CSC 509 Lecture Notes Week 9, Part 2

Details of
Formal Program Verification

I. Introductory definitions

I. Introductory definitions

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

I. 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. The problems with testing

II. **The problems with testing**

A. Cannot cover all possible cases

II. **The problems with testing**

A. Cannot cover all possible cases

B. Never 100% sure that system is correct.

II. **The problems with testing**

A. Cannot cover all possible cases

B. Never 100% sure that system is correct.

C. For some systems, this is not good enough.

II. **The problems with testing**

A. Cannot cover all possible cases

B. Never 100% sure that system is correct.

C. For some systems, this is not good enough.

D. Enter program verification.

III. **Practical applications.**

A. ***Proof-carrying code.***

1. There are potential problems with code sent between machines.

   a. Code wants to run on foreign host.

   b. Host wants to know if code works properly.

Proof-carrying code, cont'd

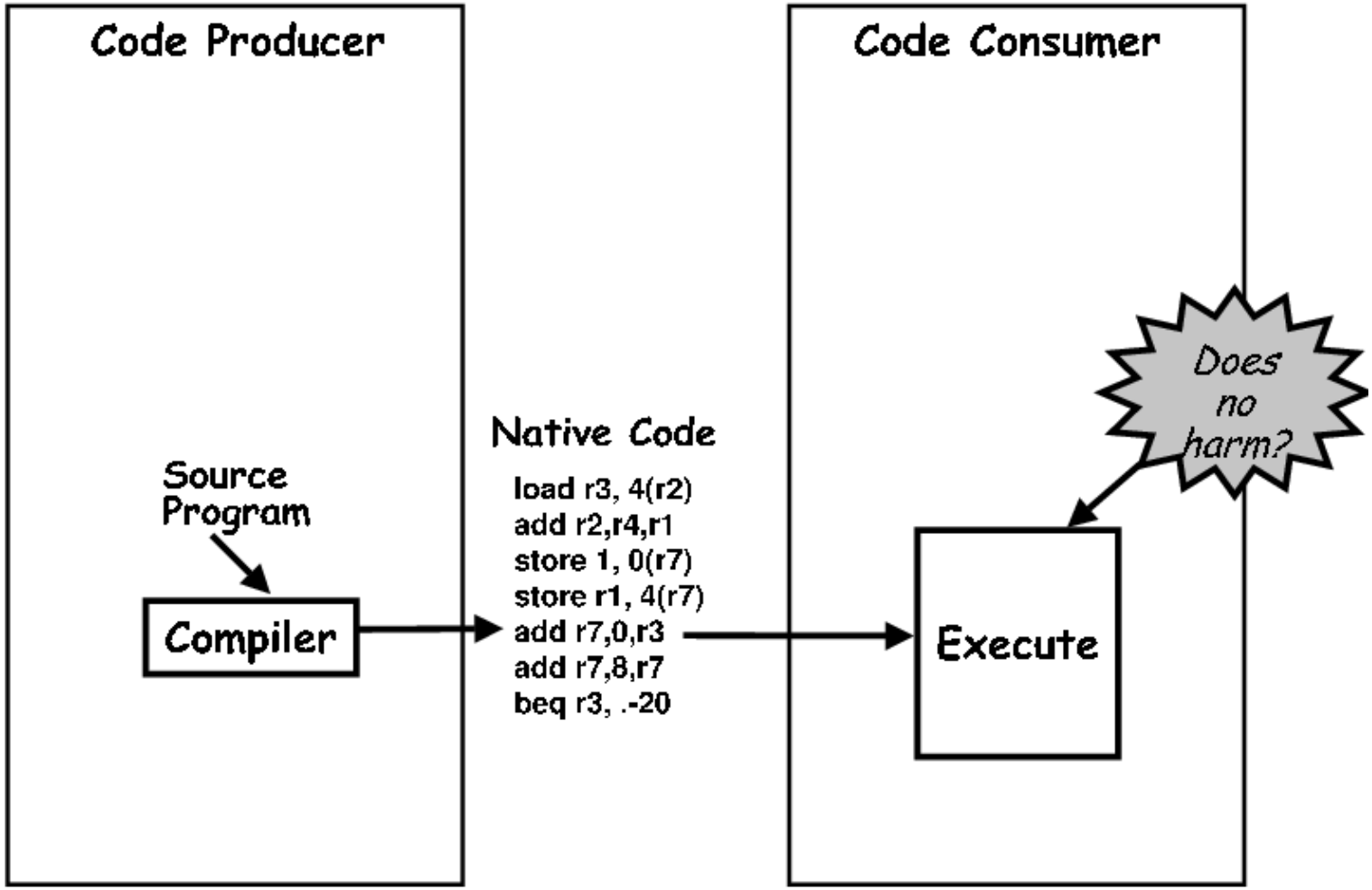
2. Terminology:

   a. *Code producer* has code that wants to run on foreign host.

   b. *Code consumer* is the host.

   c. Code producer may violate *policies* of code consumer.

Proof-carrying code, cont'd



Proof-carrying code, cont'd

3. To solve problem:

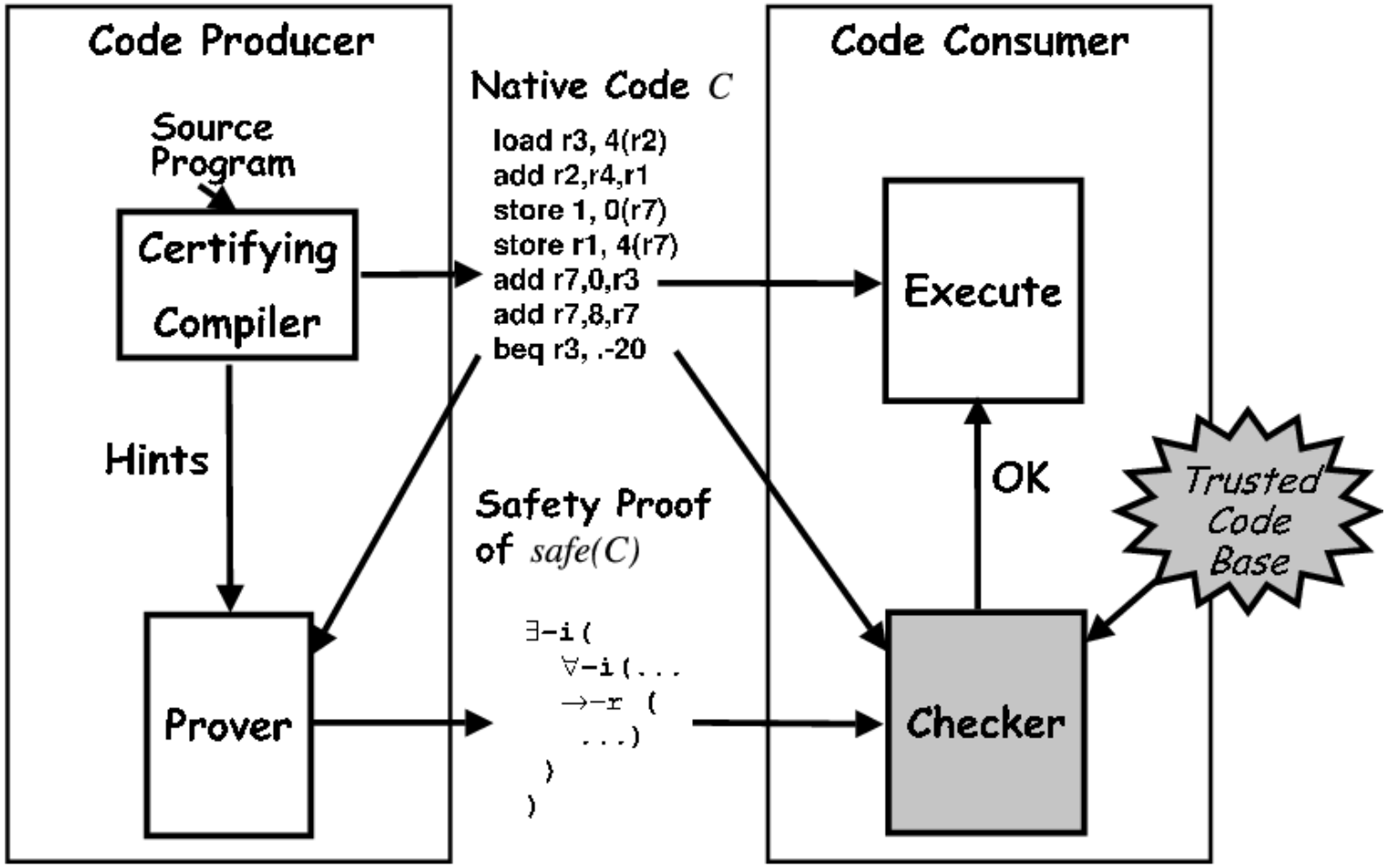
   a. Code producer compiles *and proves* code.

