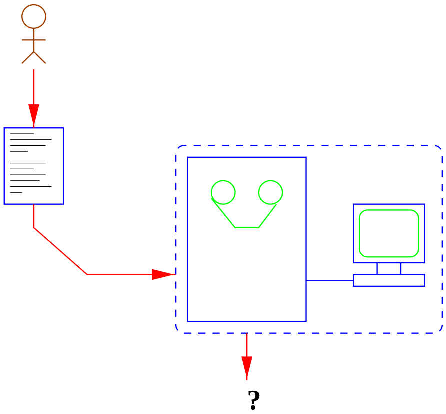
Recalling Our Intro to the Course

The Program Correctness Problem



- Conventional models of using computers – not easy to determine correctness!
  - Has become a very important issue, not just in safety-critical apps.
  - Components with assured quality, being able to give a warranty, ...
  - Being able to run untrusted code, certificate carrying code, ...

A Simple Imperative Program

- Example:

```
#include <stdio.h>
main() {
  int Number, Square;
  Number = 0;
       while(Number <= 5)
         { Square = Number * Number;
           printf("%d\n",Square);
           Number = Number + 1; } }
```

- Is it correct? With respect to what?
- A suitable formalism:
  - ◇ to provide *specifications* (describe problems), and
  - ◇ to reason about the *correctness of programs* (their *implementation*).

  is needed.

Natural Language

"Compute the squares of the natural numbers which are less or equal than 5."

Ideal at first sight, but:

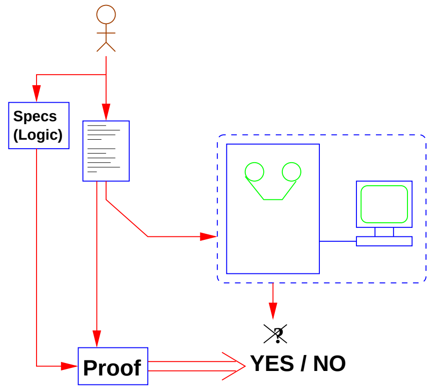
- ◇ verbose
- ◇ vague
- ◇ ambiguous
- ◇ needs context (assumed information)
- ◇ ...

Philosophers and Mathematicians already pointed this out a long time ago...

Logic

- A means of clarifying / formalizing the human thought process
- Logic for example tells us that (classical logic)
  Aristotle likes cookies, and
  Plato is a friend of anyone who likes cookies
  imply that
  Plato is a friend of Aristotle
- Symbolic logic:
  A shorthand for classical logic – plus many useful results:
  $a_1 : likes(aristotle, cookies)$
  $a_2 : \forall X \; likes(X, cookies) \rightarrow friend(plato, X)$
  $t_1 : friend(plato, aristotle)$
  $T[a_1, a_2] \vdash t_1$
- But, can logic be used:
  - To represent the problem (specifications)?
  - *Even perhaps to solve the problem?*

Using Logic



- For expressing specifications and reasoning about the correctness of programs we need:
  - Specification languages (assertions), modeling, ...
  - Program semantics (models, axiomatic, fixpoint, ...).
  - Proofs: program *verification* (and debugging, equivalence, ...).

Generating Squares: A Specifi cation (I)

Numbers —we will use “Peano” representation for simplicity:
0 → 0   1 → s(0)   2 → s(s(0))   3 → s(s(s(0)))   ...

- Defining the natural numbers:
  $nat(0) \wedge nat(s(0)) \wedge nat(s(s(0))) \wedge \ldots$
- A better solution:
  $nat(0) \wedge \forall X\ (nat(X) \rightarrow nat(s(X)))$

- Order on the naturals:
  $\forall X\ (le(0, X)) \wedge$
  $\forall X \forall Y\ (le(X, Y) \rightarrow le(s(X), s(Y))$
- Addition of naturals:
  $\forall X\ (nat(X) \rightarrow add(0, X, X)) \wedge$
  $\forall X \forall Y \forall Z\ (add(X, Y, Z) \rightarrow add(s(X), Y, s(Z)))$

