Operational Semantics: Big-Step vs. Small-Step

- **Big-Step Operational Semantics:** $e \Downarrow n$
  - Judgment $e \Downarrow n$ means that $e$ evaluates to $n$
    - In one, big step, all the way to a result
  - Hard to talk about commands that do not terminate.
    - There is no $\sigma'$ such that $<c, \sigma> \Downarrow \sigma'$
    - 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 Operational Semantics:** $e \rightarrow e'$
  - describe a single step in the evaluation
  - many steps may be needed to get a result
What is the relation $\rightarrow$ defined by these rules?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1 + n_2 \rightarrow n$</td>
<td>$e_1 \rightarrow e_1'$</td>
</tr>
<tr>
<td>$e_1 + e_2 \rightarrow e_1' + e_2$</td>
<td>$e_2 \rightarrow e_2'$</td>
</tr>
<tr>
<td>$e_1 \rightarrow e_1'$</td>
<td>$n_1 + e_2 \rightarrow n_1 + e_2'$</td>
</tr>
<tr>
<td>$e_1 * e_2 \rightarrow e_1' * e_2$</td>
<td>$e_2 \rightarrow e_2'$</td>
</tr>
<tr>
<td>$n_1 * e_2 \rightarrow n_1 * e_2'$</td>
<td></td>
</tr>
</tbody>
</table>

$n$ is the sum of $n_1$ and $n_2$

$n$ is the product of $n_1$ and $n_2$
Small-Step Evaluation Rules

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

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

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

- Contextual semantics is a small-step semantics that is specified in two parts:
  
  - What evaluation rules to apply?
    - What is an atomic reduction step?

- Where can we apply them?
  - Where should we apply the next atomic reduction step?
Small-Step Operational Semantics for IMP

• Each execution step is a rewrite of the program.
• We will define a relation \( <c, \sigma> \rightarrow <c', \sigma'> \)
  - \( c' \) is obtained from \( c \) through an atomic rewrite step.
  - E.g.: \( <x := 2+8, \sigma> \rightarrow <x := 10, \sigma> \rightarrow <\text{skip}, \sigma[x:=10]> \)
  - 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:
  
  \[ r ::= \]
  
  \[ x \]
  
  \[ | n_1 + n_2 \]
  
  \[ | x := n \]
  
  \[ | \text{skip;} c \]
  
  \[ | \text{if } \text{true then } c_1 \text{ else } c_2 \]
  
  \[ | \text{if } \text{false then } c_1 \text{ else } c_2 \]
  
  \[ | \text{while } b \text{ do } c \]
  
• Note that \((1 + 3) + 2\) is not a redex, but \(1 + 3\) is.
Local Reduction Rules for IMP

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

\(<x, \sigma> \rightarrow <\sigma(x), \sigma>\)
\(<n_1 + n_2, \sigma> \rightarrow <n, \sigma>\)  \(\text{where } n = n_1 + n_2\)
\(<n_1 = n_2, \sigma> \rightarrow <\text{true}, \sigma>\)  \(\text{if } n_1 = n_2\)
\(<x := n, \sigma> \rightarrow <\text{skip}, \sigma[x := n]>\)
\(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\)
\(<\text{if true then } c_1 \text{ else } c_2, \sigma> \rightarrow <c_1, \sigma>\)
\(<\text{if false then } c_1 \text{ else } c_2, \sigma> \rightarrow <c_2, \sigma>\)
\(<\text{while } b \text{ do } c, \sigma> \rightarrow <\text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else } \text{skip}, \sigma>\)
Review

- A redex is something that can be reduced in one step
  - E.g. $2+8$
- **Local reduction rules** reduce these redexes
  - E.g. $<2+8, \sigma> \rightarrow <10, \sigma>$

- **Next**: global reduction rules
- Consider
  - $<x := 1+(2+8), \sigma>$
  - $<\text{while false do } x := 1+(2+8), \sigma>$
- Should we also reduce $2+8$ in these cases?
Contexts

• A context is an expression or command with exactly one marker “•”
  - The marker is sometimes called a hole.
  - H[e] is obtained from H by replacing the marker • with e

• Examples
  - x := 1+•
    • Fill context H with 2+8 to yield H[2+8] = x := 1+(2+8)
    • Or fill context with 10 to yield H[10] = x := 1+10
  - while false do x := 1+•
    • Fill with 2+8 to yield H[2+8] = while false do x := 1+(2+8)
  - while false do •
  - •
Evaluation Contexts

- An **evaluation context** is a context in which the marker indicates the next place for evaluation.
  - identifies the next redex, a bit like a program counter

  \[
  H ::= \bullet \\
  \mid H + e \\
  \mid n + H \\
  \mid x := H \\
  \mid \text{if } H \text{ then } c_1 \text{ else } c_2 \\
  \mid H; c
  \]
Evaluation Contexts

- An evaluation context is a context in which the marker indicates the next place for evaluation.
  - identifies the next redex, a bit like a program counter
    \[ H ::= \]
    \[ | H + e \]
    \[ | n + H \]
    \[ | x := H \]
    \[ | \text{if } H \text{ then } c_1 \text{ else } c_2 \]
    \[ | H; c \]

- Examples
  - \( x := 1+\cdot \)
  - \( \cdot \)
  - NOT: while false do \( x := 1+\cdot \)
  - NOT: if \( b \) then \( c \) else \( \cdot \)
Contexts: Notes

• Evaluation contexts say how to find the next redex:
  - Consider $e_1 + e_2$ and its decomposition as $H[r]$.
  - If $e_1$ is $n_1$ and $e_2$ is $n_2$
    • then $H = \cdot$ and $r = n_1 + n_2$.
  - If $e_1$ is $n_1$ and $e_2$ is not $n_2$
    • then $H = n_1 + H_2$ and $e_2 = H_2[r]$.
  - If $e_1$ is not $n_1$
    • then $H = H_1 + e_2$ and $e_1 = H_1[r]$.
  - In the last two cases the decomposition is done recursively.
  - In each case the solution is unique.
The Global Reduction Rule

- **General idea of the contextual semantics:**
  - Decompose the current expression into
    - the next redex r
    - and an evaluation context H (the remaining program).
  - Reduce the redex “r” to some other expression “e”.
  - Put “e” back into the original context, yielding H[e].

