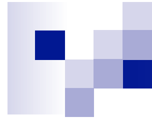
(Imperative) Program Logic

Robert Keller

February 2013



Proofs for Programs

- For many reasons, it is desirable to accompany programs with a **proof** that the program meets a certain specification.

- One way to do this is to **derive the proof along with deriving the program**.

Related text material

- Huth & Ryan
  Chapter 4, Program verification

- Note: Their “tableau proofs” should not be confused with the tableaux we have discussed so far.

- Also, they use funny braces that are a combination of parens and |: (| and |), where I and others just use { }.

Alan Turing

- Turing may have been the first to consider proving that a program is correct, in his 1949 paper (3 typewritten pages):

  "Checking a Large Routine"

  "How can one check a large routine in the sense that it's right?

  … make a number of definite <u>assertions</u> which can be checked individually, and from which the correctness of the whole program easily follows."

A corrected version, with comments, was published by
F.L. Morris and C.B. Jones in Annals of the History of Computing,
(Vol. 6, Apr. 1984)

http://www.turingarchive.org/viewer/?id=462&title=01

Robert W. Floyd

"Assigning meanings to programs", 1967

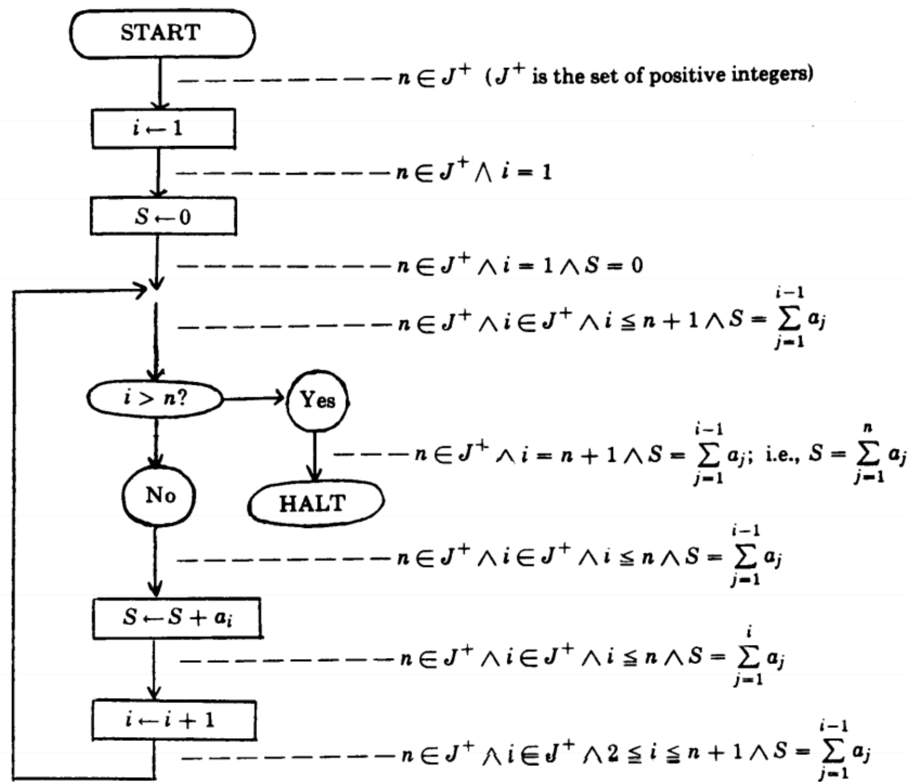
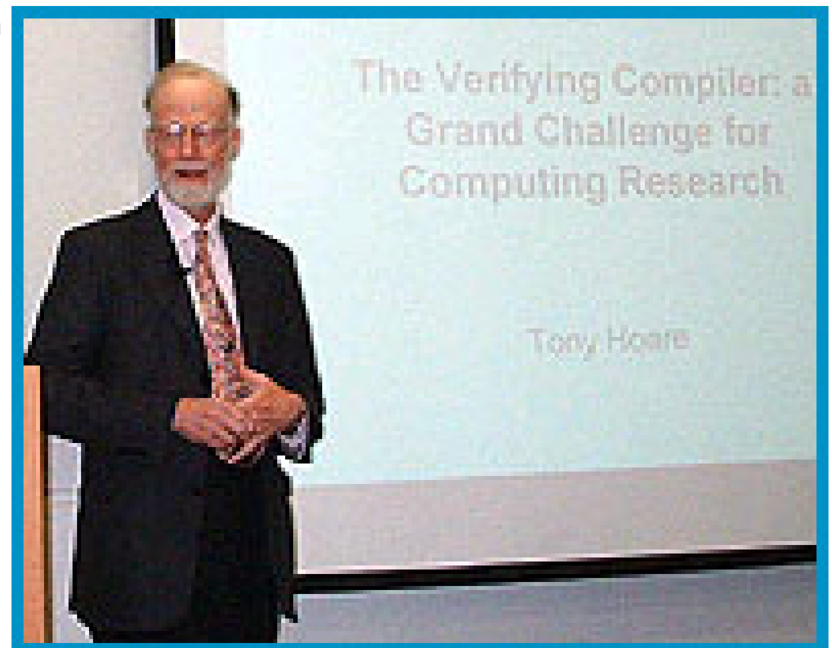


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geqq 0)$

Hoare Logic

- C.A.R. ("Tony") Hoare was the first to express program construction along with proofs of correctness as a single **unified logic**.
- **"An axiomatic basis for programming", 1969.**



Sir Tony Hoare (FRS)
Microsoft Research Laboratory, Cambridge

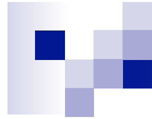
One of the Rules from Hoare's Paper

D2 Rule of Composition

If $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{(Q_1 ; Q_2)\}R$

- You may be accustomed to thinking of a program as something with “dynamic” behavior.

- A ***mathematical*** view is that a program’s behavior is just one of many paths through of a (generally-infinite) ***static*** structure, which can be analyzed with mathematics.



Programs States

- Programs work with **states**.
- Each state is a mapping from program variables into appropriate domains
- A state is much like an ***assignment*** for an interpretation our discussion of semantics of predicate calculus.

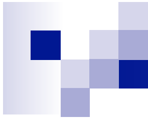
Example: Integer Square Root Program

```
s = 1;
i = 1;
r = 0;
while( s ≤ n )
    {
    r = r + 1;
    i = i + 2;
    s = s + i;
    }
```

variables

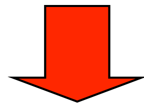
rows are states

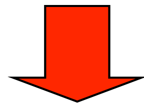
s	i	r	n
			10
1			10
1	1		10
1	1	0	10
1	1	1	10
1	3	1	10
4	3	1	10
4	3	2	10
4	5	2	10
9	5	2	10
9	5	3	10
9	7	3	10
16	7	3	10
16	7	3	10



Program as a State Transformer

starting state

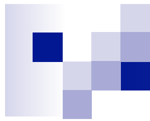




ending state

- To deal with input streams and files, we will **consider the entire file or stream**, along with the current position of the reader or writer, to be part of the state.

- We won't be dealing with such issues in this presentation.



Programs with added Assertions

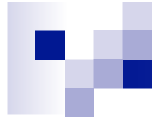
- An **assertion** is a predicate-logic expression about the variables in the program.