   b. Proof based on formal policies, defined by consumer.

   c. Producer sends code to consumer.

   d. Consumer checks that proof still holds.

Proof-carrying code, cont'd



B. ***Model checking.***

1. Large software exhibits complex behavior.

2. Idea is to prove properties of a model before it's implemented.

3. Noteworthy recent work in avionics.

4. E.g., Rushby's proof of redundancy model related to 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.

Formalizing mental models, cont'd

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.

Formalizing mental models, cont'd

4. Formal methods used for this problem:
   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 system.

Formalizing mental models, cont'd

5. 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. From the Sublime to the Trivial ...

A. Previous examples address real problems.

B. Proofs are non-trivial.

C. But how the heck do they really work?

D. We'll have a look at a very simple example.

V. A very simple example function.

```
/*
 * Compute factorial of x, for
 * positive x, using iteration.
 *
 * pre: x >= 0
 *
 * post: return == x!
 *
 */
```

Example, cont'd

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

Question: *Is this correct?*

VI. **Symbolic evaluation**

A. In testing schemes, inputs and outputs are *concrete* values.

B. Consider how we'd test factorial.

C. Table 1 shows typical unit test plan.

Symbolic evaluation, cont'd

Test No.	Input	Expected Results
1	x = -1	ERROR
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

Symbolic evaluation, cont'd

D. To test, feed in concrete values and check results (which reveals the bug).

E. Two important questions ...

1. Where do we get expected results?

2. Does it work *for all* inputs?

Symbolic evaluation, cont'd

F. One way to answer these questions is to use *symbolic* input and output.

1. Instead of concrete values for input x, just use the symbol "x".

2. Run the program several times to see what ***symbolic formula*** emerges.

Symbolic evaluation, cont'd

G. We'll use the corrected program:

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

Symbolic evaluation, cont'd

H. Details (again, for *corrected* program):

1. Start with symbolic input value "`x`".

2. Then start running the function body:

```
y = 1;
while (x > 0) { // true symbolically
    y = y * x;
```

which gives symbolic value of $y = 1 * x$,
which simplifies to just $y = x$.

Symbolic evaluation, cont'd

3. Some more symbolic computation:

```
        x = x - 1;
    }
    while (x > 0) {   // true symbolically
        y = y * x;
```

4. Results in symbolic value *x * (x - 1)* for y.

Symbolic evaluation, cont'd

5. A bit more

```
        x = x - 1;
    }
    while (x > 0) {   // true symbolically
        y = y * x;
```

Symbolic evaluation, cont'd

which results in y's symbolic value of

$x * (x - 1) * ((x - 1) - 1)$

which simplifies to

$x * (x - 1) * (x - 2)$

Symbolic evaluation, cont'd

6. The idea is we treat input values as *symbols*, not concrete values.

Symbolic evaluation, cont'd

I. After ***N*** times through the loop:

$$y = 1$$
$$\downarrow$$
$$y = 1 * x$$
$$\downarrow$$
$$y = x * (x-1)$$
$$\downarrow$$
$$y = x * (x-1) * (x-1-1)$$
$$\downarrow$$

In factorial, cont'd

↓

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

↓

...

after N times through factorial loop symbolically

↓

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

Symbolic evaluation, cont'd

J. An informative symbolic pattern develops.

K. Also interesting is the erroneous case.

Symbolic evaluation, cont'd

$$y = 1$$
$$\downarrow$$
$$y = 1 * (x-1)$$
$$\downarrow$$
$$y = (x-1) * ((x-1)-1)$$
$$\downarrow$$
$$y = (x-1) * (x-2) * (x-3)$$
$$\ldots$$
$$\downarrow$$
$$y = (x-1) * (x-2) * \ldots * (x-1-N)$$

