CMPS203 Lecture 3

Computer Science 203
Programming Languages
Fall 2004 - Lecture 3

Cormac Flanagan
University of California, Santa Cruz

What is the relation \( \rightarrow \) defined by these rules?

\[
\begin{align*}
n & \text{ is the sum of } n_1 \text{ and } n_2 \\
& n_1 + n_2 \rightarrow n \\
e_1 & \rightarrow e_1' \\
e_1 + e_2 & \rightarrow e_1' + e_2 \\
e_1 & \rightarrow e_1' \\
e_1 * e_2 & \rightarrow e_1' * e_2 \\
\end{align*}
\]

\[
\begin{align*}
n & \text{ is the product of } n_1 \text{ and } n_2 \\
n_1 * n_2 & \rightarrow n \\
e_1 & \rightarrow e_1' \\
e_2 & \rightarrow e_2' \\
\end{align*}
\]

Small-Step Evaluation Rules

- Fixed evaluation order.
- Example: \((3 + 4) + 5 \rightarrow 7 + 5 \rightarrow 12\)

An Imperative Language: IMP

syntax and operational semantics

a little more interesting

IMP Syntactic Entities

- int\quad \text{integer literals}
  - n
- bool\quad \text{booleans}
  - p \in \{\text{true, false}\}
- L\quad \text{locations (assignable variables)}
  - x, y, ...
- Aexp\quad \text{arithmetic expressions}
  - e
- Bexp\quad \text{boolean expressions}
  - b
- Comm\quad \text{commands}
  - c

Abstract Syntax (Aexp)

- For arithmetic expressions (Aexp)
  \[
  e ::= \quad n \quad \text{for } n \in \mathbb{Z} \\
| x \quad \text{for } x \in L \\
| e_1 + e_2 \quad \text{for } e_1, e_2 \in Aexp \\
| e_1 - e_2 \quad \text{for } e_1, e_2 \in Aexp \\
| e_1 * e_2 \quad \text{for } e_1, e_2 \in Aexp
  \]

- Notes:
  - Variables are not declared.
  - All variables have integer type.
  - There are no side-effects.
Abstract Syntax (Bexp)

- For boolean expressions (Bexp)
  \[ b ::= \begin{align*}
  & \text{true} \\
  & \text{false} \\
  & e_1 = e_2 \quad \text{for } e_1, e_2 \in Aexp \\
  & e_1 \leq e_2 \quad \text{for } e_1, e_2 \in Aexp \\
  & \neg b \quad \text{for } b \in Bexp \\
  & b_1 \land b_2 \quad \text{for } b_1, b_2 \in Bexp \\
  & b_1 \lor b_2 \quad \text{for } b_1, b_2 \in Bexp
  \end{align*} \]

Abstract Syntax (Comm)

- For commands (Comm)
  \[ c ::= \begin{align*}
  & \text{skip} \\
  & x := e \quad \text{for } x \in L \text{ and } e \in Aexp \\
  & c_1 ; c_2 \quad \text{for } c_1, c_2 \in \text{Comm} \\
  & \text{if } b \text{ then } c_1 \text{ else } c_2 \quad \text{for } c_1, c_2 \in \text{Comm \ and } b \in \text{Bexp} \\
  & \text{while } b \text{ do } c \quad \text{for } c \in \text{Comm \ and } b \in \text{Bexp}
  \end{align*} \]

Notes:
- The typing rules have been embedded in the syntax definition.
- Other checks may not be context-free and need to be specified separately (e.g., all variables are declared).
- Commands contain all the side-effects in the language.
- (Missing: pointers, function calls, ….)

Semantics of IMP

- The meaning of IMP expressions depends on the values of variables.
- A state \( \sigma \) is a function from \( L \) to \( Z \)
  - Represents the value of variables at a given moment
  - The set of all states is \( \Sigma = L \rightarrow Z \).

