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:

```c
#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.
“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:
  \[ a_1 : \text{likes}(\text{aristotle}, \text{cookies}) \]
  \[ a_2 : \forall X \text{ likes}(X, \text{cookies}) \rightarrow \text{friend}(\text{plato}, X) \]
  \[ t_1 : \text{friend}(\text{plato}, \text{aristotle}) \]
  \[ T[a_1, a_2] \vdash t_1 \]

- But, can logic be used:
  ◇ To represent the problem (specifications)?
  ◇ *Even perhaps to solve the problem?*
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 Specification (I)

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

\[ 0 \rightarrow 0 \quad 1 \rightarrow s(0) \quad 2 \rightarrow s(s(0)) \quad 3 \rightarrow s(s(s(0))) \ldots \]

- Defining the natural numbers:
  \[ \text{nat}(0) \land \text{nat}(s(0)) \land \text{nat}(s(s(0))) \land \ldots \]

- A better solution:
  \[ \text{nat}(0) \land \forall X (\text{nat}(X) \rightarrow \text{nat}(s(X))) \]

- Order on the naturals:
  \[ \forall X (\text{le}(0, X)) \land \]
  \[ \forall X \forall Y (\text{le}(X, Y) \rightarrow \text{le}(s(X), s(Y))) \]

- Addition of naturals:
  \[ \forall X (\text{nat}(X) \rightarrow \text{add}(0, X, X)) \land \]
  \[ \forall X \forall Y \forall Z (\text{add}(X, Y, Z) \rightarrow \text{add}(s(X), Y, s(Z))) \]
Generating Squares: A Specification (II)

- **Multiplication of naturals:**
  \[
  \forall X \ (\text{nat}(X) \rightarrow \text{mult}(0, X, 0)) \land
  \forall X \forall Y \forall Z \forall W \ (\text{mult}(X, Y, W) \land \text{add}(W, Y, Z) \rightarrow \text{mult}(s(X), Y, Z))
  \]

- **Squares of the naturals:**
  \[
  \forall X \forall Y \ (\text{nat}(X) \land \text{nat}(Y) \land \text{mult}(X, X, Y) \rightarrow \text{nat} \_ \text{square}(X, Y))
  \]

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

- **Precondition:**
  empty.

- **Postcondition:**
  \[
  \forall X \ (\text{output}(X) \leftrightarrow (\exists Y \ \text{nat}(Y) \land \text{le}(Y, s(s(s(s(0))))) \land \text{nat}_\_\text{square}(Y, X)))
  \]
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.
  - ...

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.
Operational Semantics

- States: memory configurations – values of variables.
- \( s[X] \) denotes the value of the variable \( X \) in state \( s \).
- \( < \text{statement}, s > \Rightarrow s' \) denotes that
  if \text{statement} is executed in state \( s \) the resulting state is \( s' \).
- \( < \text{expression}, s > \Rightarrow \text{value} \) denotes that
  if \text{expression} is executed in state \( s \) it returns \text{value}.

- Expressions:
  - If \( n \) is a number \( < n, s > \Rightarrow n \)
  - If \( X \) is a variable \( < X, s > \Rightarrow s[X] \)
  - If \text{expression} is of the form \( \text{exp}_1 + \text{exp}_2 \) we write:
    \[
    \frac{< \text{exp}_1, s > \Rightarrow v_1 \quad < \text{exp}_2, s > \Rightarrow v_2}{< \text{exp}_1 + \text{exp}_2, s > \Rightarrow v_1 + v_2}
    \]
Operational Semantics

- Statements:
  \( s[X/v] \) denotes a new state, identical to \( s \) but where variable \( X \) has value \( v \).

  - Noop: \(< \text{noop}, s >\) \(\Rightarrow s\)
  - Assignment:
    \[
    \frac{< \text{exp}, s >\Rightarrow v}{< X := \text{exp}, s >\Rightarrow s[X/v]}
    \]
  - Conditional:
    \[
    \frac{< \text{exp}, s >\Rightarrow 0 \quad < \text{stmt}_2, s >\Rightarrow s'}{< \text{if} \ \text{exp} \ \text{then} \ \text{stmt}_1 \ \text{else} \ \text{stmt}_2, s >\Rightarrow s'}
    \]
    \[
    \frac{< \text{exp}, s >\Rightarrow v, v \neq 0 \quad < \text{stmt}_1, s >\Rightarrow s'}{< \text{if} \ \text{exp} \ \text{then} \ \text{stmt}_1 \ \text{else} \ \text{stmt}_2, s >\Rightarrow s'}
    \]
Operational Semantics

- Statements (Contd.):
  - Sequencing:
    \[
    < stmt_1, s > \Rightarrow s_1 \quad < stmt_2, s_1 > \Rightarrow s_2 \\
    < stmt_1 ; stmt_2, s > \Rightarrow s_2
    \]
  - Loops:
    \[
    < exp, s > \Rightarrow 0 \\
    < \textbf{while} \ exp \ \textbf{do} \ stmt, s \ > \Rightarrow s \\
    < \exp, s > \Rightarrow v, v \neq 0 \quad < stmt, s > \Rightarrow s' \quad < \textbf{while} \ exp \ \textbf{do} \ stmt, s' > \Rightarrow s'' \\
    < \textbf{while} \ exp \ \textbf{do} \ stmt, s > \Rightarrow s''
    \]
Example

- Program:
  
  \[
  x := 5; \\
  y := -6; \\
  \text{if} \ (x+y) \ \text{then} \ z := x \ \text{else} \ z := y
  \]

- Semantics:

  \[
  \begin{array}{c}
  \langle x := 5, \ s_0 \rangle \Rightarrow s_1 \\
  \langle y := -6, \ s_1 \rangle \Rightarrow s_2 \\
  \langle x+y, s_2 \rangle \Rightarrow -1 \\
  \langle S_3, \ s_2 \rangle \Rightarrow s_3 \\
  \langle y := -6; S_3, \ s_1 \rangle \Rightarrow s_3 \\
  \langle x := 5; y := -6; S_3, \ s_0 \rangle \Rightarrow s_3
  \end{array}
  \]

  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
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 ⇒ require all identifier names to be unique.