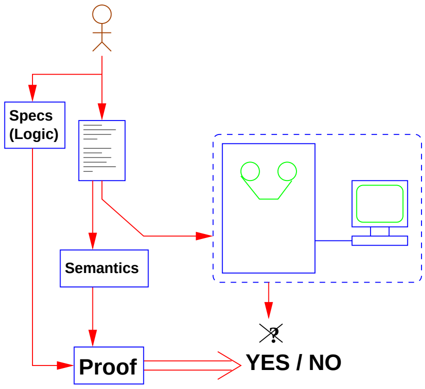
Generating Squares: A Specifi cation (II)

- Multiplication of naturals:
  $\forall X\ (nat(X) \rightarrow mult(0, X, 0)) \wedge$
  $\forall X \forall Y \forall Z \forall W\ (mult(X, Y, W) \wedge add(W, Y, Z) \rightarrow mult(s(X), Y, Z))$
- Squares of the naturals:
  $\forall X \forall Y\ (nat(X) \wedge nat(Y)\ \wedge mult(X, X, Y) \rightarrow nat\_square(X, Y))$

We can now write a *specification* of the (imperative) program, i.e., conditions that we want the program to meet:

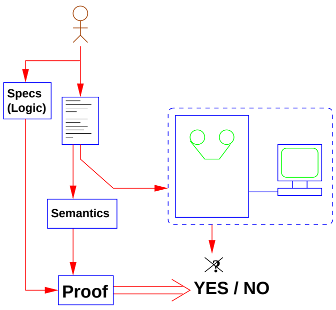
- *Precondition:*
  empty.
- *Postcondition:*
  $\forall X (output(X) \leftarrow (\exists Y\ nat(Y) \wedge le(Y, s(s(s(s(s(0)))))) \wedge nat\_square(Y, X)))$

Use of Logic



- For expressing specifications and reasoning about the correctness of programs we need:
  - Specification languages (assertions), modeling, ...
  - Program semantics (models, axiomatic, fixpoint, ...).
  - Proofs: program *verification* (and debugging, equivalence, ...).

Semantic Tasks



- Semantics:
  - A *semantics* associates a meaning (a mathematical object) to a program or program sentence.
- Semantic tasks:
  - Verification: proving that a program meets its specification.
  - Static debugging: finding where a program does not meet specifications.
  - Program equivalence: proving that two programs have the same semantics.
  - etc.

Styles of Semantics

- **Operational:**
  The meaning of program sentences is defined in terms of the steps (transformations from state to state) that computations may take during execution (derivations). Proofs by induction on derivations.
- **Axiomatic:**
  The meaning of program sentences is defined indirectly in terms of some axioms and rules of a *logic* of program properties.
- **Denotational (fixpoint):**
  The meaning of program sentences is given abstractly as *functions* on an appropriate *domain* (which is often a lattice). E.g., λ-calculus for functional programming. C.f., lattice / fixpoint theory.
- Also, **model (declarative) semantics:** (For (Constraint) Logic Programs:) The meaning of programs is given as a minimal model ("logical meaning") of the logic that the program is written in.

Operational Semantics

Traditional Operational Semantics

- Meaning of program sentences defined in terms of the steps (*state transitions*, transformations from state to state) that computations may take during executions (derivations).
- Proofs by induction on derivations.
- Examples of concrete operational semantics:
  - ◇ Semantics modeling memory for imperative programs.
  - ◇ Interpreters and meta-interpreters (self-interpreters).
  - ◇ Resolution and CLP($\mathcal{X}$) resolution, for (constraint) logic programs.
  - ◇ ...
- Examples of generic / standard methodologies:
  - ◇ *Structural operational semantics.*
  - ◇ Vienna definition language (VDL).
  - ◇ SECD machine.
  - ◇ ...

