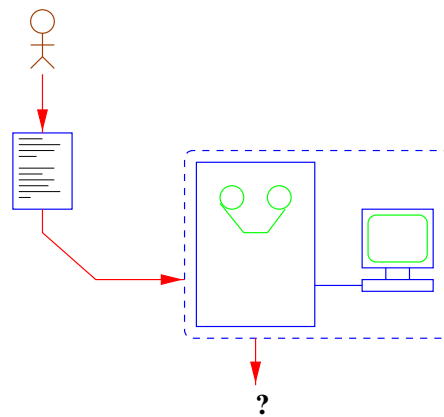


Recalling Our Intro to the Course

1

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, ...

2

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.

3

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...

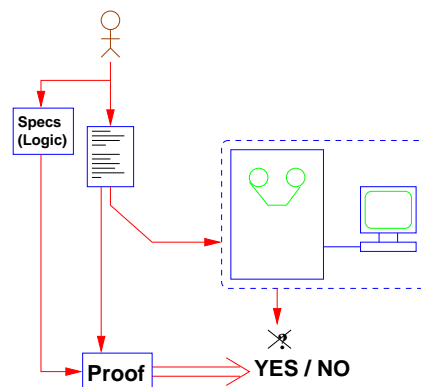
4

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:
 $a_1 : \text{likes}(\text{aristotle}, \text{cookies})$
 $a_2 : \forall X \text{ likes}(X, \text{cookies}) \rightarrow \text{friend}(\text{plato}, X)$
 $t_1 : \text{friend}(\text{plato}, \text{aristotle})$
 $T[a_1, a_2] \vdash t_1$
- But, can logic be used:
 - ◇ To represent the problem (specifications)?
 - ◇ *Even perhaps to solve the problem?*

5

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, ...).

6

Generating Squares: A Specification (I)

Numbers—we will use “Peano” representation for simplicity:

$0 \rightarrow 0$ $1 \rightarrow s(0)$ $2 \rightarrow s(s(0))$ $3 \rightarrow s(s(s(0)))$...

- Defining the natural numbers:
 $nat(0) \wedge nat(s(0)) \wedge nat(s(s(0))) \wedge \dots$
- 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:
 $\forall X (nat(X) \rightarrow add(0, X, X)) \wedge$
 $\forall X \forall Y \forall Z (add(X, Y, Z) \rightarrow add(s(X), Y, s(Z)))$

7

Generating Squares: A Specification (II)

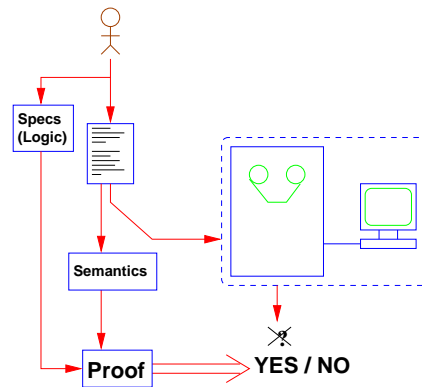
- Multiplication of naturals:
 $\forall X (nat(X) \rightarrow mult(0, X, 0)) \wedge$
 $\forall X \forall Y \forall Z \forall W (mult(X, Y, W) \wedge add(W, Y, Z) \rightarrow mult(s(X), Y, Z))$
- Squares of the naturals:
 $\forall X \forall Y (nat(X) \wedge nat(Y) \wedge mult(X, X, Y) \rightarrow nat_square(X,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

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

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

8

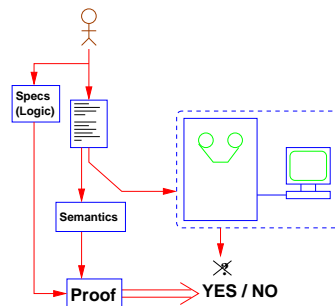
Use of Logic



- 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, ...).

9

Semantic Tasks



- Semantic tasks:
 - ◇ 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.

10

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(λ) resolution, for (constraint) logic programs.
 - ◇ ...
- Examples of generic / standard methodologies:
 - ◇ *Structural operational semantics*.
 - ◇ Vienna definition language (VDL).
 - ◇ SECD machine.
 - ◇ ...

13

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.