- Assertions can express two kinds of things:

  - An **assumption** about the state before a box (also called the **pre-condition**).

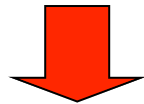
  - An **expectation** about the state after a box (also called the **post-condition**).

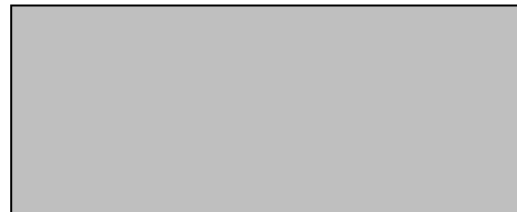
  - Sometimes, e.g. Huth & Ryan, expectations are called “guarantees”.



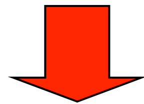
A **Program Specification** consists of

(i) **assumption** about the starting state





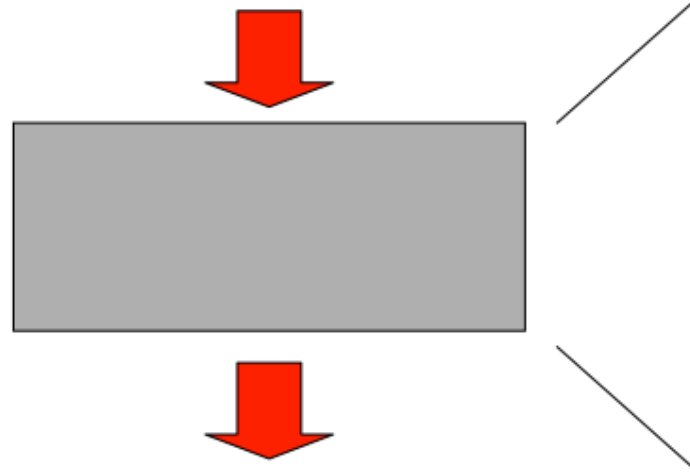
Program as
a gray box.



(ii) **expectation** about the ending state

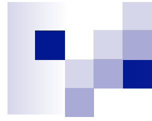
Example: For Integer Square Root

(i) **assumption:** $n \geq 0$



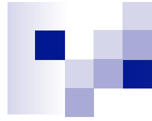
```
s = 1;
i = 1;
r = 0;
while( s ≤ n )
      {
      r = r + 1;
      i = i + 2;
      s = s + i;
      }
```

(ii) **expectation:** $r = isqrt(n)$



Relativity

- Expectations are usually *relative* to assumptions.
- Nothing in particular can be expected if the assumption is false when the program is started.



Relating Expectation to Assumption

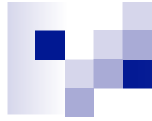
- Variables common to both expectation and assumption can be used to relate the two.
- Without such relations, the task of developing and proving a program can become trivial or meaningless.

- Assumption:

  int x[0..n-1] is an array of size n

- Expectation (x is sorted):

  $\forall i \, ((i \geq 1 \wedge i < n) \rightarrow (x[i-1] \leq x[i]))$



A trivial way to meet the specification

```
for( i = 0; i < n; i++ )
   {
    x[i] = 0;
   }
```

A More Exacting Specification

(introduces a new array x_0 not part of the program)

- Assumption:

  int x[0..n-1] is an array of size n
  $\wedge$ **$\mathbf{x = x_0}$**
  (in the sense that are two arrays with the same elements)

- Expectation:

  $\forall i \; ((i \geq 1 \wedge i < n) \rightarrow (x[i-1] \leq x[i]))$
  $\wedge$ **bagof(x, $\mathbf{x_0}$)**
  (meaning x has the same elements of the same multiplicity as x_0)

Application in Software Engineering

- “Design by Contract” (vs. “Defensive Programming”)
- Design a program module as if the assumption were true at the start.
- Design the module to meet the expectation.
- Do not build in extra checks for wrong data. (This helps reduce redundancy in the system overall.)
- Of course, at the **external** interface, the program should check for “wrong data”, but this too could be part of the specification.

http://en.wikipedia.org/wiki/Defensive_programming

http://en.wikipedia.org/wiki/Design_by_contract

Specification with *Exceptions*

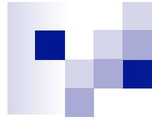
- **Assumption**:

$$\text{valid(input)} \rightarrow T$$
$$\wedge\ \neg\text{valid(input)} \rightarrow \text{red_flag}$$

- **Expectation**:

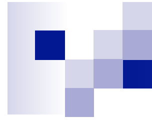
$\neg$red_flag $\rightarrow$ … normal expectation …
$\wedge$ red_flag $\rightarrow$ **exception** is indicated

- The value of valid(input) may be something the program itself computes.
But it is a predicate just the same.



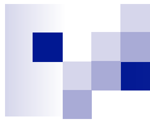
Ways of Using Logic

- **Program Synthesis**: Automate the construction of a program from a specification.
- **Formal Verification**: Create a program that is **proved** to meet its specification.
- **Model Checking**: Mechanically check that a program meets its specification (used for finite-state systems).
- **Static Analysis**: Symbolically check that no erroneous things are being done by the program (incomplete, but useful).



What ifs

- What if the assumption about the starting state doesn't hold?
  - We don't care about the result in this case.
  - However, the assumption can be made very stringent, e.g. T, in which case we will always care.

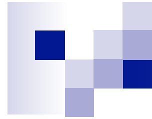


What ifs

- What if the assumption about the starting state holds, **but**

  the expectation doesn’t hold when the program terminates?

  - The program is incorrect.



Floyd Assertions

- Annotate program with logical assertions between statements.
- Prove that assertions hold, based on a form of induction.

Floyd Assertions for Integer Square Root

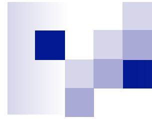
```
s = 0;          ----- n≥0
i = 1;          ----- n≥0 ∧ s=0
r = 0;          ----- n≥0 ∧ s=0 ∧ i=1
                ----- n≥0 ∧ s=r² ∧ i=2r+1
while( s < n )
    {
                ----- s<n ∧ n≥0 ∧ s=r² ∧ i=2r+1
    s = s + i;
    i = i + 2;  ----- (to be completed)
    r = r + 1;
    }           ----- r²≤n<(r+1)²
```

----- $s<n \wedge n>0 \wedge s=r2 \wedge i=2r+1$	assertion
s = s + i;	program statement

From program + assertions are derived "verification conditions", which are pure logical statements that can be proved independently of each other.

verification condition:

$(s<n \wedge n \geq 0 \wedge s=r^2 \wedge i=2r+1)$	pre-condition
$\wedge\ (s' = s + i)$	statement semantics
$\rightarrow (s'<n+i \wedge n \geq 0 \wedge s'=r^2+i \wedge i=2r+1)$	post-condition



Better Mechanization

Hoare, and later Dijkstra, demonstrated how the derivation of assertions could be partly automated,

eliminating the need to create verification conditions explicitly.

In particular, Hoare's method resembled natural deduction.

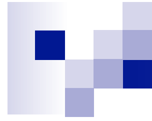
Hoare Triples are Formalized VCs

- Consider endowing a program to be designed with its assumption and expectation:

  {assumption} code {expectation}

- This is known as a “triple”, or “Hoare triple”.