History and References

- Main original papers:

- Many textbooks available.
Assertions and Correctness

- **Assertion:** a logical formula, say
  \[(m \neq 0 \land (\sqrt{m})^2 = m)\]
  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.

\[
\{PRE\} \text{ C } \{POST\} \leftarrow \text{a "Hoare triple"}
\]

- **Partial Correctness:**
  If the initial assertion (the precondition) is true and if the program terminates, then the final assertion (the postcondition) must be true.
  \[\text{Precondition + Termination } \Rightarrow \text{ Postcondition}\]

- **Total Correctness:**
  Given that the precondition for the program is true, the program must terminate and the postcondition must be true.
  \[\text{Total Correctness } = \text{ Partial Correctness + Termination}\]
Hoare Calculus: The Assignment Axiom

- Examples:
  - $\{true\} \ m := 13 \ \{m = 13\}$
  - $\{n = 3 \land c = 2\} \ n := c \cdot n \ \{n = 6 \land c = 2\}$
  - $\{k \geq 0\} \ k := k + 1 \ \{k > 0\}$

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

- Axiom for assignment command:

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

Work backwards:

- Postcondition: $P \equiv (n = 6 \land c = 2)$
- Command: $n := c \cdot n$
- Precondition: $P[V \rightarrow E] \equiv (c \cdot n = 6 \land c = 2)$
  $$\equiv (n = 3 \land c = 2)$$
**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 \(L\).
  - \(K, M, N, \ldots\) represent arbitrary numerals.

- **Axiom for read command:**
  - \(\{IN = [K|L] \land P[V \rightarrow K]\} \text{ read } V \{IN = L \land P\}\)

- **Axiom for write command:**
  - \(\{OUT = L \land E = K \land P\} \text{ write } E \{OUT = L :: [K] \land E = K \land P\}\)

- **Note:** \(L :: [K]\) is the list whose last element is \(K\) (\(::\) represents concatenation).
Hoare Calculus: Rules of Inference

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

  \[ \frac{H_1, H_2, H_n, ...}{H} \]

- **Axiom for Command Sequencing:**

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

- **Axioms for If Commands:**

  \[ \{P \land b\} C_1 \{Q\}, \quad \{P \land \neg b\} C_2 \{Q\} \quad \frac{}{\{P\} \text{ if } b \text{ then } C_1 \text{ else } C_2 \text{ endif } \{Q\}} \]

  \[ \{P \land b\} C \{Q\}, \quad (P \land \neg b) \rightarrow Q \quad \frac{}{\{P\} \text{ if } b \text{ then } C \text{ endif } \{Q\}} \]
Hoare Calculus: Rules of Inference (Contd.)

- **Weaken Postcondition:**
  \[
  \begin{align*}
  \{P\}C\{Q\}, \ Q & \rightarrow R \\
  \{P\}C\{R\}
  \end{align*}
  \]

- **Strengthen Precondition:**
  \[
  P \rightarrow Q, \ \{Q\}C\{R\} \\
  \{P\}C\{R\}
  \]

- **And and Or Rules:**
  \[
  \begin{align*}
  \{P\}C\{Q\}, \ \{P'\}C\{Q'\} \\
  \{P \land P'\}C\{Q \land Q'\}
  \end{align*}
  \]

  \[
  \begin{align*}
  \{P\}C\{Q\}, \ \{P'\}C\{Q'\} \\
  \{P \lor P'\}C\{Q \lor Q'\}
  \end{align*}
  \]

- **Observation:**
  \[
  \{ \text{false} \} \text{ any-command } \{ \text{any-postcondition} \}
  \]
Example (I)

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

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

Recall:
\{IN = [K|L] \land P[V \rightarrow K]\}
\textbf{read} \ V
\{IN = L \land P\}
Example (II)

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

read \( m \); read \( n \);
if \( m \geq n \) then
  \[ a := 2m \]
else
  \[ a := 2n \]
endif;
write a

So, \( b \equiv m \geq n = false \) and \( \neg b = true \); thus \( \{ P \land b \} = false \) and \( \{ P \land \neg b \} = P \).

