlines Hoare-style proof

Proof of Factorial, cont’d


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 Assertion

XIII. Logical derivation “y * x! = N!”

XIV. Further tips on doing the proofs

XV. Factorial (VC's)

A. Obligated to prove each VC

B. 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. Possible errors in factorial

A. Transpose loop body statements.

B. We'll get 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. "x ≥ 0" (instead of strictly > 0)

$y * x! = N!$ 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 assertions

A. A mechanical technique

B. Looks like this:

Automatic inductive assertions, cont'd

$$y = N!$$
$$\downarrow$$
$$y = N!$$
$$\downarrow$$
$$y * x = N!$$
$$\downarrow$$
$$y * (x-1) = N!$$
$$\downarrow$$
$$y * x * (x-1) = N!$$
$$\downarrow$$
$$y * (x-1) * (x-1-1) = N!$$
$$\downarrow$$
$$y * x * (x-1) * (x-2) = N!$$
$$\downarrow$$
$$\vdots$$
$$\downarrow$$
$$y * x * (x-1) * \ldots * (x-N) = N!$$

Automatic inductive assertions, cont'd

C. Inspecting the result, notice relationship y * x! = N!.

D. Also interesting to look at the erroneous case

Automatic inductive assertions, cont’d

y = N!

↓

y * x = N!

↓

y * (x-1) = N!

↓

y * x * (x-1) = N!

↓

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

↓

.

.

.

↓

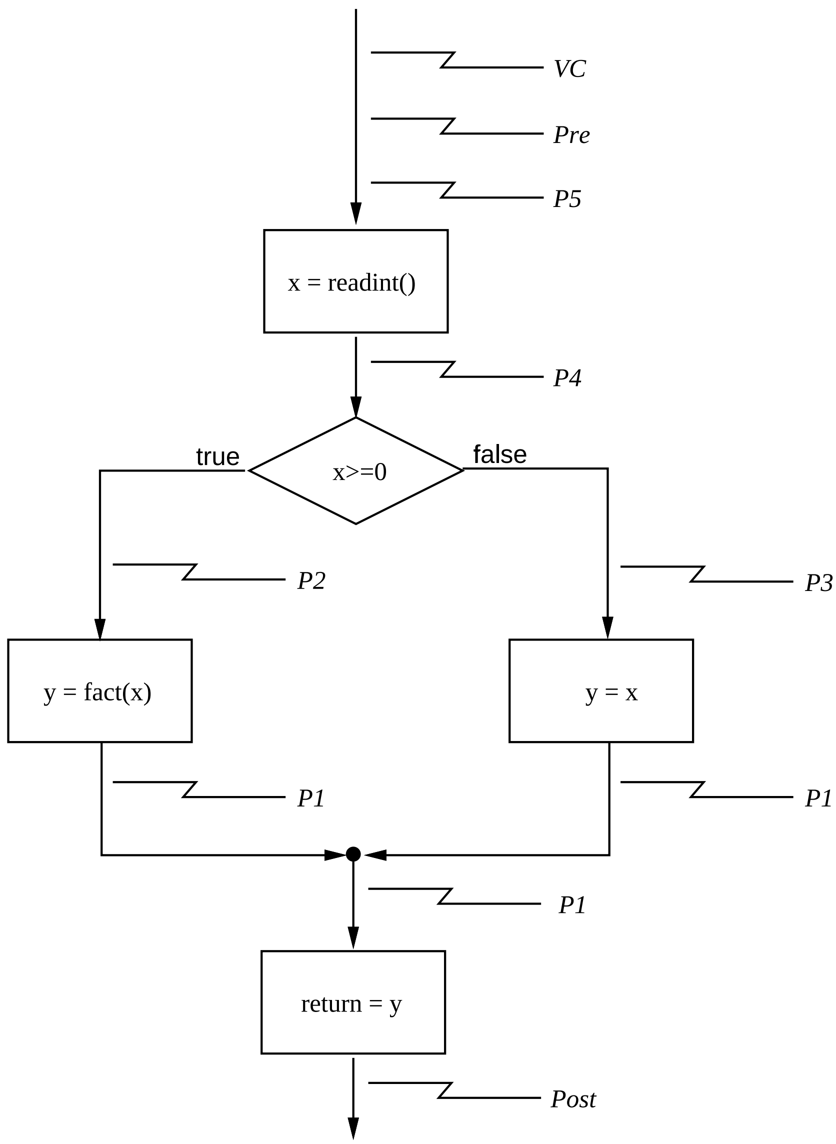
y * (x-1) * (x-2) * ... * (x-N) = N!

Automatic inductive assertions, cont'd

E. In erroneous case, symbolic eval leads to wrong loop assertion.

F. This will ultimately cause the verification to fail.

XVIII. Factorial is never called with false precond.



Details of the proof

```
Label       Predicate
---------------------------------------------------------------
VC:     true => forall (x: integer)                          Rul
            if (x>=0) then x!==x! else x==x                    c
        =>
        true                                                 Inc

Pre:    true                                                 Giv

P5:     forall (x: integer)                                  Rul
            if (x>=0) then x!==x! else x==x

P4:     if (x>=0) then                                       Rul
            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                      Sim

P3:     if (x>=0) then y==x! else x==x                       Rul

P2:     if (x>=0) then x!==x! else x!==x                     Rul

P1:     if (x>=0) then y==x! else y==x                       Rul

Post:   if (x>=0) then return==x! else return==x             Giv
```

XIX. **Verification & program style ...**

XX. **Critical questions**

A. Question: Can it scale up?

B. Question: Why hasn’t it caught on (yet)?

C. Question: Will it ever catch on?