- Originally Hoare put the braces around the code, instead of around the assertions. Now the opposite is more common.



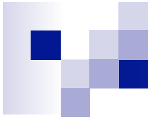
Example of a Triple

{assumption} code {expectation}

$\{x \leq y \wedge x \leq z\}$ *...TBD...* $\{x \leq y \wedge y \leq z\}$

Design then becomes the process of filling in the TBD code.

[TBD = “To be Determined”]



Some triples are more stringent than others

{assumption} code {expectation}

$\{x \leq y \wedge x \leq z\}$ *TBD* $\{x \leq y \wedge y \leq z\}$

$\{x \leq y\}$ *TBD* $\{x \leq y \wedge y \leq z\}$

$\{T\}$ *TBD* $\{x \leq y \wedge y \leq z\}$

$\{T\}$ *TBD* $\{x \leq y \wedge y \leq z \wedge z \leq w\}$

Stringency (can skip on first pass)

- {assumption} code {expectation}
- T_1:{A} c {E} for short
- T_2:{A} c {E'}
- T_3:{A'} c {E}

- If E' → E, is T_2 more or less stringent than T_1?

- If A' → A, is T_3 more or less stringent than T_1?

Rationale

- If E' → E, then any state satisfying E' must also satisfy E, but not necessarily conversely, so {A} c {E'} is more stringent than {A} c {E}.
- If A' → A, then any state satisfying A' must also satisfy A, so {A'} c {E} is less stringent than {A} c {E}.
- In other words,
    - {A} c {E'} meets the expectation and **possibly more**.
    - {A'} c {E} **assumes more** to get the same job done.

Maximal Stringency

- {T} c {⊥}

- Not assuming anything, but meeting every expectation.

- Not seen too often, as this implies the program will never terminate under any condition.

- $\{\bot\}\ c\ \{\top\}$

- Assuming everything, not expecting anything.

- Not seen too often either, as there is no point in running this program. The assumptions cannot be satisfied.

First Rule of Inference

- Consequent Rule

$$\frac{A \rightarrow A',\ \ \{A'\}\ c\ \{E'\},\ \ E' \rightarrow E}{\{A\}\ c\ \{E\}} \qquad \text{consequent}$$

- Here triples meet logic ($\rightarrow$ is implies).
- Effectively this says that any triple can be derived from a more stringent one.
- This is called the “Implied” rule in Huth&Ryan, p 270.

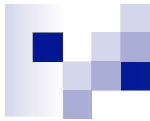
Special Cases

- JAPE's consequent(L):

$$\frac{A \rightarrow A', \quad \{A'\}\ c\ \{E\}}{\{A\}\ c\ \{E\}} \qquad \text{consequent(L)}$$

- JAPE's consequent(R):

$$\frac{\{A'\}\ c\ \{E'\}, \ E' \rightarrow E}{\{A\}\ c\ \{E\}} \qquad \text{consequent(R)}$$



Composition of Triples

- Suppose we have a triple:
  {Assumption} Code {Expectation}

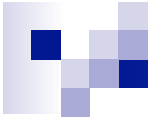
- To develop the code, we can decompose into two parts:

  {Assumption 1} Code 1 {Expectation 1}

  {Assumption 2} Code 2 {Expectation 2}

  We want Code = Code 1; Code 2 (concatenation)

  What do we need for this to work?



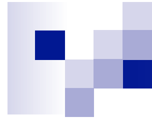
Composition Rule

We need Expectation1 = Assumption2.

In the form of a natural deduction rule:

$$\frac{\{A\}\ S1\ \{B\} \qquad\qquad \{B\}\ S2\ \{C\}}{\{A\}\ S1;S2\ \{C\}}\ \text{composition}$$

1. $\{T\}\ S1\ \{x \leq y\}$
2. $\{x \leq y\}\ S2\ \{x \leq y \wedge y \leq z\}$
3. $\{T\}\ S1;\ S2\ \{x \leq y \wedge y \leq z\}$ Comp. 1, 2



What if Conditions don't Match

- Sometimes we need to compose segments of code, but the expectation of the first doesn't match the assumption of the second.
- In this case, we seek help from the consequent rule, together with composition.

Example of Weakening/Strengthening

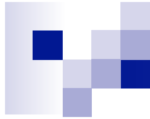
1. {T} *S1* $\{x < y\}$

2. $\{x \leq y\}$ *S2* $\{x \leq y \wedge y \leq z\}$

3. To compose these we can either use consequent, then composition to get:

   {T} *S1;S2* $\{x \leq y \wedge y \leq z\}$

   since $x < y \rightarrow x \leq y$



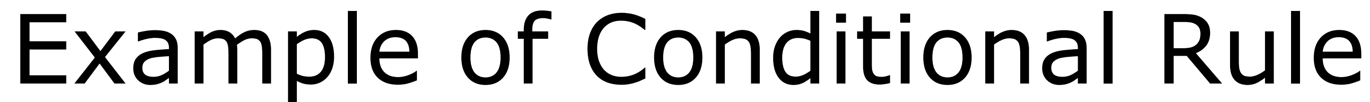
Generalized Composition Rule

$$\frac{\{A\}\ S1\ \{B\} \qquad B \rightarrow C \qquad \{C\}\ S2\ \{D\}}{\{A\}\ S1;S2\ \{D\}}\ \text{compose}$$

$$\frac{\{A \wedge P\}\ S1\ \{B\} \qquad \{A \wedge \neg P\}\ S2\ \{B\}}{\{A\}\ \ \textbf{if(}\ P\ \textbf{)}\ S1\ \textbf{else}\ S2\ \ \{B\}}\ \text{cond}$$

There is a strong resemblance to ∨**−Elimination** in natural deduction, with the ∨-formula being $P \vee \neg P$.

Called “If” in H&R, “choice” in JAPE.



Example of Conditional Rule

1. $\{x \leq y \wedge (y > z)\}$ *S1* $\{x \leq y \wedge y \leq z\}$

2. $\{x \leq y \wedge \neg(y > z)\}$ *S2* $\{x \leq y \wedge y \leq z\}$

same expectations

3. $\{x \leq y\}$

   if(y > z) S1 **else** S2

   $\{x \leq y \wedge y \leq z\}$ cond 1, 2

One-Sided Conditional Rule

$$\frac{\{A \wedge P\}\ S1\ \{B\} \qquad (A \wedge \neg P) \rightarrow B}{\{A\}\ \textbf{if(}\ P\ \textbf{)}\ S1\ \{B\}}\ \text{cond-1}$$

Example of One-Sided Conditional Rule

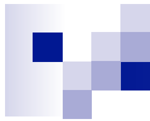
1. $\{x \leq y \wedge y > z\}\ S1\ \{x \leq y \wedge y \leq z\}$

2. $((x \leq y) \wedge \neg(y > z)) \rightarrow (x \leq y \wedge y \leq z)$

3. $\{x \leq y\}$

   if(y > z) S1

   $\{x \leq y \wedge y \leq z\}$ cond-1, 1, 2



While Rule

$$\frac{\{I \wedge P\}\ S\ \{I\}}{\{I\}\ \textbf{while(P) S}\ \{I \wedge \neg P\}}\ \text{while}$$

I is known as the “loop invariant”

S is the “body”, P is the “test”

Example of While Rule

1. $\{x \leq y \wedge y \geq z\}\ S\ \{x \leq y\}$

2. $\{x \leq y\}$

   while($y \geq z$) S

   $\{x \leq y \wedge \neg(y \geq z)\}$ while, 1

Assignment Statement Rule

$$\frac{}{\{A[\varepsilon/\nu]\} \quad \nu := \varepsilon \quad \{A\}} \text{ assign}$$

ν is a variable, an ε expression.