A Simple Imperative Language

```
Program    ::= Statement
Statement  ::= Statement ; Statement
           |   noop
           |   Id := Expression
           |   if Expression then Statement else Statement
           |   while Expression do Statement
Expression ::= Numeral
           |   Id
           |   Expression + Expression
```

- Only integer data types.
- Variables do not need to be declared.

Operational Semantics

- States: memory configurations –values of variables.
- $s[X]$ denotes the value of the variable X in state s.
- $< statement, s > \Rightarrow s'$ denotes that
  if $statement$ is executed in state s the resulting state is s'.
- $< expression, s > \Rightarrow value$ denotes that
  if $expression$ is executed in state s it returns $value$.
- Expressions:
  - If n is a number $< n, s > \Rightarrow n$
  - If X is a variable $< X, s > \Rightarrow s[X]$
  - If $expression$ is of the form exp_1+exp_2 we write:

$$\frac{< exp_1, s > \Rightarrow v_1 \qquad < exp_2, s > \Rightarrow v_2}{< exp_1\text{+}exp_2, s > \Rightarrow v_1 + v_2}$$

Operational Semantics

- Statements:
  $s[X/v]$ denotes a new state, identical to s but where variable X has value v.
  - Noop: $<$ **noop** $, s > \Rightarrow s$
  - Assignment:

$$\frac{< exp, s > \Rightarrow v}{< X \textbf{ := } exp, s > \Rightarrow s[X/v]}$$

  - Conditional:

$$\frac{< exp, s > \Rightarrow 0 \qquad < stmt_2, s > \Rightarrow s'}{< \textbf{ if } exp \textbf{ then } stmt_1 \textbf{ else } stmt_2, s > \Rightarrow s'}$$

$$\frac{< exp, s > \Rightarrow v, v \neq 0 \qquad < stmt_1, s > \Rightarrow s'}{< \textbf{ if } exp \textbf{ then } stmt_1 \textbf{ else } stmt_2, s > \Rightarrow s'}$$

Operational Semantics

- Statements (Contd.):
  - Sequencing:

$$\frac{< stmt_1, s > \Rightarrow s_1 \qquad < stmt_2, s_1 > \Rightarrow s_2}{< stmt_1 \; \mathbf{;} \; stmt_2, s > \Rightarrow s_2}$$

  - Loops:

$$\frac{< exp, s > \Rightarrow 0}{< \textbf{while } exp \textbf{ do } stmt, s > \Rightarrow s}$$

$$\frac{< exp, s > \Rightarrow v, v \neq 0 \qquad < stmt, s > \Rightarrow s' \qquad < \textbf{while } exp \textbf{ do } stmt, s' > \Rightarrow s''}{< \textbf{while } exp \textbf{ do } stmt, s > \Rightarrow s''}$$

Example

- Program:

```
x := 5;
y := -6;
if (x+y) then z := x else z := y
```

- Semantics:

$$\frac{< x \mathbf{:=} 5, \; s_0 > \Rightarrow s_1 \qquad \dfrac{< y \mathbf{:=} -6, \; s_1 > \Rightarrow s_2 \qquad \dfrac{\dfrac{< x\mathbf{+}y, s_2 > \Rightarrow -1 \qquad < z \mathbf{:=} x, s_2 > \Rightarrow s_3}{< S_3, \; s_2 > \Rightarrow s_3}}{}}{< y \mathbf{:=} -6 \, \mathbf{;} \, S_3, \; s_1 > \Rightarrow s_3}}{< x \mathbf{:=} 5 \, \mathbf{;} \, y \mathbf{:=} -6 \, \mathbf{;} \, S_3, \; s_0 > \Rightarrow s_3}$$

where $S_3 =$ `if (x+y) then z := x else z := y`.
And:
$s_1 = s_0[x/5]$
$s_2 = s_1[y/-6]$
$s_3 = s_2[z/5]$

Axiomatic Semantics

Axiomatic Semantics

- **Characteristics:**
  - ◇ Based on techniques from predicate logic.
  - ◇ There is no concept of *state of the machine* (as in operational or denotational semantics).
  - ◇ More abstract than, e.g., denotational semantics.
  - ◇ Semantic meaning of a program is based on assertions about relationships that remain the same each time the program executes.
- **Classical application:**
  - ◇ Proving programs to be correct w.r.t. specifications.
- **(Typical, classical) limitations:**
  - ◇ Side-effects disallowed in expressions.
  - ◇ `goto` command difficult to treat.
  - ◇ Aliasing not allowed.
  - ◇ Scope rules difficult to describe $\Rightarrow$ require all identifier names to be unique.