So, for \( C_2 \) we have:
\[
\{ P \land \neg b \} = \{ P \} = \\
\{ IN = [16] \land OUT = [0, 1, 2] \land m = 4 \land n = 9 \} \rightarrow \\
\{ IN = [16] \land OUT = [0, 1, 2] \land m = 4 \land n = 9 \land 2 \times n = 18 \} \\
\[ a := 2n \] \\
\{ IN = [16] \land OUT = [0, 1, 2] \land m = 4 \land n = 9 \land a = 18 \} \\
\]
and for \( C_1 \) we can have anything since the premise is false:
\[
\{ P \land b \} = false \\
\[ a := 2m \] \\
\{ IN = [16] \land OUT = [0, 1, 2] \land m = 4 \land n = 9 \land a = 18 \} \\
\]
Example (III)

\{IN = [16] \land OUT = [0, 1, 2] \land m = 4 \land n = 9\} \leftarrow
\begin{array}{l}
\text{if } m \geq n \text{ then} \\
\quad a := 2 \times m \\
\text{else} \\
\quad a := 2 \times n
\end{array}
\text{endif};
\{IN = [16] \land OUT = [0, 1, 2] \land m = 4 \land n = 9 \land a = 18\}

and
\{IN = [16] \land OUT = [0, 1, 2] \land m = 4 \land n = 9 \land a = 18\}
\text{write a}
\{IN = [16] \land OUT = [0, 1, 2] :: [18] \land m = 4 \land n = 9 \land a = 18\}

which implies
\{IN = [16] \land OUT = [0, 1, 2, 18]\}
While Command

\[
\begin{align*}
\{P \land b\} C \{P\} \\
\{P\} \textbf{ while } b \textbf{ do } C \textbf{ endwhile } \{P \land \neg b\}
\end{align*}
\]

- **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 (\(\{P \land b\}\)).
  - *Completion:* show that the loop invariant and the exit condition produce the final assertion (\(\{P \land \neg b\}\)).

- **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 \land A \geq 0\}

\begin{align*}
    k &:= N; \quad s := 1; \\
    \textbf{while} \quad k > 0 \textbf{ do} \\
    &\quad s := A \cdot s; \\
    &\quad k := k - 1
\end{align*}

\textbf{ 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 $s \cdot 2^k = 32$ remains constant throughout the execution of the loop.
Example (Exponent)

\[ \{ N \geq 0 \land A \geq 0 \} \]

\[
\begin{align*}
 k & := N; & s & := 1; \\
\text{while } & k > 0 \text{ do} & \\
 & s & := A \cdot s; \\
 & k & := k - 1 \\
\text{endwhile}
\end{align*}
\]

\[ \{ s = A^N \} \]

\[
\begin{array}{cccc}
 k & s & 2^k & s \cdot 2^k \\
 5 & 1 & 32 & 32 \\
 4 & 2 & 16 & 32 \\
 3 & 4 & 8 & 32 \\
 2 & 8 & 4 & 32 \\
 1 & 16 & 2 & 32 \\
 0 & 32 & 1 & 32
\end{array}
\]

- Observe that \( s \) and \( 2^k \) change when \( k \) changes.
- Their product is constant, namely \( 32 = 2^5 = A^N \).
- This suggests that \( s \cdot A^k = A^N \) 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 \( s \cdot A^k = A^N \), we get the postcondition \( s = A^N \).

**Loop Invariant:** \( \{ k \geq 0 \land s \cdot A^k = A^N \} \).
Verification of the Program

Initialization:
\(\{N \geq 0 \land A \geq 0\} \rightarrow \{N = N \land N \geq 0 \land A \geq 0 \land 1 = 1\}\)
\[k := N; s := 1;\]
\(\{k = N \land N \geq 0 \land A \geq 0 \land s = 1\} \rightarrow \{k \geq 0 \land s \ast A^k = A^N\}\)

Preservation:
\(\{k \geq 0 \land s \ast A^k = A^N \land k > 0\} \rightarrow \{k > 0 \land s \ast A^k = A^N\} \rightarrow\)
\(\{k > 0 \land s \ast A \ast A^{k-1} = A^N\} \rightarrow \{k > 0 \land A \ast s \ast A^{k-1} = A^N\}\)
\[s := A \ast s;\]
\(\{k > 0 \land s \ast A^{k-1} = A^N\} \rightarrow \{k - 1 \geq 0 \land s \ast A^{k-1} = A^N\}\)
\[k := k - 1\]
\(\{k \geq 0 \land s \ast A^k = A^N\}\)

Completion:
\(\{k \geq 0 \land s \ast 2^k = A^N \land k \leq 0\} \rightarrow \{k = 0 \land s \ast 2^k = A^N\} \rightarrow \{s = A^N\}\)
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.