Symbolic evaluation, cont'd

L. Here an incorrect formula emerges.

M. Symbolic evaluation by hand is way tedious

N. A number of automated tools exist, e.g, the KeY project from Karlsruhe university:

/www.key-project.org/

VII. **Moving on to formal verification**

A. Symbolic eval involves informal analysis.

B. We want mathematical certitude.

C. I.e., a proof that program meets its spec.

On to formal verification, cont'd

D. General steps:

1. Define *axiomatic* semantics for programming language.

2. Define general procedure to assign meaning a program.

On to formal verification, cont'd

E. Given semantics and verification procedure, state formal pre and post conditions for all functions (i.e., methods).

On to formal verification, cont'd

F. The desired result is

pre $\supset$ post, *through the function*

G. New notation is

pre {function body} post

H. Called a "Hoare triple".

On to formal verification, cont'd

I. Final step is to prove *termination condition* (more later).

J. We will now look at a set of verification rules for a very simple programming language.

VIII. **Simple Flowchart Programs**

A. Graphical flowchart form.

B. Helpful form for presenting proof rules.

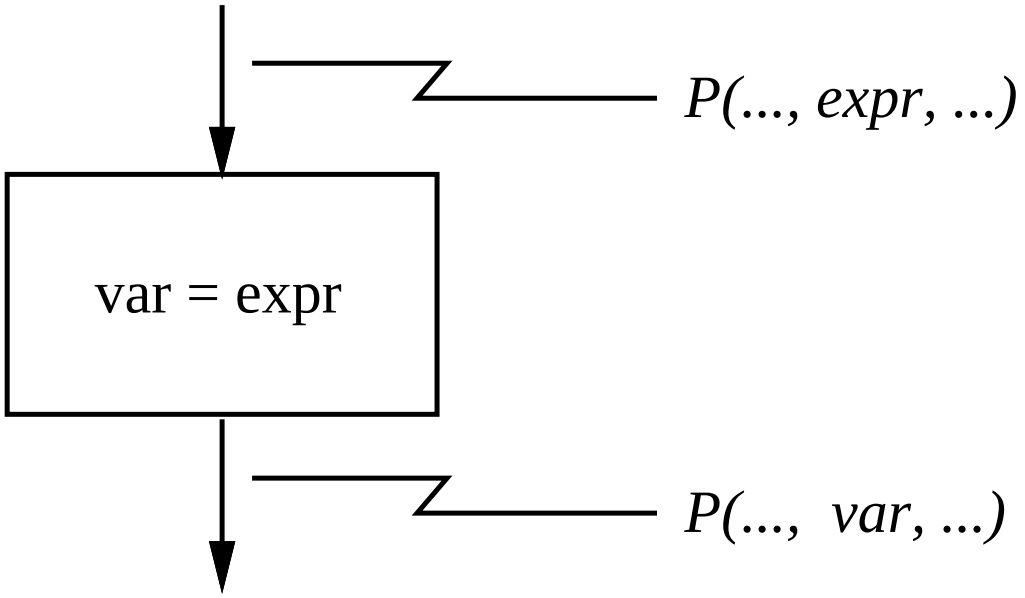
Simple Flowchart Programs, cont'd

C. Basic constructs are:

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

IX. Semantic rules for SFPs

A. The rule of assignment



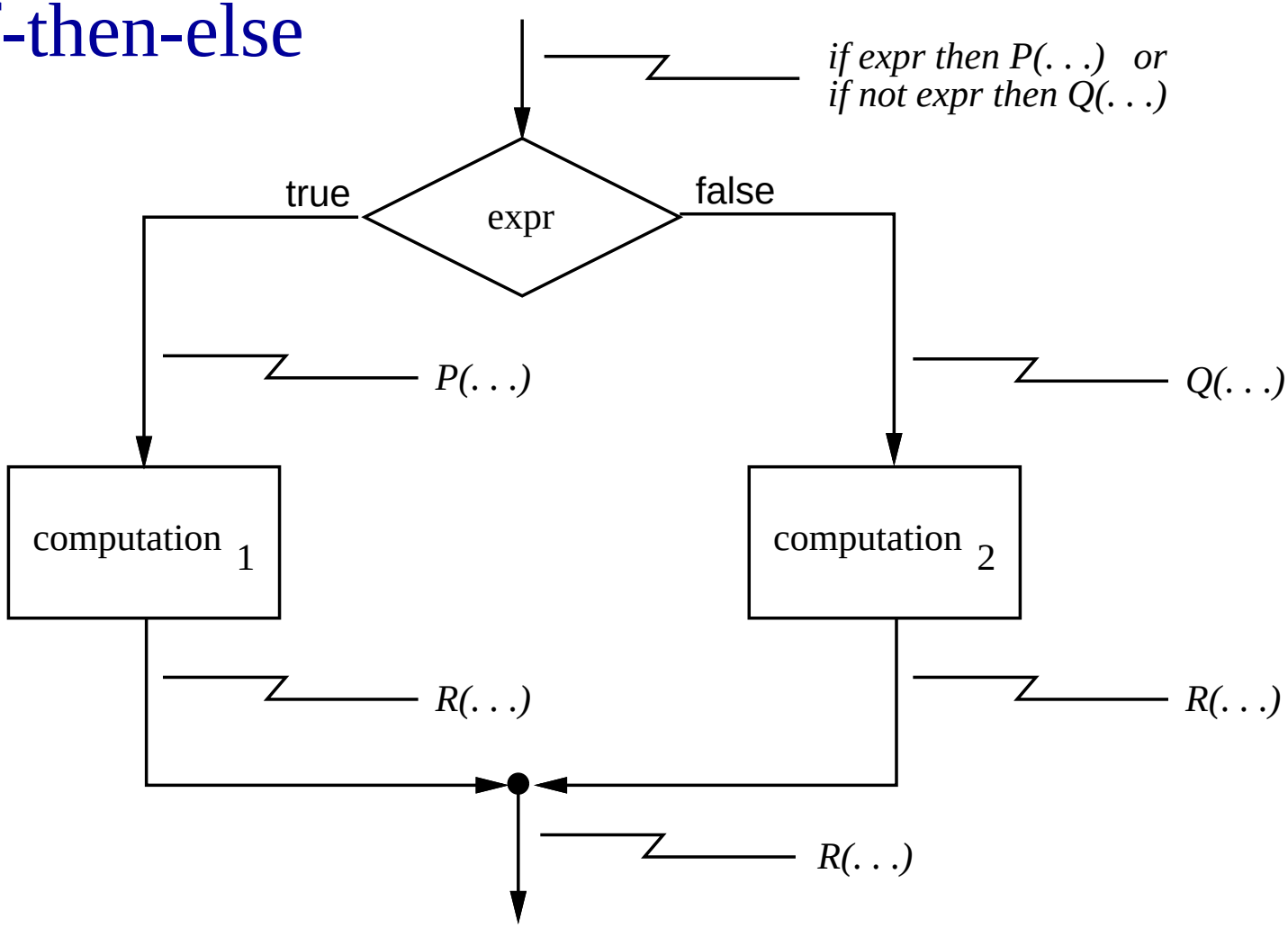
Semantic rules, cont'd

1. Defines meaning in terms of variable substitution.

2. Precond is derived from postcond by systematically substituting `var` with `expr`.

Semantic rules, cont'd

B. if-then-else