As in predicate logic, $A[\varepsilon/\nu]$ denotes the result of replacing **free** occurrences of variable ν in A with ε.

(This rule has an **empty** antecedent, i.e. an “axiom”.)

“Assignment” here should not be confused with assignment as in the interpretation of logic formulas.

Those assignments are like program states.

Example of Assignment Rule

$\{x \leq z\}$ **y := z** $\{x \leq y\}$ assign

Here ν is identified with y

ε is identified with z

$\{A[\varepsilon/\nu]\}$ $\nu := \varepsilon$ $\{A\}$

It is easiest to “work backward” from the expectation.

More Examples of Assignment Rule

$\{A[\varepsilon/\nu]\} \quad \nu := \varepsilon \; \{A\}$

1. $\{x \leq y+1\}$ **y := y+1** $\{x \leq y\}$ — assign
2. $\{x*y \leq n\}$ **y := x*y** $\{y \leq n\}$ — assign
3. $\{x+1 \leq n+1\}$ **x := x+1** $\{x \leq n+1\}$ — assign

Using Conditional Rule with Derived Weakest Pre-Conditions

The basic conditional rule has been stated:

$$\frac{\{A \wedge P\}\ S1\ \{B\} \quad \{A \wedge \neg P\}\ S2\ \{B\}}{\{A\}\ \textbf{if(}\ P\ \textbf{)}\ S1\ \textbf{else}\ S2\ \{B\}}\ \text{cond}$$

Let’s suppose that pre-conditions C1 and C2 for S1 and S2 have been derived from B, e.g. by using the assignment rule:

{C1} S1 {B} {C2} S2 {B}.

The weakest pre-condition A that follows is then:

$(P \rightarrow C1) \wedge (\neg P \rightarrow C2)$

The equivalent in a programming language assertion might be P ? C1 : C2 (Java, C++), or C1 if P else C2 (Python).

Consequent Rule is Used a Lot in What Follows
This is a reminder:

- To derive:

  $\{P\}\ S\ \{Q\}$

- it **suffices** to derive:

  $\{P'\}\ S\ \{Q'\}$

- provided we can derive the logical statements

  $P \rightarrow P'$ and $Q' \rightarrow Q$

(i.e. derive a triple with a weaker assumption and stronger expectation)

Examples of Derivations of Small Programs: **Exchange Program**

To derive: A program that exchanges the values in variables x and y.

$\{x = x_0 \wedge y = y_0\}$ **z := x; x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$

This can be done by applying the **assignment rule** working from the final assertion backward to derive the intermediate assertions.

The resulting triples are composed using the **composition rule**.

Examples of Derivations of Small Programs: **Exchange Program Step 1 (working backward)**

To derive: A program that exchanges the values in variables x and y.

$\{x = x_0 \wedge y = y_0\}$ **z := x; x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$

1.
2.
3. $\{z = x_0 \wedge x = y_0\}$ **y := z;** $\{y = x_0 \wedge x = y_0\}$ assign

Examples of Derivations of Small Programs:
Exchange Program Step 2 (working backward)

To derive: A program that exchanges the values in variables x and y.

$\{x = x_0 \wedge y = y_0\}$ **z := x; x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$

1.
2. $\{z = x_0 \wedge y = y_0\}$ **x := y;** $\{z = x_0 \wedge x = y_0\}$ assign
3. $\{z = x_0 \wedge x = y_0\}$ **y := z;** $\{y = x_0 \wedge x = y_0\}$ assign

Examples of Derivations of Small Programs:
Exchange Program Step 3 (working backward)

To derive: A program that exchanges the values in variables x and y.

$\{x = x_0 \wedge y = y_0\}$ **z := x; x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$

1. $\{x = x_0 \wedge y = y_0\}$ **z := x;** $\{z = x_0 \wedge y = y_0\}$ assign
2. $\{z = x_0 \wedge y = y_0\}$ **x := y;** $\{z = x_0 \wedge x = y_0\}$ assign
3. $\{z = x_0 \wedge x = y_0\}$ **y := z;** $\{y = x_0 \wedge x = y_0\}$ assign

Examples of Derivations of Small Programs: **Exchange Program Step 4 (composing)**

To derive: A program that exchanges the values in variables x and y.

$\{x = x_0 \wedge y = y_0\}$ **z := x; x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$

1. $\{x = x_0 \wedge y = y_0\}$ **z := x;** $\{z = x_0 \wedge y = y_0\}$ assign
2. $\{z = x_0 \wedge y = y_0\}$ **x := y;** $\{z = x_0 \wedge x = y_0\}$ assign
3. $\{z = x_0 \wedge x = y_0\}$ **y := z;** $\{y = x_0 \wedge x = y_0\}$ assign
4. $\{z = x_0 \wedge y = y_0\}$ **x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$ compose 2,3

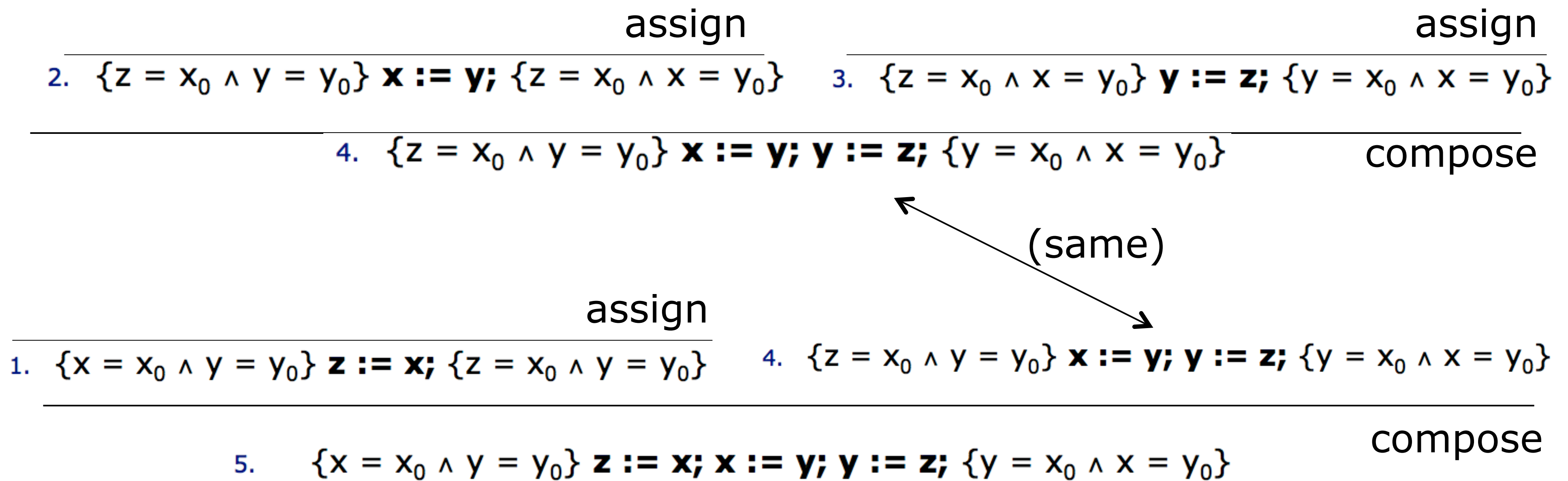
Examples of Derivations of Small Programs: **Exchange Program Step 5 (composing)**

To derive: A program that exchanges the values in variables x and y.

