A Simple Programming Task

Program: Print the first 10 prime numbers.

(Test) Run: 2 3 5 7 11 13 17 19 (hang) OOPS!! 〈debug, debug〉

(Test) Run: 2 3 5 7 11 13 17 19 21 23 OOPS!! 〈debug, debug〉

(Test) Run: 2 3 5 7 11 13 17 19 23 31 OOPS!! 〈debug, debug〉

(Test) Run: 2 3 5 7 11 13 17 19 23 29 OK!

(Release software)

(User) Run: 2 3 5 7 11 13 17 19 23 29 OK!

A Less Simple Programming Task

Program: Print the first prime larger than a given number.

(Test) Run 38: 39 OOPS!! 〈debug, debug〉

(Test) Run 38: 43 OOPS!! 〈debug, debug〉

(Test) Run 41: 41 OK!

(Test) Run 44: 47 OK!

(Test) Run 62: 67 OK!

(Release software)

(User) Run 52: 57 OOPS!! 〈sue, sue〉

Grading

Your grade shall be calculated as follows:

- 25% from Assignments.
- 75% from Examination.

There will be 3 assignments, each of which must be submitted on time!

You should work in pairs (or, exceptionally, in triples) and submit a single set of solutions with all names at the top. Do not work in teams and then submit several copies of (nearly) identical solutions without recording this cooperation. Such plagiarism will incur stiff penalties.

Semantics and principles of programming languages

Course notes ("Kompendium") - buy it from UTHgårds.

Glynn Winskel: The Formal Semantics of Programming Languages.

(The Winskel book is a good reference, but you don't need it to pass the course.)
Three Styles of Semantics

- **Operational Semantics**
  - describes how a program is executed.
  - useful for compiler writers.

- **Denotational Semantics**
  - describes what a program computes.
  - useful for language designers.

- **Axiomatic Semantics**
  - describes the logical properties of a program.
  - useful for program writers.
  (It is the topic of its own course, so it is not covered in this course.)

---

A Taste of Axiomatic Semantics
(The art of loop invariants.)

Let \([P] C [Q] \) mean: "If property P holds now, then after executing C property Q will hold."

Then we have the following inference rules:

\[
\frac{[P] C_0 [Q] \quad [Q] C_1 [R]}{[P] (C_0 C_1) [R]}
\]

\[
\frac{[P] \mathbf{while} \ b \ \mathbf{do} \ c \quad [P \land \neg b]}{[P]}
\]

Such rules go together to give a method for proving programs correct (so-called Floyd-Hoare triples).

**Example:** Computing 10!

\[
\begin{align*}
  x &:= 1; y := 1; \\
  \{x \leq 10 \land y = x!\} \\
  \text{while } x < 10 \ \text{do} \ (x := x + 1; y := y \times x) \\
  \{x = 10 \land y = x!\}
\end{align*}
\]

--

Partially-Order Sets

The domain of "information" is a partially-ordered set (**poset**), that is, a set with a reflexive, transitive, antisymmetric relation \(\sqsubseteq\).

A computable function is a monotone function: "more information input yields more information output."

(Computer) programs denote such functions.

---

Mathematical Induction

If \(P(0)\) holds; and for any \(k \geq 0\), \(P(k+1)\) holds under the assumption that \(P(k)\) holds; then \(P(n)\) holds for every \(n \geq 0\).

**Example:** \(\sum_{i=0}^{n} i = \frac{n(n+1)}{2}\).

**Proof:** By induction on \(n\).

The equality holds when \(n = 0\), since

\[
\sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2}
\]

Suppose the equality holds for some \(k \geq 0\). Then

\[
\sum_{i=0}^{k+1} i = \sum_{i=0}^{k} i + (k+1) = \frac{k(k+1)}{2} + (k+1) \quad \text{(By induction)}
\]

\[
= \frac{k(k+1)+2(k+1)}{2} = \frac{(k+1)(k+2)}{2}.
\]

Hence the equality holds for \(k+1\); so it holds for all \(n \geq 0\).
**Strong Induction**

If \( P(n) \) holds under the assumption that \( P(k) \) holds for every \( k < n \); then \( P(n) \) holds for every \( n \geq 0 \).