Semantic rules, cont'd

1. If-then-else defined as logical implication.
2. Syntactically sugared logical implication,

   if X then Y

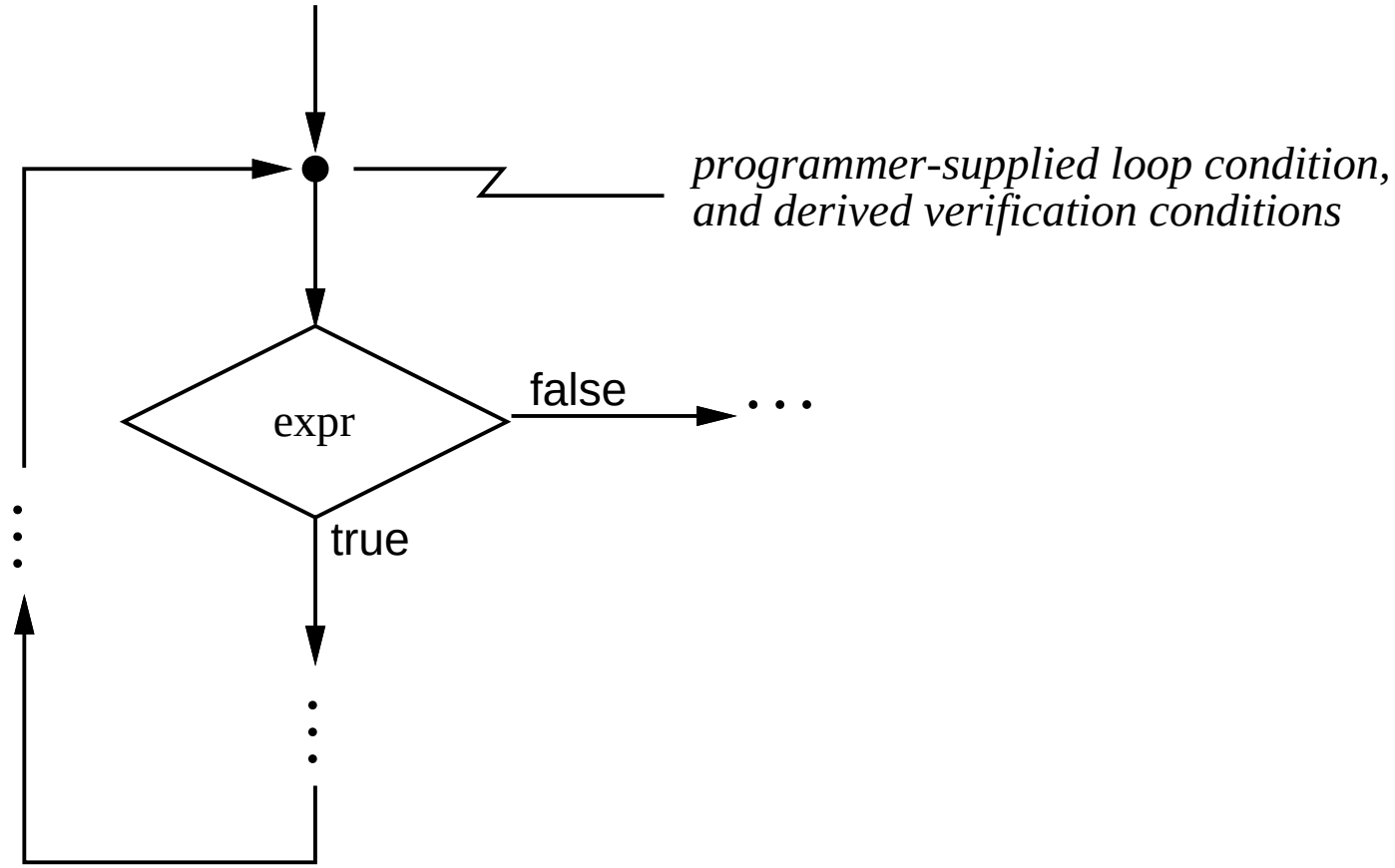
   is equivalent to

   X implies Y

3. Predicates *P(...)*, *Q(...)* derived from *R(...)* by applying proof rules for computation$_1$ and computation$_2$, resp.

Semantic rules, cont'd

C. The rule for loops



Semantic rules, cont'd

1. Loop rule requires programmer to supply *loop invariant*.

2. It's in addition to pre- and postconds.

3. Invariant is true throughout loop body.

4. Stated in terms of variables used and modified in loop body.

X. Application of semantic rules

A. Goal is to prove

pre {function body} post

B. Precond implies postcond *through* body.

C. Semantic rules allow us to *push predicates through* a program.

XI. Backwards substitution technique

A. A kind of symbolic evaluation.

B. Evaluating predicates rather than values.

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

Backwards substitution, cont'd

D. The steps:

1. Annotate program with pre and post conds.
2. At each loop, provide invariant.
3. Take postcond and *push it through* program.

Semantic rules, cont'd

4. When a "pushed-through" predicate "runs into" a supplied predicate, we have a *verification condition (VC)*.

5. After all VCs are proved, program proof is complete, except for termination.

6. We won't deal with proof of termination.

XII. A stunning result

A. Here's the program:

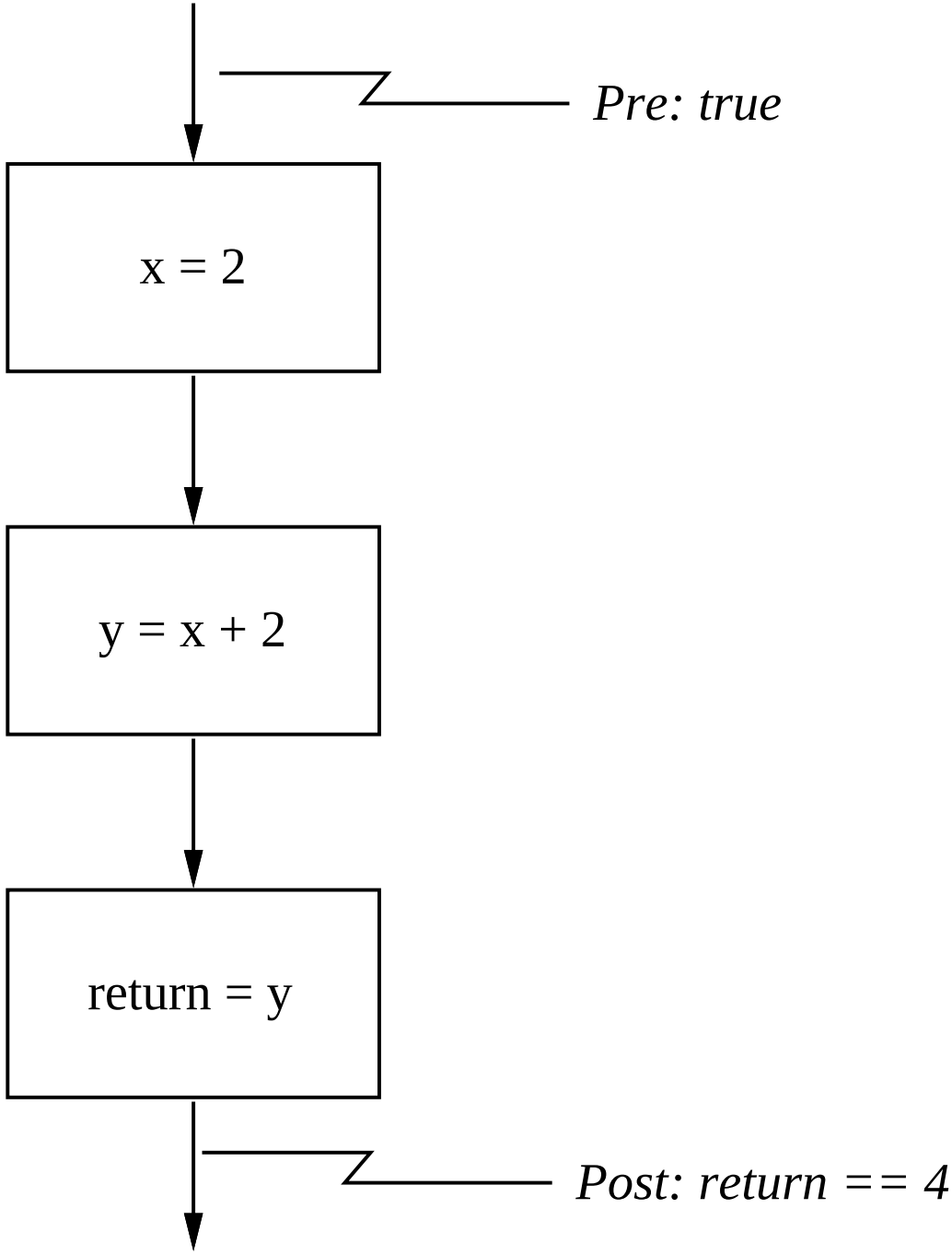
```
int Duh() {
    /*
     * Add 2 to 2 and return
     * the result.
     *
     * pre: ;
     * post: return == 4;
     *
     */
```