$\{x = x_0 \wedge y = y_0\}$ **z := x; x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$

1. $\{x = x_0 \wedge y = y_0\}$ **z := x;** $\{z = x_0 \wedge y = y_0\}$ assign
2. $\{z = x_0 \wedge y = y_0\}$ **x := y;** $\{z = x_0 \wedge x = y_0\}$ assign
3. $\{z = x_0 \wedge x = y_0\}$ **y := z;** $\{y = x_0 \wedge x = y_0\}$ assign
4. $\{z = x_0 \wedge y = y_0\}$ **x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$ compose 2,3
5. $\{x = x_0 \wedge y = y_0\}$ **z := x; x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$ compose 1, 4

Examples of Derivations of Small Programs:
Exchange Program: Derivation as Tree



Examples of Derivations of Small Programs: Ordering two numbers

$\{x = x_0 \wedge y = y_0\}$

if(x > y) {z := x; x := y; y := z;}

$\{x \leq y \wedge ((x = x_0 \wedge y = y_0) \vee (y = x_0 \wedge x = y_0))\}$

One-Sided Conditional Rule

$$\frac{\{A \wedge P\}\ S\ \{B\} \qquad (A \wedge \neg P) \rightarrow B}{\{A\}\ \textbf{if(}\ P\ \textbf{)}\ S\ \{B\}}$$

cond-1

Pattern Matching:

$\{x = x_0 \wedge y = y_0\}$ A

if(x > y) {z := x; x := y; y := z;} P S B

$\{x \leq y \wedge ((x = x_0 \wedge y = y_0) \vee (y = x_0 \wedge x = y_0))\}$

P is **x > y**

B is $x \leq y \wedge ((x = x_0 \wedge y = y_0) \vee (y = x_0 \wedge x = y_0))$

A is $(x = x_0 \wedge y = y_0)$

S is **z := x; x := y; y := z;**

So we need to derive

- $\{A \wedge P\}\ S\ \{B\}$ which is

$\{(x = x_0 \wedge y = y_0) \wedge x > y\}$

z := x; x := y; y := z;

$\{x \leq y \wedge ((x = x_0 \wedge y = y_0) \vee (y = x_0 \wedge x = y_0))\}$

- $(A \wedge \neg P) \rightarrow B$ which is

$((x = x_0 \wedge y = y_0) \wedge \neg(x > y))$

$\rightarrow \{x \leq y \wedge ((x = x_0 \wedge y = y_0) \vee (y = x_0 \wedge x = y_0))\}$

This part is pure logic.

Derivation of the Triple

Similar to the derivation in the previous example,

$\{x = x_0 \wedge y = y_0\}$ **z := x; x := y; y := z;** $\{y = x_0 \wedge x = y_0\}$

we can derive:

$\{y \leq x \wedge x = x_0 \wedge y = y_0\}$
z := x; x := y; y := z;
$\{x \leq y \wedge y = x_0 \wedge x = y_0\}$

Then use the consequent rule twice to get the triple
$\{(x = x_0 \wedge y = y_0) \wedge x > y\}$
z := x; x := y; y := z;
$\{x \leq y \wedge ((x = x_0 \wedge y = y_0) \vee (y = x_0 \wedge x = y_0))\}$

using $x > y \rightarrow y \leq x$,
and $(y = x_0 \wedge x = y_0) \rightarrow ((x = x_0 \wedge y = y_0) \vee (y = x_0 \wedge x = y_0))$

- integer x, n

  $\{x \leq n\}$ **while(x < n) x := x+1** $\{x = n\}$

- We can use the while rule, provided that we can rely on properties of **integer** arithmetic such as:

$$(x < n) \rightarrow ((x+1) \leq n)$$

$$((x \leq n) \wedge \neg(x < n)) \equiv (x = n)$$

Examples of Derivations of Small Programs

1.	$((x \leq n) \wedge \neg(x < n)) \equiv (x = n)$	Premise
2.	$(x < n) \rightarrow ((x+1) \leq n)$	Premise
3.	$\{x+1 \leq n\}$ **x := x+1** $\{x \leq n\}$	Assignment
4.	$\{x < n\}$ **x := x+1** $\{x \leq n\}$	Consequent 3, 2
5.	$\{x \leq n \wedge x < n\}$ **x := x+1** $\{x \leq n\}$	Consequent 4
6.	$\{x \leq n\}$ while(x < n) **x := x+1** $\{x \leq n \wedge \neg(x < n)\}$	While 4
7.	$\{x \leq n\}$ while(x < n) **x := x+1**$\{x = n\}$	Consequent 6

Verification Conditions Viewpoint

- An alternate, less formal, way to view a triple, such as:

  $\{x+1 \leq n\}$ **x := x+1** $\{x \leq n\}$

- Think of the assignment in terms of primed (after) and unprimed values:

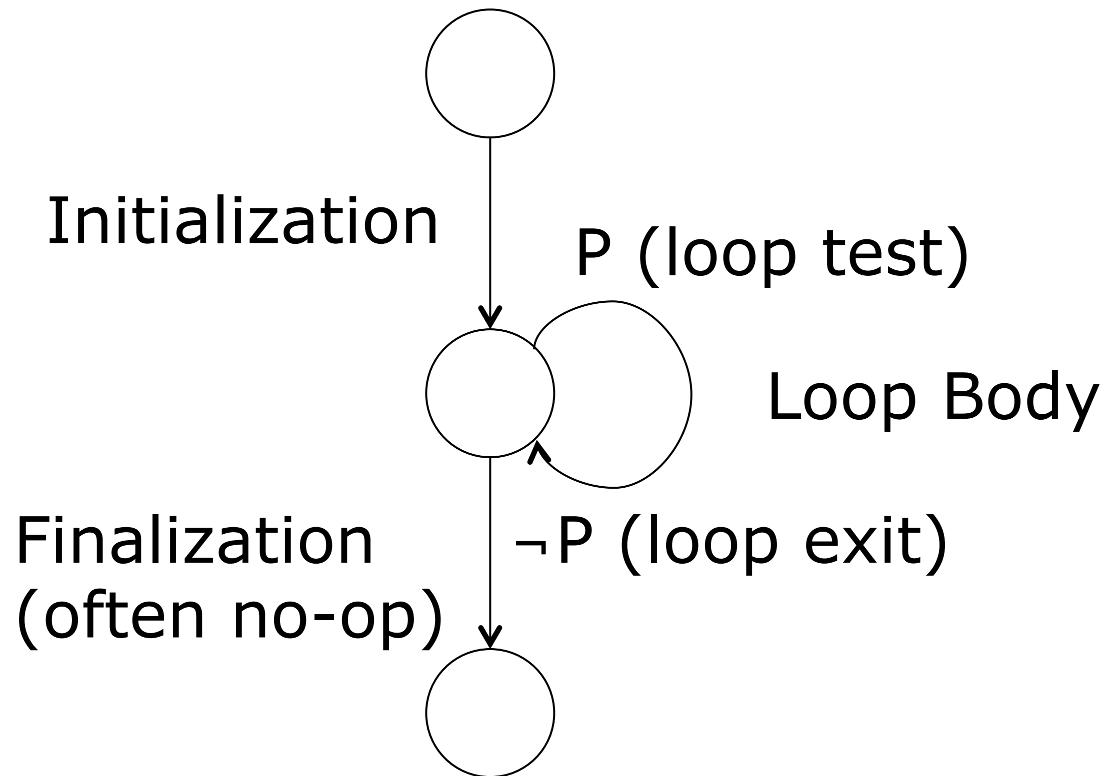
  $x' = x + 1$ (mathematical equality)