**Example:** Let \( f(n) = \begin{cases} 0, & \text{if } n=0; \\ 2 \cdot f(n/2), & \text{if } n \geq 0 \text{ even}; \\ f(n-1) + 1, & \text{if } n \text{ odd}. \end{cases} \)

**Theorem:** \( f(n) = n \) for every \( n \geq 0 \).

**Proof:** By (strong) induction on \( n \), arguing by cases on the “structure” of \( n \).

\( n=0: \) \( f(0) = 0 \).

\( n>0 \text{ even:} \) \( f(n) = 2 \cdot f(n/2) = 2 \cdot (n/2) = n \) (By induction)

\( n \text{ odd:} \) \( f(n) = f(n-1) + 1 = (n-1) + 1 = n \) (By induction)

**Induction over Words**

Defining words (strings) over alphabet \([a, b] \):

\( \{ s \mid s = \varepsilon | as | bs \} \)

That is, the set of words \( [a, b]^* \) is defined to be the least set containing the empty word \( \varepsilon \), and such that whenever \( s \) is in the set, then so are \( as \) and \( bs \).

The length of a word is inductively defined by:

\[
\begin{align*}
\text{length}(\varepsilon) & = 0 \\
\text{length}(as) & = 1 + \text{length}(s) \\
\text{length}(bs) & = 1 + \text{length}(s)
\end{align*}
\]

**Theorem:** \( aw \neq wb \) for all words \( w \).

**Proof:** By induction on the length of \( w \), arguing by cases on the structure of \( w \).

\( w \equiv \varepsilon: \) \( aw = a \neq b = wb \).

\( w \equiv as: \) By induction, \( as \neq sb \), so \( asw \neq asb \).

But then \( aw = asw \neq asb = wb \).

\( w \equiv bs: \) \( aw = abs \neq sbw = wb \).

**Induction over Trees**

Defining binary trees:

\[
\{ t \mid t = L | N(t_1, t_2) \}
\]

That is, the set of binary trees is defined to be the least set containing the leaf \( L \), and such that whenever \( t_1 \) and \( t_2 \) are in the set, then so is \( N(t_1, t_2) \).

The depth of a tree is inductively defined by:

\[
\begin{align*}
\text{depth}(L) & = 0 \\
\text{depth}(N(t_1, t_2)) & = 1 + \left( \max(\text{depth}(t_1), \text{depth}(t_2)) \right)
\end{align*}
\]

**Theorem:** Every tree \( t \) has exactly one more leaf than (internal) node.

**Proof:** By induction on the structure of \( t \).

\( t \equiv L: \) \( L \) has 1 leaf and 0 nodes.

\( t \equiv N(t_1, t_2): \) By induction, \( t_i \) (\( i = 1, 2 \)) must have \( n_i \) nodes and \( n_i + 1 \) leaves, for some \( n_1, n_2 \). But then \( N(t_1, t_2) \) must have \( n_1 + n_2 + 1 \) nodes and \((n_1 + 1) + (n_2 + 1) = (n_1 + n_2 + 1) + 1 \) leaves.

**Induction via Contradiction**

**Theorem:** For all \( n \geq 0 \), \( \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \)

**Proof:** By contradiction.

Suppose that for some \( n \geq 0 \), \( \sum_{i=0}^{n} i \neq \frac{n(n+1)}{2} \).

Let \( n \) be the least number for which the theorem fails to hold; that is, suppose the theorem is true for all values less than \( n \).

We argue by cases on the “structure” of \( n \):

\( n=0: \) \( 0 + 1 + 2 + \cdots + 0 = 0 = \frac{0(0+1)}{2} = \frac{n(n+1)}{2} \) \( \text{contradiction!} \)

\( n = k+1: \)

\[
0 + 1 + 2 + \cdots + n = \sum_{i=0}^{n} i = \sum_{i=0}^{k} i + (k+1)
\]

\[
= \sum_{i=0}^{k} i + (k+1) \quad \text{(by assumption)}
\]

\[
= \frac{k(k+1)}{2} + (k+1) = \frac{n(n+1)}{2}
\]

\( \text{contradiction!} \)