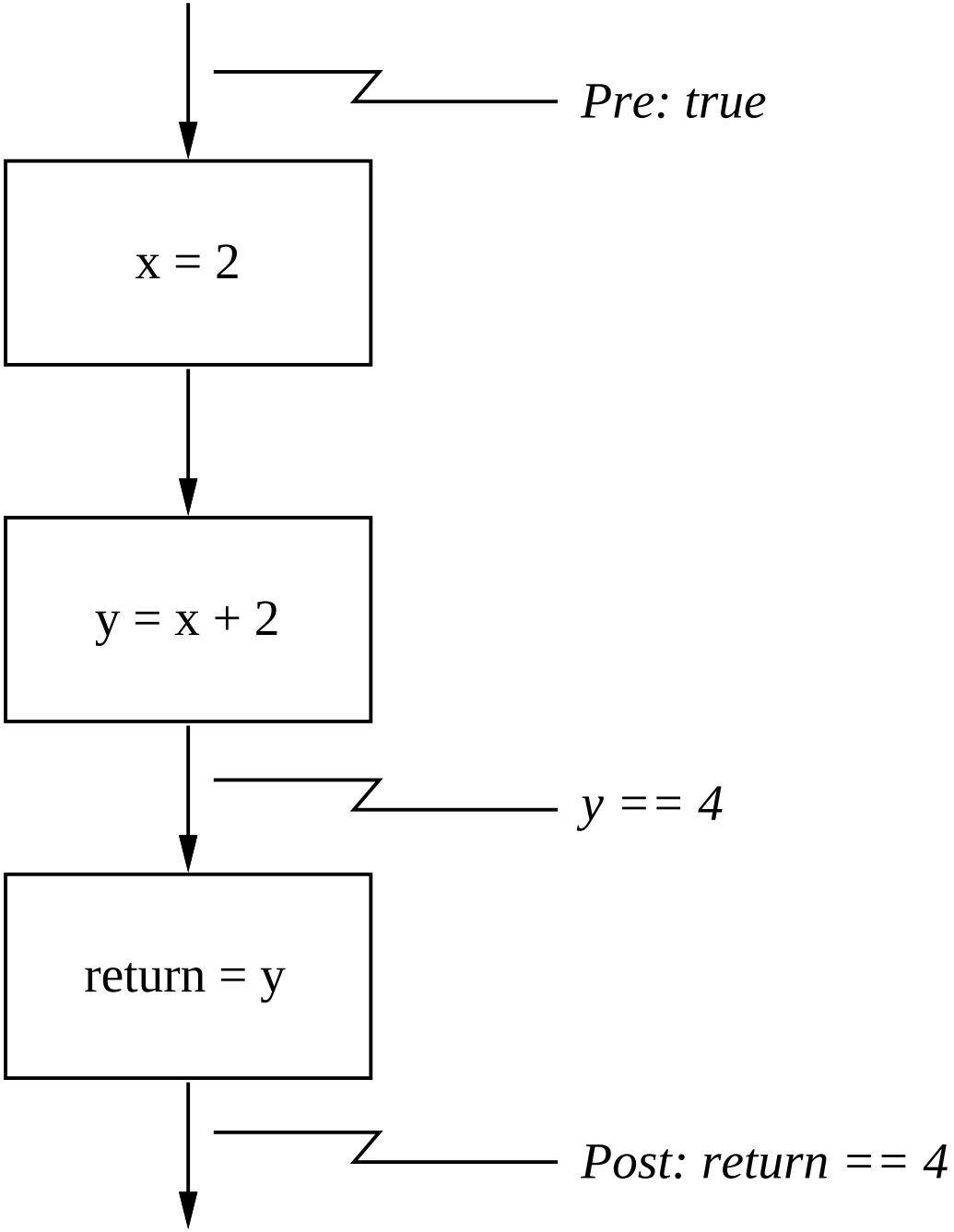
Stunning result, cont'd

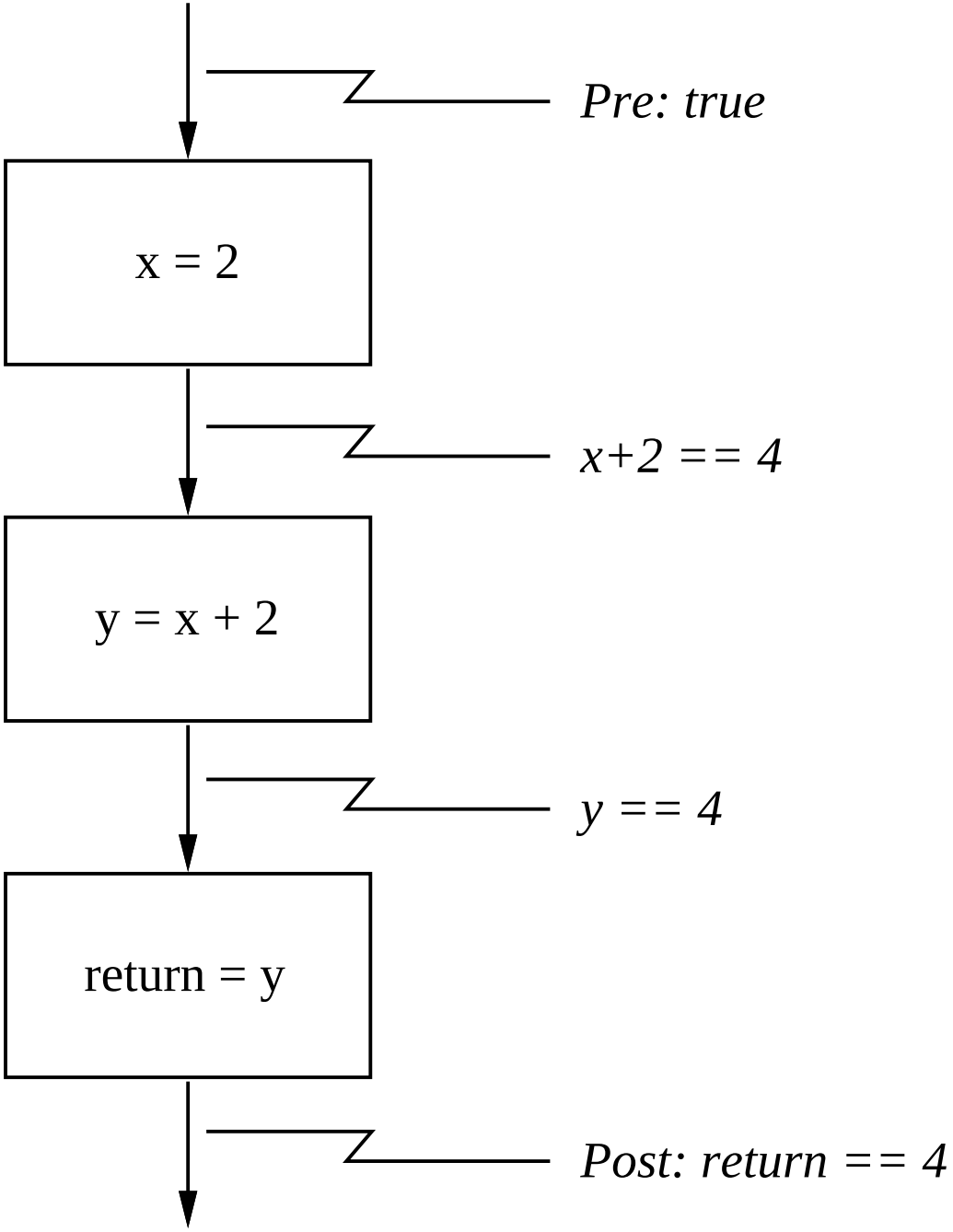
```
        int x,y;

        x = 2;
        y = x + 2;
        return y;

    }
```