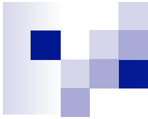
  Then what we are proving is the following **verification condition**:

  $(x+1 \leq n \wedge (x' = x + 1)) \rightarrow (x' \leq n)$

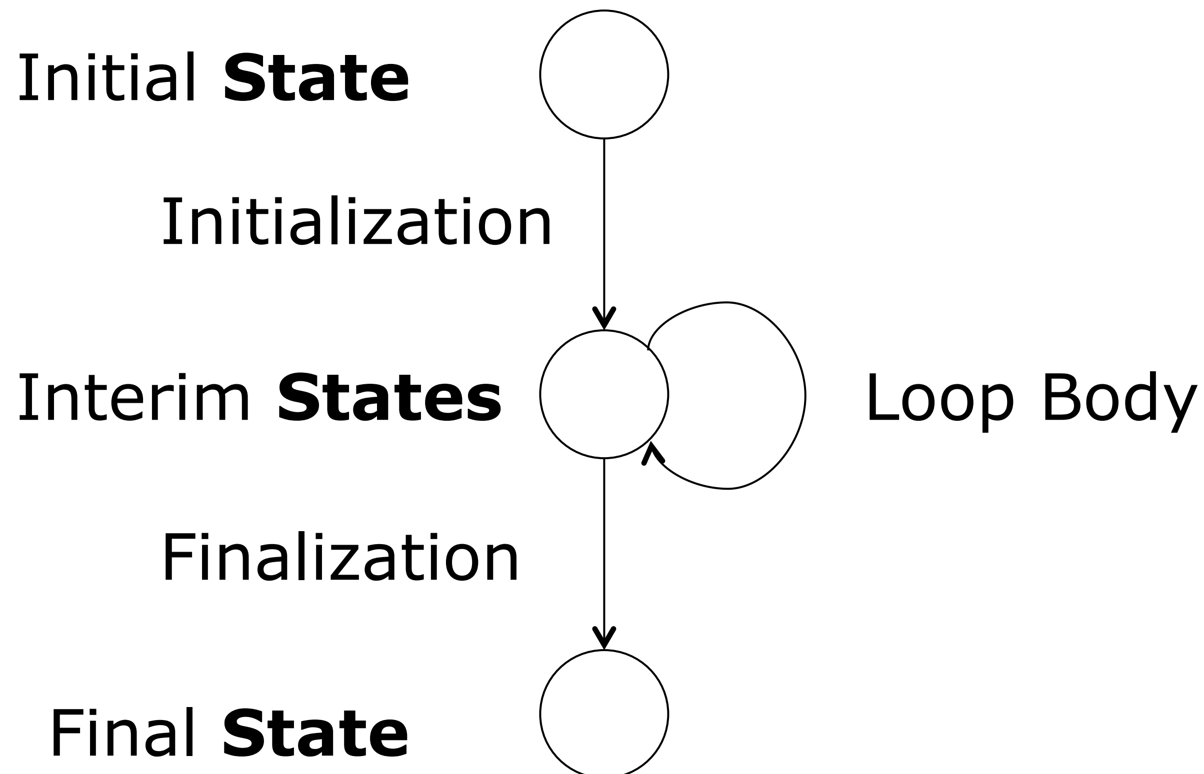
  Proving the program reduces to proving a set of verification conditions, one for each transition in the program.
  Once the VC's are constructed, the program can be forgotten.
  This was **Floyd's method**.

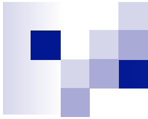
General Structure of a *while* Loop





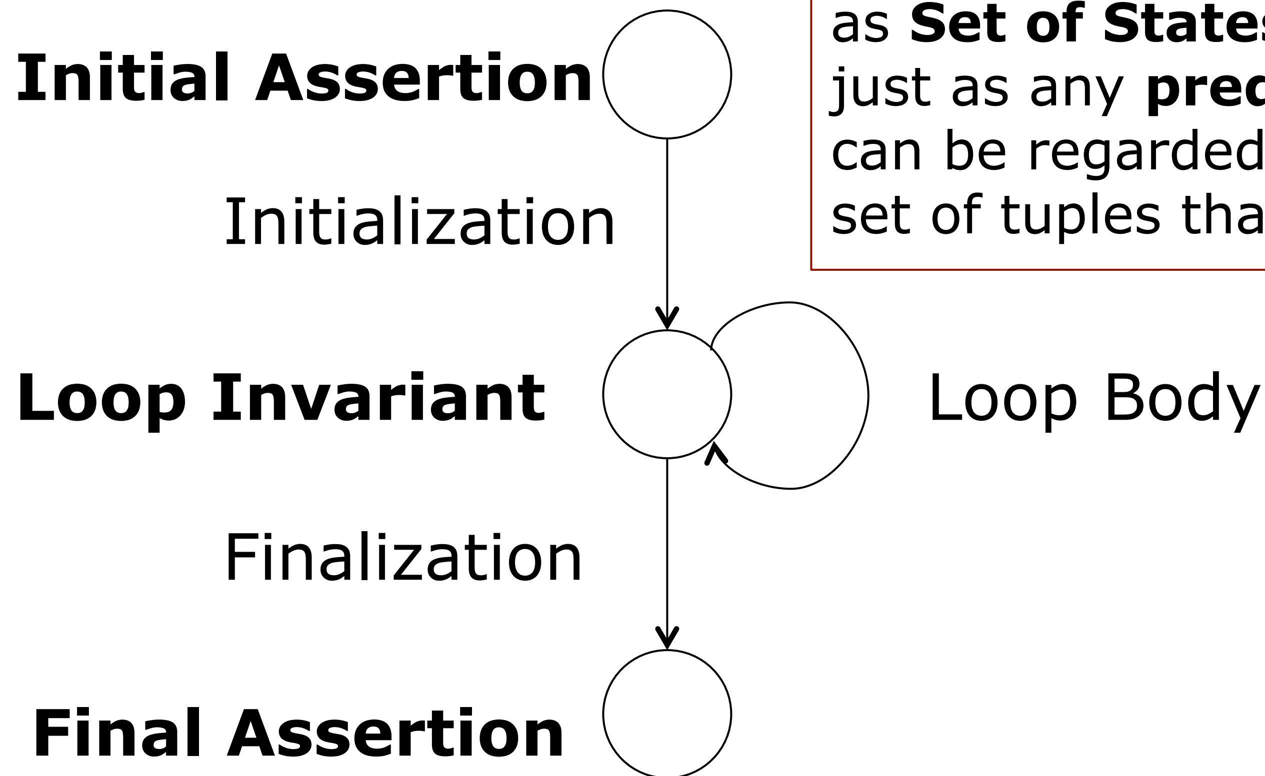
Structure of a ***while*** Loop





Assertions for a ***while*** Loop

Can regard **Assertion**
as **Set of States**,
just as any **predicate**
can be regarded as the
set of tuples that satisfy it.