Evaluation Rules (for Aexp)

\[
\begin{align*}
& <n, \sigma> \cup n \\
& <x, \sigma> \cup \alpha(x) \\
& <e_1 + e_2, \sigma> \cup n_1 + n_2 \\
& <e_1 - e_2, \sigma> \cup n_1 - n_2 \\
& <e_1 \ast e_2, \sigma> \cup n_1 \ast n_2 \\
& <e_1 \land e_2, \sigma> \cup n_1 \land n_2 \\
& <e_1 \lor e_2, \sigma> \cup n_1 \lor n_2 \\
& <e_1 \leq e_2, \sigma> \cup n_1 \leq n_2 \\
& <e_1 = e_2, \sigma> \cup n_1 = n_2
\end{align*}
\]

Evaluation Rules (for Bexp)

\[
\begin{align*}
& <\text{true}, \sigma> \cup \text{true} \\
& <\text{false}, \sigma> \cup \text{false} \\
& <e_1, \sigma> \cup n_1 \\
& <e_2, \sigma> \cup n_2 \\
& <e_1 \leq e_2, \sigma> \cup n_1 \leq n_2 \\
& <e_1 = e_2, \sigma> \cup n_1 = n_2 \\
& <b_1 \land b_2, \sigma> \cup \text{false} \\
& <b_1 \lor b_2, \sigma> \cup \text{false} \\
& <b_1, \sigma> \cup \text{true} \\
& <b_1 \land b_2, \sigma> \cup \text{true} \\
& <b_1 \lor b_2, \sigma> \cup \text{true}
\end{align*}
\]

Operational Semantics of IMP

- Evaluation judgment for expressions
  - A ternary relation: on an expression, a state, and a value.
  - We write: \( <e, \sigma> \downarrow n \)
  - The evaluation of expressions does not have side-effects, so no resulting state on the right
  - In this case we can also view this judgment as a function of two arguments (written to the left of \( \downarrow \))

- Evaluation judgement for commands
  - A ternary relation: on an expression, a state, and a new state.
  - Evaluation of a command has side effects but no direct result
  - The "result" of a Comm is a new state: \( <c, \sigma> \downarrow \sigma' \)
  - The evaluation of a command might not terminate.
Evaluation Rules (for Comm)

\[
\begin{align*}
\langle e, \sigma \rangle \uparrow n & \quad \text{Def: } \sigma(x := n)(x) = n, \quad \sigma(x := n)(y) = \sigma(y) \\
\langle x := e, \sigma \rangle \uparrow \sigma[x := n] & \\
\langle \text{skip}, \sigma \rangle \uparrow \sigma & \\
\langle b, \sigma \rangle \uparrow \text{true} & \quad \langle c_1 ; c_2, \sigma \rangle \uparrow \sigma' \\
\langle b, \sigma \rangle \uparrow \text{false} & \quad \langle c_1, \sigma \rangle \uparrow \sigma' \\
\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \uparrow \sigma' & \\
\langle \text{while } b \text{ do } c, \sigma \rangle \uparrow \sigma' & \\
\langle b, \sigma \rangle \uparrow \text{true} & \quad \langle c_1, \sigma \rangle \uparrow \sigma' \\
\langle b, \sigma \rangle \uparrow \text{false} & \quad \langle c_2, \sigma \rangle \uparrow \sigma' \\
\end{align*}
\]

Evaluation of Commands: Notes

- The order of evaluation is important and explicit.
  - $c_1$ is evaluated before $c_2$ in $c_1; c_2$.
  - $c_2$ is not evaluated in "if true then $c_1$ else $c_2$".
  - $c$ is not evaluated in "while false do $c$".
  - $b$ is evaluated first in "if $b$ then $c_1$ else $c_2$".
- The evaluation rules are not syntax-directed.
  - See the rule for while.
- The evaluation rules do suggest an interpreter.
- Conditional constructs have multiple evaluation rules, but only one can be applied at one time.