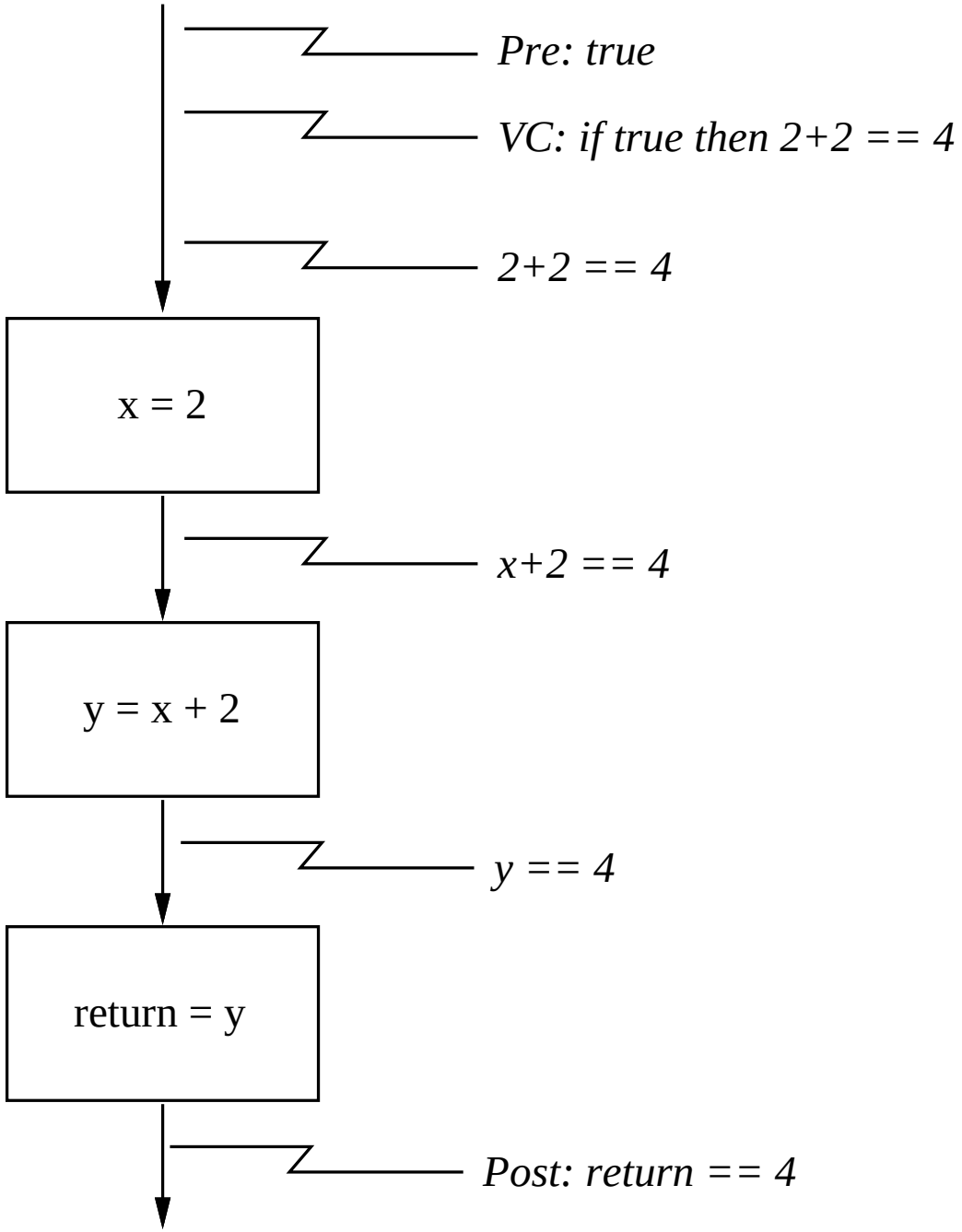
Stunning result, cont'd

B. Here are the steps of the proof:


Pre: true
x = 2
y = x + 2
return = y
Post: return == 4


Pre: true
x = 2
y = x + 2
y == 4
return = y
Post: return == 4


Pre: true
x = 2
x+2 == 4
y = x + 2
y == 4
return = y
Post: return == 4


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

XIII. A stunned result

A. Let's try to prove

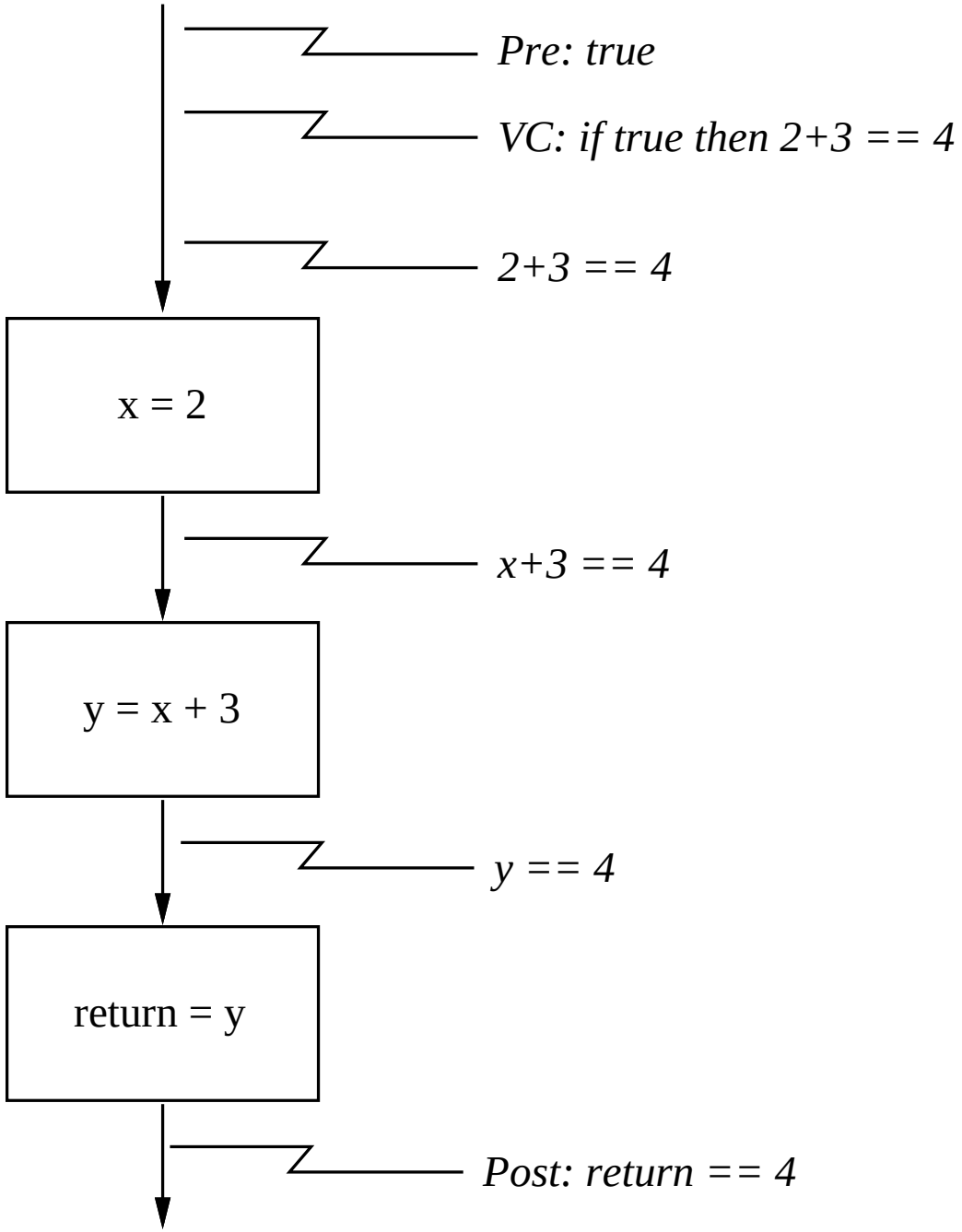
```
int ReallyDuh() {
/*
 * Add 2 to 3 and return
 * the result.
 *
 * pre: ;
 * post: return == 4;
 */
```