\( \text{A-9} \)
The Syntax of the Language Imp

Syntactic Domains

\[ n, m \in \mathbb{N} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \] numbers (integers)

\[ t \in \mathcal{B} = \{\text{true, false}\} \] truth values (booleans)

\[ x, y \in \text{Var} = \{x, y, \ldots\} \] variables

\[ a \in \text{Aexp} \] arithmetic (integer) expressions

\[ b \in \text{Bexp} \] boolean expressions

\[ c \in \text{Com} \] commands

The Language

<table>
<thead>
<tr>
<th>( a \rightarrow n )</th>
<th>( b \rightarrow \text{true} )</th>
<th>( c \rightarrow \text{skip} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>false</td>
<td>( x := a )</td>
</tr>
<tr>
<td>( a_0 + a_1 )</td>
<td>( a_0 \cdot a_1 )</td>
<td>( a_0 \cdot a_1 )</td>
</tr>
<tr>
<td>( a_0 - a_1 )</td>
<td>( a_0 &lt; a_1 )</td>
<td>( \text{if } b \text{ then } c_0 \text{ else } c_1 )</td>
</tr>
<tr>
<td>( \neg b )</td>
<td>( b_0 \text{ and } b_1 )</td>
<td>( b_0 \text{ or } b_1 )</td>
</tr>
</tbody>
</table>

Example: Computing \( x! \)

\[ y := 1; \text{ while } 1 < x \text{ do } (y := y \cdot x; x := x - 1) \]

Syntactic Decompositions

The command (program)

\[ y := 1; \text{ while } 1 < x \text{ do } (y := y \cdot x; x := x - 1) \]

breaks down into the following parse tree.

---

Operational Semantics of Imp

Evaluating arithmetic expressions

A state \( \sigma \in \Sigma \) is a mapping from variables to integers:

\[ \Sigma \overset{\text{def}}{=} \text{Var} \rightarrow \mathbb{Z} \]

Evaluating arithmetic expressions:

\[ \langle a_0, \sigma \rangle \rightarrow n \]

“Arithmetic expression \( a_0 \in \text{Aexp} \) in state \( \sigma \in \Sigma \) evaluates to the integer value \( n \in \mathbb{Z} \).”

Evaluation rules

\[ \langle n, \sigma \rangle \rightarrow n \]

\[ \langle x, \sigma \rangle \rightarrow \sigma(x) \]

\[ \langle a_0 + a_1, \sigma \rangle \rightarrow n_0 + n_1 \]

\[ \langle a_0 - a_1, \sigma \rangle \rightarrow n_0 - n_1 \]

\[ \langle a_0 \cdot a_1, \sigma \rangle \rightarrow n_0 \cdot n_1 \]

\[ \langle a_0 \cdot a_1, \sigma \rangle \rightarrow n_0 \cdot n_1 \]

Operational Semantics of Imp

Evaluating boolean expressions

Evaluation rules:

\[ \langle \text{true}, \sigma \rangle \rightarrow \text{true} \]

\[ \langle \text{false}, \sigma \rangle \rightarrow \text{false} \]

\[ \langle \text{not} b, \sigma \rangle \rightarrow \neg t \]

\[ \langle b_0 \text{ and } b_1, \sigma \rangle \rightarrow t_0 \land t_1 \]

\[ \langle b_0 \text{ or } b_1, \sigma \rangle \rightarrow t_0 \lor t_1 \]

\[ \langle a_0, \sigma \rangle \rightarrow n_0 \]

\[ \langle a_0 = a_1, \sigma \rangle \rightarrow \text{true} \]

\[ \langle a_0 \neq a_1, \sigma \rangle \rightarrow \text{false} \]

\[ \langle a_0 < a_1, \sigma \rangle \rightarrow \text{false} \]

\[ \langle a_0 > a_1, \sigma \rangle \rightarrow \text{false} \]

\[ \langle a_0 \leq a_1, \sigma \rangle \rightarrow \text{true} \]

\[ \langle a_0 \geq a_1, \sigma \rangle \rightarrow \text{false} \]
Example Evaluation Inferences

**Example:** Evaluating \((x-6)*(3+y)\)
in a state \(\sigma\) where \(\sigma(x) = 9\) and \(\sigma(y) = 4\):

\[
\langle x, \sigma \rangle \rightarrow 9 \quad \langle 6, \sigma \rangle \rightarrow 6 \quad \langle 3, \sigma \rangle \rightarrow 3 \quad \langle y, \sigma \rangle \rightarrow 4 \\
\langle (x-6)*(3+y), \sigma \rangle \rightarrow 2
\]