Proving a General While Loop

- Three separate proofs are required in general:
  - {Initial} Initialization {Invariant}
  - {Invariant ∧ Test} Body {Invariant}
  - {Invariant ∧ ¬Test} Finalization {Final}
- In braces are logical **assertions**.
- Not in braces is **code**.
- The **Invariant** has to be devised to make this work.

Example

- $\{x \leq n\}$ **while(x < n) x := x+1** $\{x = n\}$

- A suggested loop invariant is:

  $x \leq n$

- The triples to be proved are then:

  $\{x \leq n\}$ **no-op** $\{x \leq n\}$

  $\{x \leq n\}$ **while(x < n) x := x+1** $\{x = n\}$

  $\{x = n\}$ **no-op** $\{x = n\}$

Example continued

- The main triple left **to be proved** is then:

  $\{x \leq n\}$ **while(x < n) x := x+1** $\{x = n\}$ **(A)**

- Using the **while** rule we can expect to get:

  $\{x \leq n\}$ **while(x < n) x := x+1** $\{x \leq n \wedge \neg(x < n)\}$ **(B)**

- We can get A from B by using the consequent rule, if we can just show:

  $(x \leq n \wedge \neg(x < n)) \rightarrow x = n$

- This is a logical property of integer arithmetic.

Example continued

- $\{x \leq n\}$ **while(x < n) x := x+1** $\{x \leq n \wedge \neg(x < n)\}$ **(B)**
  follows from the **while** rule, provided we can derive
- $\{x \leq n \wedge x < n\}$ x := x+1 $\{x \leq n\}$ **(C)**

- Using the **assignment** rule, we can derive
  $\{x+1 \leq n\}$ x := x+1 $\{x \leq n\}$ **(D)**

- So we are done, by using the **consequent** rule, if we can just show:
  $(x \leq n \wedge x < n) \rightarrow x+1 \leq n$

- This is also a logical property of integer arithmetic.

Summary of the Example

$\{x \leq n\}$

while(x < n)

  $\{x \leq n \land x < n\}$

  $\{x+1 \leq n\}$

  x := x+1

  $\{x \leq n\}$

$\{x \leq n \land \neg(x < n)\}$

$\{x = n\}$

Assign. rule

Cons. rule

while rule

Cons. rule

Augmented Example

{$0 \leq n$ }

x := 0

y := 1

while(x < n)

        x := x+1

        y := y * x

{$y = n!$}

$\{0 \leq n \}$

x := 0

y := 1

$\{x \leq n \wedge y = x!\}$

while(x < n)

x := x+1

y := y * x

$\{y = n!\}$

Add Intermediate Assertions, Then Separate

$\{0 \leq n\}$

x := 0

y := 1

$\{x \leq n \wedge y = x!\}$

while(x < n)

    $\{x < n \wedge x \leq n \wedge y = x!\}$

    x := x+1

    y := y * x

    $\{x \leq n \wedge y = x!\}$

$\{y = n!\}$

Initialization

$\{0 \leq n\}$

x := 0

y := 1

$\{x \leq n \wedge y = x!\}$

$\{0 \leq n\}$

$\{0 \leq n \wedge 1 = 0!\}$

x := 0

$\{x \leq n \wedge 1 = x!\}$

y := 1

$\{x \leq n \wedge y = x!\}$

$\{x < n \wedge x \leq n \wedge y = x!\}$

x := x+1

y := y * x

$\{x \leq n \wedge y = x!\}$

Using Assignment Rule (working backward) & Consequent

$\{x < n \wedge x \leq n \wedge y = x!\}$

$\{x+1 \leq n \wedge y*(x+1) = (x+1)!\}$

x := x+1

$\{x \leq n \wedge y*x = x!\}$

y := y * x

$\{x \leq n \wedge y = x!\}$

The second line is a consequent of the first:

- $x < n$ implies $x+1 \leq n$ by integer arithmetic
- $y = x!$ implies $y*(x+1) = (x+1)!$ by definition of !

Finalization
Use Consequent Rule

$\mathbf{\neg(x < n)} \wedge \{x \leq n \wedge y = x!\}$

$\{y = n!\}$

$\neg(x < n) \wedge x \leq n \wedge y = x!$

implies $x = n \wedge y = x!$

which implies $y = n!$

Incorrect Example

This similar program is **not correct.**
This shows up when we try to prove it.

$\{0 \leq n\}$

```
x := 0
y := 1
while( x < n )
     y := y * x
     x := x+1
```

$\{y = n!\}$

Add Intermediate Assertions, Then Separate

$\{0 \leq n\}$

x := 0

y := 1

$\{x \leq n \wedge y = x!\}$

while(x < n)

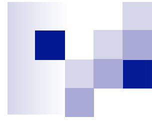
    $\{x < n \wedge x \leq n \wedge y = x!\}$

    y := y * x

    x := x+1

    $\{x \leq n \wedge y = x!\}$

$\{y = n!\}$



Initialization and Finalization

These are the same as before.

$\{x < n \wedge x \leq n \wedge y = x!\}$

y := y * x
x := x+1

$\{x \leq n \wedge y = x!\}$

$\{x < n \land x \leq n \land y = x!\}$

$\{x+1 \leq n \land y*x = (x+1)!\}$

y := y * x

$\{x+1 \leq n \land y = (x+1)!\}$

x := x+1

$\{x \leq n \land y = x!\}$

At the top, **the proof grinds to a halt**. We cannot get $y*x = (x+1)!$ in the second line from $y = x!$ in the first, as $x*x! = (x+1)!$ is false.

Other Loops with Invariants

$\{x \geq m\}$

```
while( x > m )
      x := x - 1
```

$\{x = m\}$

(e.g. $m = 0$)

What is the invariant?

Assume a is an array of size n.

```
{n ≥ 0}
sum = 0
i = 0
{… ? …}
while i < n:
    sum = sum + a[i]
    i = i + 1
{sum = a[0] + a[1] + … + a[n-1]}
```

What is the invariant?

Assume a is an array of size n.

```
{n ≥ 0}
sum = 0
i = n-1
{… ? …}
while i >= 0:
    sum = sum + a[i]
    i = i - 1
{sum = a[n-1] +  … + a[0]}
```

Assume x is a list.

$\{x = (x_1\ x_2 \ldots x_n)\}$

sum = 0

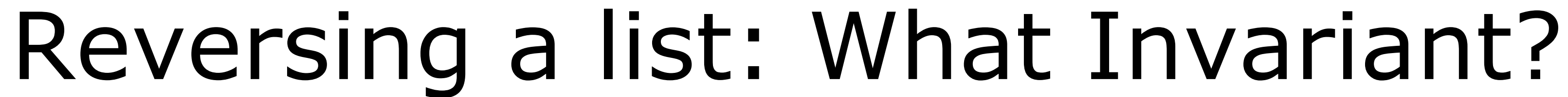
{… ? …}

while x != ():

    sum = sum + first(x)

    x = rest(x)

$\{sum = x_1 + x_2 + \ldots + x_n\}$



Reversing a list: What Invariant?