History and References

- Main original papers:
  - ◇ 1967: Floyd. *Assigning Meanings to Programs.*
  - ◇ 1969: Hoare. *An Axiomatic Basis of Computer Programming.*
  - ◇ 1976: Dijkstra. *A Discipline of Programming.*
  - ◇ 1981: Gries. *The Science of Programming.*
- Many textbooks available.

Assertions and Correctness

- **Assertion:** a logical formula, say

  $$(m \neq 0 \wedge (\sqrt{m})^2 = m)$$

  that is true when a point in the program is reached.
- **Precondition:** Assertion before a command (← *includes a whole program*).
- **Postcondition:** Assertion after a command.

$\boxed{\{PRE\}\ \text{C}\ \{POST\}}$ ← a "Hoare triple"

- **Partial Correctness:**
  If the initial assertion (the precondition) is true and if the program terminates, then the final assertion (the postcondition) must be true.
  Precondition + Termination ⇒ Postcondition
- **Total Correctness:**
  Given that the precondition for the program is true, the program must terminate and the postcondition must be true.
  Total Correctness = Partial Correctness + Termination

Hoare Calculus: The Assignment Axiom

- Examples:
  - $\{true\}$ `m := 13` $\{m = 13\}$
  - $\{n = 3 \wedge c = 2\}$ `n := c*n` $\{n = 6 \wedge c = 2\}$
  - $\{k \geq 0\}$ `k := k + 1` $\{k > 0\}$
- Notation:
  - {*Precondition*} *command* {*Postcondition*}
  - $P[V \to E]$ denotes substitution: putting E in place of V in P
- **Axiom for assignment command:**

$$\boxed{\{P[V \to E]\}\ V \mathbin{\textbf{:=}} E\ \{P\}}$$

Work backwards:

- Postcondition: $P \equiv (n = 6 \wedge c = 2)$
- Command: n **:=** c*n
- Precondition: $P[V \to E] \equiv (c * n = 6 \wedge c = 2)$
  $\equiv (n = 3 \wedge c = 2)$

Hoare Calculus: Read and Write Commands

- **Notation:**
  - Use "$IN = [1, 2, 3]$" and "$OUT = [4, 5]$" to represent input and output files.
  - $[\text{M}|L]$ denotes list whose head is M and tail is L.
  - K, M, N, … represent arbitrary numerals.
- **Axiom for read command:**
  - $\{IN = [\text{K}|L] \wedge P[V \to \text{K}]\}$ **read** V $\{IN = L \wedge P\}$
- **Axiom for write command:**
  - $\{OUT = L \wedge E = \text{K} \wedge P\}$ **write** E $\{OUT = L :: [\text{K}] \wedge E = \text{K} \wedge P\}$
- Note: $L :: [\text{K}]$ is the list whose last element is K ($::$ represents concatenation).

Hoare Calculus: Rules of Inference

- **Format** (c.f. structural operational semantics)**:**

$$\frac{H_1, H_2, H_n, ...}{H}$$

- **Axiom for Command Sequencing:**

$$\frac{\{P\}C_1\{Q\},\quad \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}}$$

- **Axioms for If Commands:**

$$\frac{\{P \wedge b\}C_1\{Q\},\quad \{P \wedge \neg b\}C_2\{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{endif}\ \{Q\}}$$

$$\frac{\{P \wedge b\}C\{Q\},\quad (P \wedge \neg b) \rightarrow Q}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ C\ \textbf{endif}\ \{Q\}}$$

Hoare Calculus: Rules of Inference (Contd.)