14

Operational Semantics

- States: memory configurations –values of variables.
- $s[X]$ denotes the value of the variable X in state s .
- $\langle \text{statement}, s \rangle \Rightarrow s'$ denotes that if *statement* is executed in state s the resulting state is s' .
- $\langle \text{expression}, s \rangle \Rightarrow \text{value}$ denotes that if *expression* is executed in state s it returns *value*.
- Expressions:
 - ◇ If n is a number $\langle n, s \rangle \Rightarrow n$
 - ◇ If X is a variable $\langle X, s \rangle \Rightarrow s[X]$
 - ◇ If *expression* is of the form $\text{exp}_1 + \text{exp}_2$ we write:

$$\frac{\langle \text{exp}_1, s \rangle \Rightarrow v_1 \quad \langle \text{exp}_2, s \rangle \Rightarrow v_2}{\langle \text{exp}_1 + \text{exp}_2, s \rangle \Rightarrow v_1 + v_2}$$

15

Operational Semantics

- Statements:
 - $s[X/v]$ denotes a new state, identical to s but where variable X has value v .
 - ◇ Noop: $\langle \text{noop}, s \rangle \Rightarrow s$
 - ◇ Assignment:

$$\frac{\langle \text{exp}, s \rangle \Rightarrow v}{\langle X := \text{exp}, s \rangle \Rightarrow s[X/v]}$$
 - ◇ Conditional:

$$\frac{\langle \text{exp}, s \rangle \Rightarrow 0 \quad \langle \text{stmt}_2, s \rangle \Rightarrow s'}{\langle \text{if exp then stmt}_1 \text{ else stmt}_2, s \rangle \Rightarrow s'}$$

$$\frac{\langle \text{exp}, s \rangle \Rightarrow v, v \neq 0 \quad \langle \text{stmt}_1, s \rangle \Rightarrow s'}{\langle \text{if exp then stmt}_1 \text{ else stmt}_2, s \rangle \Rightarrow s'}$$

16

Operational Semantics

- Statements (Contd.):

- ◇ Sequencing:

$$\frac{\langle stmt_1, s \rangle \Rightarrow s_1 \quad \langle stmt_2, s_1 \rangle \Rightarrow s_2}{\langle stmt_1 ; stmt_2, s \rangle \Rightarrow s_2}$$

- ◇ Loops:

$$\frac{\langle exp, s \rangle \Rightarrow 0}{\langle \text{while } exp \text{ do } stmt, s \rangle \Rightarrow s}$$

$$\frac{\langle exp, s \rangle \Rightarrow v, v \neq 0 \quad \langle stmt, s \rangle \Rightarrow s' \quad \langle \text{while } exp \text{ do } stmt, s' \rangle \Rightarrow s''}{\langle \text{while } exp \text{ do } stmt, s \rangle \Rightarrow s''}$$

Example

- Program:

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

- Semantics:

$$\frac{\langle x := 5, s_0 \rangle \Rightarrow s_1 \quad \frac{\langle y := -6, s_1 \rangle \Rightarrow s_2 \quad \frac{\langle x+y, s_2 \rangle \Rightarrow -1 \quad \langle z := x, s_2 \rangle \Rightarrow s_3}{\langle S_3, s_2 \rangle \Rightarrow s_3}}{\langle y := -6 ; S_3, s_1 \rangle \Rightarrow s_3}}{\langle x := 5 ; y := -6 ; S_3, s_0 \rangle \Rightarrow s_3}$$

where $S_3 = \text{if } (x+y) \text{ then } z := x \text{ else } z := y.$

And:

$$s_1 = s_0[x/5]$$

$$s_2 = s_1[y/-6]$$

$$s_3 = s_2[z/5]$$

Axiomatic Semantics

19

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.

20

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.

21

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 (\leftarrow includes a whole program).
- **Postcondition:** Assertion after a command.

$$\boxed{\{PRE\} C \{POST\}}$$

\leftarrow 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.

$$Precondition + Termination \Rightarrow Postcondition$$

- **Total Correctness:**

Given that the precondition for the program is true, the program must terminate and the postcondition must be true.

$$Total\ Correctness = Partial\ Correctness + Termination$$

22

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:**

- ◇ $\{Precondition\} command \{Postcondition\}$
- ◇ $P[V \rightarrow E]$ denotes substitution: putting E in place of V in P

- **Axiom for assignment command:**

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

Work backwards:

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

Hoare Calculus: Read and Write Commands

- **Notation:**

- ◇ Use “ $IN = [1, 2, 3]$ ” and “ $OUT = [4, 5]$ ” to represent input and output files.
- ◇ $[M|L]$ denotes list whose head is M and tail is L .
- ◇ K, M, N, \dots represent arbitrary numerals.