**Example:** Evaluating \(x-3 < y\)
in a state \(\sigma\) where \(\sigma(x) = 9\) and \(\sigma(y) = 4\):

\[
\langle x, \sigma \rangle \rightarrow 9 \quad \langle 3, \sigma \rangle \rightarrow 3 \quad \langle y, \sigma \rangle \rightarrow 4 \\
\langle (x-3)*y, \sigma \rangle \rightarrow false
\]

Structural Induction

**Definition:**

\[
\begin{align*}
\text{Var}(\emptyset) &= \emptyset \\
\text{Var}(x) &= \{x\} \\
\text{Var}(\alpha + \beta) &= \text{Var}(\alpha) \cup \text{Var}(\beta) \\
\text{Var}(\alpha - \beta) &= \text{Var}(\alpha) \cup \text{Var}(\beta) \\
\text{Var}(\alpha * \beta) &= \text{Var}(\alpha) \cup \text{Var}(\beta)
\end{align*}
\]

**Theorem:** For all \(\alpha \in \text{Aexp}\) and all \(\sigma, \sigma' \in \Sigma\), if

- \(\sigma(x) = \sigma'(x)\) for every \(x \in \text{Var}(\alpha)\); and
- \(\langle \alpha, \sigma \rangle \rightarrow m\) and \(\langle \alpha, \sigma' \rangle \rightarrow m'\);

then \(m = m'\).

**Proof:** By induction on the structure of \(\alpha \in \text{Aexp}\).

Operational Semantics of Imp

Executing commands

**Evaluation rules:**

\[
\begin{align*}
\text{skip}, \sigma &\rightarrow \sigma' \\
\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{while } b \text{ do } c, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{true}, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{false}, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

Example Execution Inference

Executing \(\text{while } 1 < x \text{ do } (y := x*y; x := x-1)\)
in a state \(\sigma\) where \(\sigma(x) = 3\) and \(\sigma(y) = 1\):

Let:

- \(b \equiv 1 < x\)
- \(c \equiv y := x*y; x := x-1\)

\[
\begin{align*}
\langle 1, \sigma \rangle \rightarrow 1 \quad \langle 5, \sigma \rangle \rightarrow 3 & \quad \langle 3, \sigma \rangle \rightarrow 3 \quad \langle 1, \sigma \rangle \rightarrow 1 \\
\langle b, \sigma \rangle \rightarrow \text{true} & \quad \langle c, \sigma \rangle \rightarrow \sigma' \\
\langle b, \sigma \rangle \rightarrow \text{false} & \quad \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma'
\end{align*}
\]

**Example:** Evaluating \((x-6)*(3+y)\)
in a state \(\sigma\) where \(\sigma(x) = 9\) and \(\sigma(y) = 4\):

\[
\langle x, \sigma \rangle \rightarrow 9 \quad \langle 6, \sigma \rangle \rightarrow 6 \quad \langle 3, \sigma \rangle \rightarrow 3 \quad \langle y, \sigma \rangle \rightarrow 4 \\
\langle (x-6)*(3+y), \sigma \rangle \rightarrow 2
\]

**Example:** Evaluating \(x-3 < y\)
in a state \(\sigma\) where \(\sigma(x) = 9\) and \(\sigma(y) = 4\):

\[
\langle x, \sigma \rangle \rightarrow 9 \quad \langle 3, \sigma \rangle \rightarrow 3 \quad \langle y, \sigma \rangle \rightarrow 4 \\
\langle (x-3)*y, \sigma \rangle \rightarrow false
\]
**Induction on Depth of Inference**

**Definition:**

\[
\begin{array}{c}
\text{Var}[\text{skip}] = \emptyset \\
\text{Var}(x := a) = \{x\} \\
\text{Var}(c_0 \text{;} c_1) = \text{Var}(c_0) \cup \text{Var}(c_1) \\
\text{Var}(\text{if } b \text{ then } c_0 \text{ else } c_1) = \text{Var}(c_0) \cup \text{Var}(c_1) \\
\text{Var}(\text{while } b \text{ do } c) = \text{Var}(c)
\end{array}
\]

**Theorem:** For all \( c \in \text{Com} \), if \( y \not\in \text{Var}_i(c) \) and \( \langle c, \sigma \rangle \rightarrow \sigma' \) then \( \sigma(y) = \sigma'(y) \).