Stunned result, cont'd

```
        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 2+3 == 4
2+3 == 4
x = 2
x+3 == 4
y = x + 3
y == 4
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 VC nearest to statement in which error occurs.

XIV. **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 program verification, we assume p is true.

D. Hence, VC proof will fail if q is false.

XV. Proof of factorial example.

A. The (correct) definition:

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

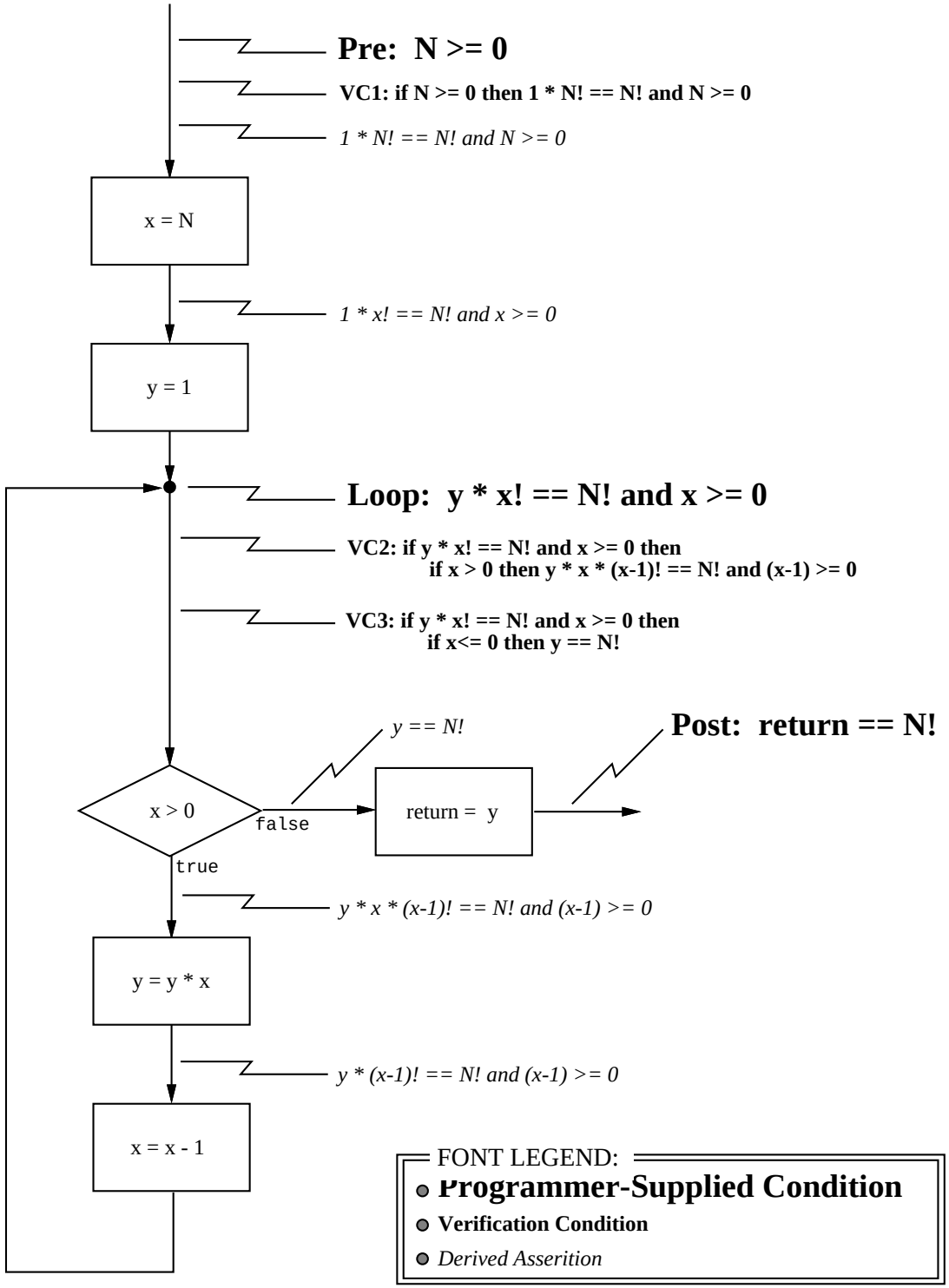
Proof of factorial, cont'd

```
    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. Slightly different than earlier version.

C. Figure 1 outlines the proof.


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

XVI. **Logical derivation of "y * x! = N!"**

A. At top of loops, what relationship should exist between loop variables?

B. Characterizes the *meaning* of the loop.

C. For fact, meaning is something like "y *approximates* N!".

Derivation of "y * x! = N!", cont'd

D. More precisely,

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

for some relation R; in this case, **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?

Derivation of "y * x! = N!", cont'd

It looks like y is growing by a multiplicative factor of x each time through,

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

F. I.e., y * x! = N!

Derivation of “y * x! = N!”, cont’d

G. This reasoning is typical for loop assertions.

H. An alternative is to use symbolic evaluation.

XVII. **Further tips on doing proofs**

A. Often, VC proofs not that difficult.

B. Use simple algebraic formula reduction.

C. 141 book has rules.

Further tips on doing proofs, cont'd

D. Some rules:

1. if t then P1 else P2 <=>
   (t ⊃ P1) and (not t ⊃ P2)

2. if t then t and P <=>
   if t then P

3. if t1 then if t2 then P <=>
   if t1 and t2 then P

4. t and (if t then P) => P *(modus ponens)*

XVIII. **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

Factorial VC's, cont'd

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

XIX. 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 $x > 0 \supset y * (x-1) * (x-1)! = N!$ =>
no go

Possible errors, cont'd

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 $x < 0 \supset y = N!$ =>

no go

XX. Automatic derivation of loop invariants

A. A mechanical technique

B. Looks like this:

Automatic loop invariants, cont'd

$$y = N!$$
$$\downarrow$$
$$y * x = N!$$
$$\downarrow$$
$$y * x * (x-1) = N!$$
$$\downarrow$$
$$y * x * (x-1) * (x-2) = N!$$
$$\downarrow$$
$$y * x * (x-1) * (x-2) * (x-3) = N!$$
$$\downarrow$$
$$\ldots$$

Automatic loop invariants, cont'd

...

↓

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

Automatic loop invariants, cont'd

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

D. This is the *loop invariant.*

E. Also interesting to look at erroneous case.

Automatic loop invariants, cont'd

$$y = N!$$
$$\downarrow$$
$$y * (x-1) = N!$$
$$\downarrow$$
$$y * (x-1) * (x-2) = N!$$
$$\downarrow$$
$$y * (x-1) * (x-2) * (x-3) = N!$$
$$\downarrow$$
$$\ldots$$

Automatic loop invariants, cont'd

...