$\{x = x_0\}$

y = ()

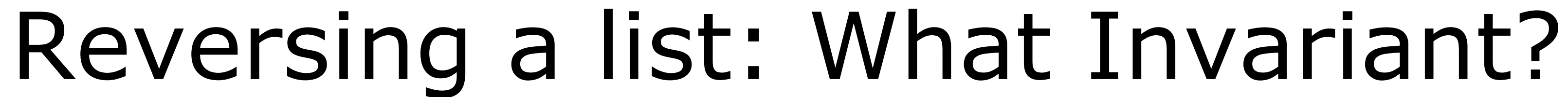
{… ? …}

while x != ():

    y = cons(first(x), y)

    x = rest(x)

$\{y = \text{reverse}(x_0)\}$



Reversing a list: What Invariant?

$\{x = x_0\}$

y = () empty

$\{\text{reverse}(x) + y = \text{reverse}(x_0)\}$ + is concat.

while x != ():

    y = cons(first(x), y)

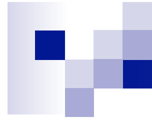
    x = rest(x)

$\{y = \text{reverse}(x_0)\}$ Since reverse(()) = ()
and () + y = y.

Finding Index of minimum

Assume a is an array of size $n \geq 1$.

```
{n ≥ 1}
min_index = 0
i = 0
{… ? …}
while i < n:
    if a[i] < a[min_index]:
        min_index = i
    i = i + 1
{a[min_index] ≤ min(a[0], a[1],  …,  a[n-1])}
```



Nested Loops

- When there are nested loops, we need an invariant for each level.
- We prove the invariants one level at a time.

Example: Selection Sorting Program

Array a = a[0] … a[n-1] of size n is initially a_0

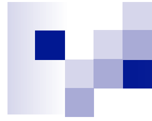
```
{a = a_0}
k = 0
{… ? …}
while k < n:
        min_index = k
        i = k
        {… ? …}
        while i < n:
                if a[i] < a[min_index]:
                    min_index = i
                i = i + 1
        a[k], a[min_index] = a[min_index], a[k]
     k = k + 1
```

{a[0] $\leq$ a[1] $\leq$ . . . $\leq$ a[n-1] and same_values(a, a_0)}

Example: Insertion Sorting Program

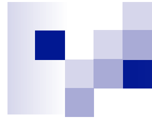
Array a = a[0] … a[n-1] of size n is initially a_0

```
{a = a0}
i = 1
{…?...}
while i < n:
    value = a[i]
    k = i
    {…?...}
    while k > 0 and value < a[k - 1]:    # find where value belongs
        a[k] = a[k - 1]                  # shifting other things up
        k -= 1
    a[k] = value
    i += 1
{a[0] ≤ a[1] ≤ . . . ≤ a[n-1] and same_values(a, a0)}
```



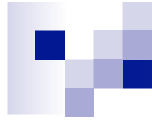
What Invariants can Do for You

- If you try to prove your program and find you cannot achieve closure, it might point out an error in your code.
- Or it could be an error in one of your assertions.
- Either way, assertions provide a kind of checks-and-balance.



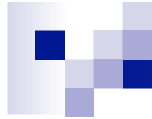
Can’t Testing do This?

- Testing can show that errors are present.
- In most cases, testing cannot show that errors are absent.
- Model-checking can be used to approach the exceptional cases.



Other Invariants to Ponder

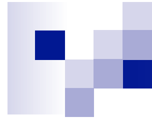
- Quicksort
- Binary search
- Dijkstra’s Algorithm
- Gaussian Elimination
- Garbage Collection



Partial vs. Total Correctness

- So far, only dealt with "partial correctness":
  - **If** the assumption is true **and** the program terminates, **then** the expectation will be true.

- Of greater interest is "total correctness":
  - **If** the assumption is true, **then** the program **terminates** with the expectation being true.



Partial vs. Total Correctness

- Total Correctness =

  Partial Correctness + Termination

How to Prove Termination?

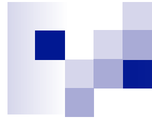
- A program terminates if it progress inexorably to a final state.

- Identify a function η on states (called a **variant**):

  η: States $\rightarrow$ N (**Natural Numbers**)

  such that, **on every iteration**, η **decreases** in value.

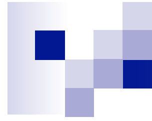
- Because the range of η is **non-negative**, there is a limit to the number of iterations.



Variants in Loops

- Generally want
  - $\eta > 0$ at start of loop body.
  - $\eta = 0$ at loop exit.

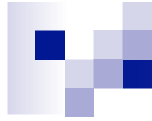
- May use assertions/invariant to prove this is the case.



Termination Example

```
n := n0;
while( n > 0 )
    {
    …
    n := n-1;
    }
```

What is an acceptable η in this case?



Termination Example 2

```
n := 0;
while( n < n0 )
    {
    …
    n := n+1;
    }
```

What is an acceptable η in this case?

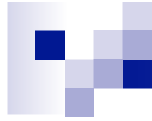
Total Correctness Example: gcd

$\{m_0 > 0 \wedge n_0 > 0\}$ // assumption

$m := m_0$; $n := n_0$;

```
while( ¬(m = n) )
    {
    if( m < n ) n := n-m; else m := m-n;
    }
```

$\{m = gcd(m_0, n_0)\}$ // expectation



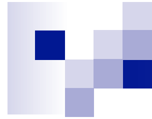
Proof of the previous program

- What is the loop invariant?

- What is an appropriate variant?

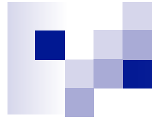
Lemmas for gcd

- gcd(A, A) = A
- gcd(A, B) = X $\vdash$ gcd(B, A) = X
- gcd(A, B) = X $\vdash$ gcd(A-B, B) = X
- gcd(A, B) = X $\vdash$ gcd(A, B-A) = X



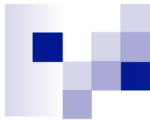
Informal Proof of gcd(A, B) = X $\vdash$ gcd(A-B, B) = X

- Show that pairs {A, B} and {A-B, B} have the **same** divisors. Therefore they have the same gcd.
- If d divides both A and B, then there are A' and B' such that A=dA' and B=dB'.
- But then A-B = d(A'-B'), so d divides A-B as well.
- Conversely, if d divides both A-B and B, then d divides (A-B)+B, which is A.



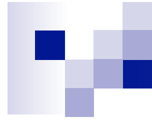
Proposed GCD Loop Invariant

- gcd(m, n) = gcd(m0, n0)



Choice of variant

- The variant must be chosen so that the two goals are provable.
- It may be necessary to **revisit the invariant**, to add to it conditions that make the goals provable.



Invariants in Object-Oriented Programming

- Often want methods to preserve an invariant when called.
  - Example: Sorted List
    - insert and remove maintain the invariant that the list is sorted.
  - Example: Priority Queue using a Heap
    - insert and remove maintain the heap invariant.