**Proof:** By induction on the depth of inference of \( \langle c, \sigma \rangle \rightarrow \sigma' \). Suppose we have an inference of \( \langle c, \sigma \rangle \rightarrow \sigma' \), and that for every \( \langle c_0, \sigma_0 \rangle \rightarrow \sigma'_0 \) with a shorter inference such that \( y \not\in \text{Var}_i(c_0) \) we have that \( \sigma_0(y) = \sigma'_0(y) \).

We shall demonstrate that \( \sigma(y) = \sigma'(y) \), arguing by cases on the structure of \( c \in \text{Com} \).

1. \( c \equiv \text{skip}; \ldots \)
2. \( c \equiv x := a; \ldots \)
3. \( c \equiv c_0 \text{;} c_1 \ldots \)
4. \( c \equiv \text{if } b \text{ then } c_0 \text{ else } c_1; \ldots \)
5. \( c \equiv \text{while } b \text{ do } c; \ldots \)

**Totality**

**Theorem:** The transition system for arithmetic expressions is total: if \( \langle a, \sigma \rangle \rightarrow m_0 \) and \( \langle a, \sigma \rangle \rightarrow m_1 \) then \( m_0 = m_1 \).

**Proof:** By induction on the structure of \( a \in \text{Aexp} \).

**Theorem:** The transition system for boolean expressions is total: if \( \langle b, \sigma \rangle \rightarrow t_0 \) and \( \langle b, \sigma \rangle \rightarrow t_1 \) then \( t_0 = t_1 \).

**Proof:** By induction on the structure of \( b \in \text{Bexp} \).

**Theorem:** The transition system for commands is total: if \( \langle c, \sigma \rangle \rightarrow \sigma_0 \) and \( \langle c, \sigma \rangle \rightarrow \sigma_1 \) then \( \sigma_0 = \sigma_1 \).

**Proof:** By induction on the depth of inference of \( \langle c, \sigma \rangle \rightarrow \sigma' \), arguing by cases on the structure of \( c \in \text{Com} \).

**Determinacy**

**Theorem:** The transition system for arithmetic expressions is deterministic: if \( \langle a, \sigma \rangle \rightarrow m_0 \) and \( \langle a, \sigma \rangle \rightarrow m_1 \) then \( m_0 = m_1 \).

**Proof:** By induction on the structure of \( a \in \text{Aexp} \).

**Theorem:** The transition system for boolean expressions is deterministic: if \( \langle b, \sigma \rangle \rightarrow t_0 \) and \( \langle b, \sigma \rangle \rightarrow t_1 \) then \( t_0 = t_1 \).

**Proof:** By induction on the structure of \( b \in \text{Bexp} \).

**Theorem:** The transition system for commands is deterministic: if \( \langle c, \sigma \rangle \rightarrow \sigma_0 \) and \( \langle c, \sigma \rangle \rightarrow \sigma_1 \) then \( \sigma_0 = \sigma_1 \).

**Proof:** By induction on the depth of inference of \( \langle c, \sigma \rangle \rightarrow \sigma' \), arguing by cases on the structure of \( c \in \text{Com} \).

**Theorem:** If \( \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma' \) then \( \langle b, \sigma' \rangle \rightarrow \text{false} \).

**Proof:** By induction on the depth of inference of \( \langle b, \sigma' \rangle \rightarrow \text{false} \).

In order to deduce that \( \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma' \), we must use either the rule

\[
\langle b, \sigma \rangle \rightarrow \text{false} \\
\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma'
\]

in which case \( \sigma' = \sigma \) and the result follows;

or the rule

\[
\langle b, \sigma \rangle \rightarrow \text{true} \\
\langle \text{while } b \text{ do } c, \sigma' \rangle \rightarrow \sigma' \\
\langle \text{while } b \text{ do } c, \sigma' \rangle \rightarrow \sigma''
\]

in which case the result follows by induction, as \( \langle b, \sigma' \rangle \rightarrow \sigma'' \) must have a shorter inference.