- **Weaken Postcondition:**

$$\frac{\{\mathrm{P}\}C\{\mathrm{Q}\},\ Q \rightarrow R}{\{\ \mathrm{P}\ \}C\{\ \mathrm{R}\ \}}$$

- **Strengthen Precondition:**

$$\frac{P \rightarrow Q,\ \{\mathrm{Q}\}C\{\mathrm{R}\}}{\{\ \mathrm{P}\ \}C\{\ \mathrm{R}\ \}}$$

- **And and Or Rules:**

$$\frac{\{P\}C\{Q\},\ \{P'\}C\{Q'\}}{\{P \wedge P'\}C\{Q \wedge Q'\}}$$

$$\frac{\{P\}C\{Q\},\ \{P'\}C\{Q'\}}{\{P \vee P'\}C\{Q \vee Q'\}}$$

- **Observation:**

{ *false* } *any-command* { *any-postcondition* }

Example (I)

$\{IN = [4, 9, 16] \wedge OUT = [0, 1, 2]\}$
read m; **read** n;
if m $\geq$ n **then**
a **:=** 2*m
else
a **:=** 2*n
endif;
write a
$\{IN = [16] \wedge OUT = [0, 1, 2, 18]\}$

$\{IN = [4, 9, 16] \wedge OUT = [0, 1, 2]\} \rightarrow \{IN = [4|[9, 16]] \wedge OUT = [0, 1, 2] \wedge 4 = 4\}$
read m;
$\{IN = [9, 16] \wedge OUT = [0, 1, 2] \wedge m = 4\} \rightarrow$
$\{IN = [9|[16]] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge 9 = 9\}$
read n;
$\{IN = [16] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge n = 9\}$

Recall:
$\{IN = [\kappa|L] \wedge P[V \rightarrow \kappa]\}$
read V
$\{IN = L \wedge P\}$

Example (II)

We have $P = \{IN = [16] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge n = 9\}$
read m; **read** n;
if m $\geq$ n **then**
a **:=** 2*m
else
a **:=** 2*n
endif;
write a

$$\frac{\{P \wedge b\}C_1\{Q\}, \quad \{P \wedge \neg b\}C_2\{Q\}}{\{P\} \textbf{ if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ endif } \{Q\}}$$

So, $b \equiv m \geq n = false$ and $\neg b = true$; thus $\{P \wedge b\} = false$ and $\{P \wedge \neg b\} = P$.
So, for C_2 we have:
$\{P \wedge \neg b\} = \{P\} =$
$\{IN = [16] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge n = 9\} \rightarrow$
$\{IN = [16] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge n = 9 \wedge 2 * n = 18\}$
a **:=** 2*n
$\{IN = [16] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge n = 9 \wedge a = 18\}$

$\{P[V \rightarrow E]\}$ V **:=** E $\{P\}$

and for C_1 we can have anything since the premise is false:
$\{P \wedge b\} = false$
a **:=** 2*m
$\{IN = [16] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge n = 9 \wedge a = 18\}$

Example (III)

$\{IN = [16] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge n = 9\}$

if m $\geq$ n **then**

a **:=** 2*m

else

a **:=** 2*n

endif;

$\{IN = [16] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge n = 9 \wedge a = 18\}$

and

$\{IN = [16] \wedge OUT = [0, 1, 2] \wedge m = 4 \wedge n = 9 \wedge a = 18\}$

write a

$\{IN = [16] \wedge OUT = [0, 1, 2] :: [18] \wedge m = 4 \wedge n = 9 \wedge a = 18\}$

which implies

$\{IN = [16] \wedge OUT = [0, 1, 2, 18]\}$

While Command

$$\frac{\{P \wedge b\}C\{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ C\ \textbf{endwhile}\ \{P \wedge \neg b\}}$$

- **Loop Invariant:** P
  - Preserved during execution of the loop.
- **Loop steps:**
  - *Initialization:* show that the loop invariant $\{P\}$ is initially true.
  - *Preservation:* show the loop invariant remains true when the loop executes ($\{P \wedge b\}$).
  - *Completion:* show that the loop invariant and the exit condition produce the final assertion ($\{P \wedge \neg b\}$).
- **Main Problem:**
  - Constructing the loop invariant.