Disadvantages of Natural Operational Semantics

- Natural (or big-step) operational semantics has two disadvantages:
  - It is hard to talk about commands whose evaluation does not terminate.
    - There is no $\sigma'$ such that $\langle c, \sigma \rangle \downarrow \sigma'$.
    - But we do not have an explanation of how $c$ runs or fails.
  - It does not give us a way to talk about intermediate states:
    - Thus we cannot say that on a parallel machine the execution of two commands is interleaved.
- Small-step semantics overcomes these limitations.
  Execution is modeled as a sequence of states (possible infinite).

Contextual Semantics

- Contextual semantics is a small-step semantics where the atomic execution step is a rewrite of the program.
- We will define a relation $\langle c, \sigma \rightarrow c', \sigma' \rangle$:
  - $c'$ is obtained from $c$ through an atomic rewrite step.
  - E.g.: $\langle x := 2 + 8, \sigma \rightarrow x := 10, \sigma \rightarrow \text{skip}, \sigma[x := 10] \rangle$.
  - Evaluation terminates when the program has been rewritten to a terminal program (one from which we cannot make further progress).
- For IMP the terminal command is "skip".
- As long as the command is not "skip" we can make progress.
- Some commands never reduce to skip (e.g., while true do skip).

What is an Atomic Reduction?

- We need to define:
  - What constitutes an atomic reduction step?
    - Granularity is a choice of the semantics designer.
      - E.g., choice between an addition of arbitrary integers, or an addition of 32-bit integers.
    - How to select the next reduction step, when several are possible?
      - This is the order of evaluation issue.

Redexes

- A redex is a syntactic expression or command that can be reduced (transformed) in one atomic step.
- For brevity, we mix expression and command redexes (and also omit some redexes and contexts).
- Redexes are defined by a grammar:
  \[
  r ::= x \\
  | n_1 + n_2 \\
  | x := n \\
  | \text{skip} \\
  | \text{if } b \text{ then } c_1 \text{ else } c_2 \\
  | \text{while } b \text{ do } c \\
  \]
- Note that $(1 + 3) * 2$ is not a redex, but $1 + 3$ is.
### Local Reduction Rules for IMP

- One for each redex: \( \langle r, \sigma \rangle \rightarrow \langle e, \sigma' \rangle \)
  - This means that in state \( \sigma \), the redex \( r \) can be replaced in one step with the expression \( e \).

<table>
<thead>
<tr>
<th>Redex Form</th>
<th>Rule</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle )</td>
<td></td>
<td></td>
<td>where ( n = n_1 + n_2 )</td>
</tr>
<tr>
<td>( \langle n_1 + n_2, \sigma \rangle \rightarrow \langle n, \sigma \rangle )</td>
<td></td>
<td></td>
<td>if ( n_1 = n_2 )</td>
</tr>
<tr>
<td>( \langle x := n, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[x := n] \rangle )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \langle \text{skip}; c, \sigma \rangle \rightarrow \langle c, \sigma \rangle )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \langle \text{if} \ true \ \text{then} \ c_1 \ \text{else} \ c_2, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \langle \text{if} \ false \ \text{then} \ c_1 \ \text{else} \ c_2, \sigma \rangle \rightarrow \langle c_2, \sigma \rangle )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \langle \text{while} \ b \ \text{do} \ c, \sigma \rangle \rightarrow \langle \text{if} \ b \ \text{then} \ c \ \text{while} \ b \ \text{do} \ c \ \text{else} \ \text{skip}, \sigma \rangle )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Cheating

- All work you turn in must be your own
- If you don’t know if something is allowed, please ask
- Any cheating will result in failure of the course and other standard measures

- You are encouraged to discuss course material and assignments with others
- You are not allowed do homeworks with others
- You may use any conversations, texts, or other material, as long as you cite your sources

### Reading

- For Tuesday, read up on induction
- Winskel’s book: chapters 3 and 4
  - can read ahead on chapter 6 if you have time
- Also read a small proof by induction, by Faron Moller, available on the course web pages.
- Please read and use the class newsgroup:
  - ucsc.class.cmps203 on news server news.ucsc.edu