- **Axiom for read command:**

- ◇ $\{IN = [K|L] \wedge P[V \rightarrow K]\} \mathbf{read} V \{IN = L \wedge P\}$

- **Axiom for write command:**

- ◇ $\{OUT = L \wedge E = K \wedge P\} \mathbf{write} E \{OUT = L :: [K] \wedge E = K \wedge P\}$

- **Note:** $L :: [K]$ is the list whose last element is K ($::$ represents concatenation).

Hoare Calculus: Rules of Inference

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

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

- **Axiom for Command Sequencing:**

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

- **Axioms for If Commands:**

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

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

25

Hoare Calculus: Rules of Inference (Contd.)

- **Weaken Postcondition:**

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

- **Strengthen Precondition:**

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

- **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:**

$$\{ \text{false} \} \text{ any-command } \{ \text{any-postcondition} \}$$

26

Example (I)

```

{IN = [4, 9, 16] ∧ OUT = [0, 1, 2]}
read m; read n;
if m ≥ n then
    a := 2*m
    else
        a := 2*n
    endif;
write a
{IN = [16] ∧ 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] ∧ OUT = [0, 1, 2] ∧ m = 4} →
{IN = [9|[16]] ∧ OUT = [0, 1, 2] ∧ m = 4 ∧ 9 = 9}
read n;
{IN = [16] ∧ OUT = [0, 1, 2] ∧ m = 4 ∧ n = 9}

```

Recall:

```

{IN = [κ|L] ∧ P[V → κ]}
read V
{IN = L ∧ P}

```

27

Example (II)

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

```

read m; read n;
if m ≥ n then
    a := 2*m
    else
        a := 2*n
    endif;
write a

```

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

endif;
write a

So, $b \equiv m \geq n = \text{false}$ and $\neg b = \text{true}$; thus $\{P \wedge b\} = \text{false}$ and $\{P \wedge \neg b\} = P$.

So, for C_2 we have:

```

{P ∧ ¬b} = {P} =
{IN = [16] ∧ OUT = [0, 1, 2] ∧ m = 4 ∧ n = 9} →
{IN = [16] ∧ OUT = [0, 1, 2] ∧ m = 4 ∧ n = 9 ∧ 2 * n = 18}
a := 2*n
{IN = [16] ∧ OUT = [0, 1, 2] ∧ m = 4 ∧ n = 9 ∧ a = 18}

```

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

and for C_1 we can have anything since the premise is false:

```

{P ∧ b} = false
a := 2*m
{IN = [16] ∧ OUT = [0, 1, 2] ∧ m = 4 ∧ n = 9 ∧ a = 18}

```

28

Example (III)

```
{IN = [16] ∧ OUT = [0, 1, 2] ∧ m = 4 ∧ n = 9}
if m ≥ n then
    a := 2*m
    else
    a := 2*n
endif;
{IN = [16] ∧ OUT = [0, 1, 2] ∧ m = 4 ∧ n = 9 ∧ a = 18}
and
{IN = [16] ∧ OUT = [0, 1, 2] ∧ m = 4 ∧ n = 9 ∧ a = 18}
write a
{IN = [16] ∧ OUT = [0, 1, 2] :: [18] ∧ m = 4 ∧ n = 9 ∧ a = 18}
which implies
{IN = [16] ∧ OUT = [0, 1, 2, 18]}
```

29

While Command

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

- **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.

30

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 ≥ 0 ∧ A ≥ 0}
k := N;   s := 1;
while    k > 0 do
           s := A*s;
           k := k-1
endwhile
{s = AN}
```

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 ≥ 0 ∧ A ≥ 0}
k := N;   s := 1;
while    k > 0 do
    s := A*s;
    k := k-1
endwhile
{s = AN}

```

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

33

Verification of the Program

Initialization:

```

{N ≥ 0 ∧ A ≥ 0} → {N = N ∧ N ≥ 0 ∧ A ≥ 0 ∧ 1 = 1}
k := N; s := 1;
{k = N ∧ N ≥ 0 ∧ A ≥ 0 ∧ s = 1} → {k ≥ 0 ∧ s * Ak = AN}

```

Preservation:

```

{k ≥ 0 ∧ s * Ak = AN ∧ k > 0} → {k > 0 ∧ s * Ak = AN} →
{k > 0 ∧ s * A * Ak-1 = AN} → {k > 0 ∧ A * s * Ak-1 = AN}
s := A*s;
{k > 0 ∧ s * Ak-1 = AN} → {k - 1 ≥ 0 ∧ s * Ak-1 = AN}
k := k-1
{k ≥ 0 ∧ s * Ak = AN}

```

Completion:

```

{k ≥ 0 ∧ s * 2k = AN ∧ k ≤ 0} → {k = 0 ∧ s * 2k = AN} → {s = AN}

```

34

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.