Loop Invariant

- A relationship among the variables that does not change as the loop is executed.
- “Inspiration” tips:
  - Look for some expression that can be combined with $\neg b$ to produce part of the postcondition.
  - Construct a table of values to see what stays constant.
  - Combine what has already been computed at some stage in the loop with what has yet to be computed to yield a constant of some sort.

Study carefully many examples!

Example (exponent)

$\{N \geq 0 \wedge A \geq 0\}$

```
 k := N;    s := 1;
 while      k>0 do
            s := A*s;
            k := k-1
 endwhile
```

$\{s = A^N\}$

We follow the “tips:”

- Trace algorithm with small numbers $A = 2$, $N = 5$.
- Build a table of values to find loop invariant.
- Notice that k is decreasing and that 2^k represents the computation that still needs to be done.
- Add a column to the table for the value of 2^k.
- The value $s * 2^k = 32$ remains constant throughout the execution of the loop.

Example (Exponent)

$\{N \geq 0 \wedge A \geq 0\}$

```
k := N;     s := 1;
while       k>0 do
            s := A*s;
            k := k-1
endwhile
```

$\{s = A^N\}$

k	s	2^k	s*2^k
5	1	32	32
4	2	16	32
3	4	8	32
2	8	4	32
1	16	2	32
0	32	1	32

- Observe that s and 2^k change when k changes.
- Their product is constant, namely $32 = 2^5 = A^N$.
- This suggests that $s * A^k = A^N$ is part of the invariant.
- The relation $k \geq 0$ seems to be invariant, and when combined with "$\neg b$", which is $k \leq 0$, establishes $k = 0$ at the end of the loop.
- When $k = 0$ is joined with $s * A^k = A^N$, we get the postcondition $s = A^N$.

Loop Invariant: $\{k \geq 0 \wedge s * A^k = A^N\}$.

Verifi cation of the Program

Initialization:

$\{N \geq 0 \wedge A \geq 0\} \rightarrow \{N = N \wedge N \geq 0 \wedge A \geq 0 \wedge 1 = 1\}$
  k := N; s := 1;
$\{k = N \wedge N \geq 0 \wedge A \geq 0 \wedge s = 1\} \rightarrow \{k \geq 0 \wedge s * A^k = A^N\}$

Preservation:

$\{k \geq 0 \wedge s * A^k = A^N \wedge k > 0\} \rightarrow \{k > 0 \wedge s * A^k = A^N\} \rightarrow$
$\{k > 0 \wedge s * A * A^{k-1} = A^N\} \rightarrow \{k > 0 \wedge A * s * A^{k-1} = A^N\}$
  s := A*s;
$\{k > 0 \wedge s * A^{k-1} = A^N\} \rightarrow \{k - 1 \geq 0 \wedge s * A^{k-1} = A^N\}$
  k := k-1
$\{k \geq 0 \wedge s * A^k = A^N\}$

Completion:

$\{k \geq 0 \wedge s * 2^k = A^N \wedge k \leq 0\} \rightarrow \{k = 0 \wedge s * 2^k = A^N\} \rightarrow \{s = A^N\}$

Further Topics

- Dealing with other language features:
  - Nested loops.
  - Procedure calls.
  - Recursive procedures.
  - ...
- Proving termination / total correctness.
  - Well founded orderings.

Acknowledgments

- Some slides and examples taken from:
  - Enrico Pontelli
  - Jim Lipton
  - Ken Slonneger and Barry L. Kurtz.
    Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach.
    Addison-Wesley, Reading, Massachusetts.