↓

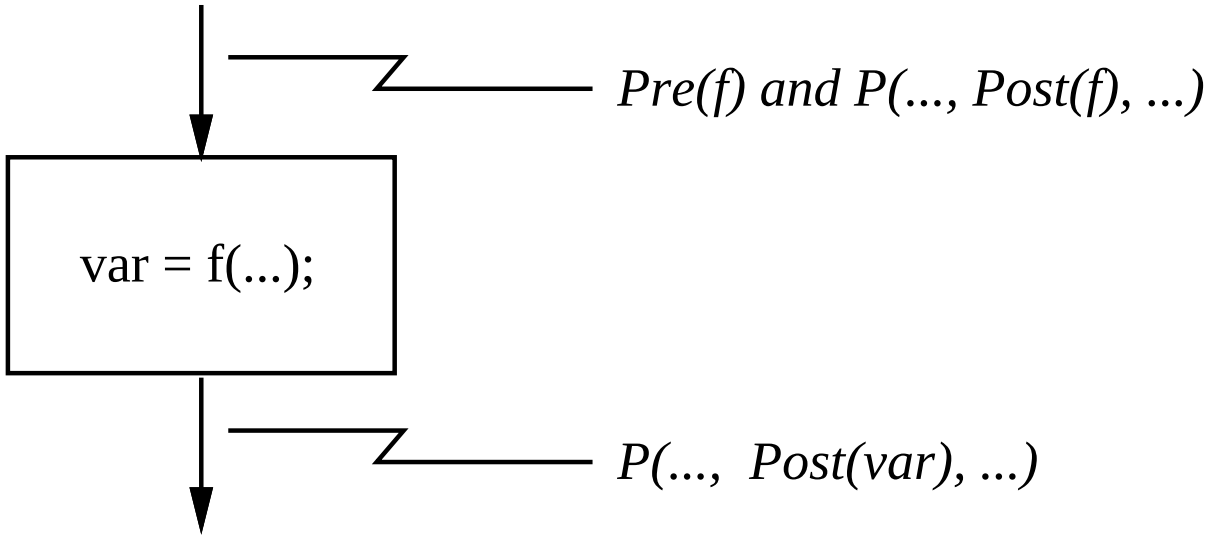
$$y * (x-1) * (x-2) * \ldots * (x-N) = N!$$

Automatic loop invariants, cont'd

F. In erroneous case, symbolic eval leads to wrong loop invariant.

G. Will ultimately cause verification to fail.

XXI. Verification rule for function calls



where *Post(var)* is postcond of function f in which *var* appears; *Post(f)* is postcond of f with appropriate variable substitution.

Rule for function calls, cont'd

A. Substituting function precond for postcond.

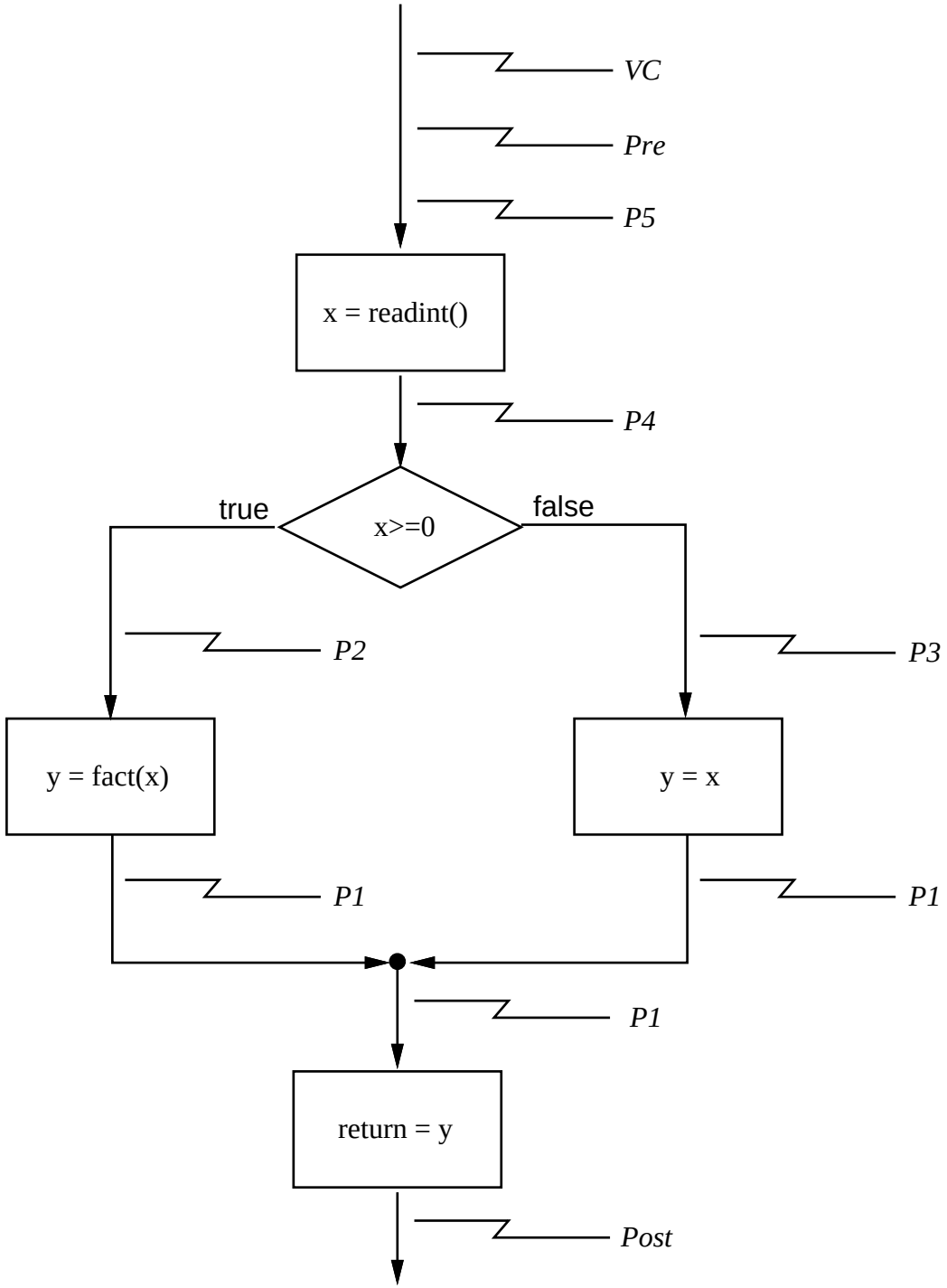
B. Recall two methods to ensure precond is met:

   1. Exceptions thrown by function.

   2. Verify function will never be called if is precond is false.

C. We're now in a position to do the latter.

XXII. Verify that factorial is never called with false precond.


VC
Pre
P5
x = readint()
P4
true
x>=0
false
P2
P3
y = fact(x)
y = x
P1
P1
P1
return = y
Post

Details of the proof:

Label **Predicate**

```
VC:     true => forall (x: integer)
             if (x>=0) then x!==x! else x==x
        =>
        true

Pre:    true

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

Details of proof, cont'd:

```
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

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

Details of proof, cont'd:

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

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

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

XXIII. **Partial versus total correctness**

A. Preceding methodology demonstrates *partial* correctness.

B. Program is correct, *if and only it terminates*.

C. *Total* correctness requires additional proof of termination.

D. Generally involves an induction.

XXIV. **Verif'n and programming style**

A. Certain stylistic rules must be obeyed.

B. A summary:

1. Functions cannot have side effects.
2. Input parameters cannot be modified.
3. Restricted control flow constructs.

XXV. Some critical questions

A. Can it scale up?

B. Why hasn’t it caught on yet?

C. When will it catch on?

Critical questions, cont'd

1. when software engineers receive adequate training in formal methods

2. when production-quality tools become available

3. when software users get sufficiently sick of crappy products

Critical questions, cont'd

D. Verification tools include:

1. formal spec languages
2. automatic invariant generators
3. automatic theorem provers

E. Tools used by researchers and a few commercial developers.

XXVI. **Optimistic conclusion --** it *will* happen,

when some or all of above conditions are met.