- **Formalized as a small step rule:**

  If \(<r, \sigma> \rightarrow <e, \sigma'>\) then \(<H[r], \sigma> \rightarrow <H[e], \sigma'>\)
The Global Reduction Rule: Example

• Consider the command \( x := 1+(2+8) \)
• Split into an evaluation context \( H \) and a redex \( r \)
• Get
  \[
  \begin{align*}
  H &= x := 1+ \\
  r &= 2+8 \\
  H[r] &= x := 1+(2+8) \quad \text{(original command)}
  \end{align*}
  \]
• Have
  \[
  \begin{align*}
  &\langle 2+8, \sigma \rangle \rightarrow \langle 10, \sigma \rangle \quad \text{(local reduction rule)}
  \end{align*}
  \]
• Define global reduction
  \[
  \begin{align*}
  &\langle H[2+8], \sigma \rangle \rightarrow \langle H[10], \sigma \rangle \\
  &\langle x := 1+(2+8), \sigma \rangle \rightarrow \langle x := 1+10, \sigma \rangle \\
  &\text{or, equivalently}
  \end{align*}
  \]
**Contextual Semantics: Example**

- Consider the small-step evaluation of \( x := 1; x := x + 1 \) in the initial state \([x := 0]\)

<table>
<thead>
<tr>
<th>State</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x := 1; x := x + 1, [x := 0])&gt;</td>
<td>•; ( x := x + 1 )</td>
<td>( x := 1 )</td>
</tr>
<tr>
<td>(&lt;\text{skip}; x := x + 1, [x := 1])&gt;</td>
<td>•</td>
<td>( \text{skip}; x := x + 1 )</td>
</tr>
<tr>
<td>(&lt;x := x + 1, [x := 1])&gt;</td>
<td>( x := \bullet + 1 )</td>
<td>( x )</td>
</tr>
<tr>
<td>(&lt;x := 1 + 1, [x := 1])&gt;</td>
<td>( x := \bullet )</td>
<td>( 1 + 1 )</td>
</tr>
<tr>
<td>(&lt;x := 2, [x := 1])&gt;</td>
<td>•</td>
<td>( x := 2 )</td>
</tr>
<tr>
<td>(&lt;\text{skip}, [x := 2])&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Normal vs Short-Circuit Boolean Operators

• What if we want normal evaluation of $\land$?
  - Define the following contexts, redexes, and local rules:
    
    $H ::= \ldots | H \land b_2 | p_1 \land H$
    
    $r ::= \ldots | p_1 \land p_2$
    
    $\langle p_1 \land p_2, \sigma \rangle \rightarrow \langle p, \sigma \rangle \quad$ where $p = p_1 \land p_2$
Normal vs Short-Circuit Boolean Operators

• What if we want normal evaluation of $\land$?
  - Define the following contexts, redexes, and local rules:
    \[
    H ::= \ldots | H \land b_2 | p_1 \land H \\
    r ::= \ldots | p_1 \land p_2 \\
    \langle p_1 \land p_2, \sigma \rangle \rightarrow \langle p, \sigma \rangle \quad \text{where } p = p_1 \land p_2
    \]

• What if we want short-circuit evaluation of $\land$?
  - Define the following contexts, redexes, and local rules:
    \[
    H ::= \ldots | H \land b_2 \\
    r ::= \ldots | \text{true} \land b_2 | \text{false} \land b_2 \\
    \langle \text{true} \land b_2, \sigma \rangle \rightarrow \langle b_2, \sigma \rangle \\
    \langle \text{false} \land b_2, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle
    \]
  - The local reduction kicks in before $b_2$ is evaluated.
Contextual Semantics: Notes

- One can think of the • as representing the program counter.
- The advancement rules for • are not trivial.
  - At each step the entire command is decomposed.
  - This makes contextual semantics inefficient to implement directly.

- The major advantage of contextual semantics is that it allows a mix of local and global reduction rules.
  - For IMP we have only local reduction rules: only the redex is reduced.
  - Sometimes it is useful to work on the context too.
Some Further Topics

• Treatment of errors in operational semantics
  - with an explicit “error” result,
    as in \((3/0) \rightarrow \text{error}\),
  - with an “error” expression,
    as in \((3 + \text{error})\),
  - with “stuck” computations,
    so \((3/0) \rightarrow r\) for no \(r\).

• Treatment of overflow (see homework 2)
Contextual Semantics: Notes

• For example: \( c = c_1; c_2 \)
  - either \( c_1 = \) skip and then \( c = H[skip; c_2] \) with \( H = \bullet \)
  - or \( c_1 \neq \) skip and then \( c_1 = H[r] \);
    so \( c = c_1; c_2 = H[r]; c_2 = H'[r] \) where \( H' = H; c_2 \)

• For example: \( c = \text{if} \ b \ \text{then} \ c_1 \ \text{else} \ c_2 \)
  - either \( b = \) true or \( b = \) false and then \( c = H[r] \) with \( H = \bullet \)
  - or \( b \) is not a value and \( b = H[r] \);
    so \( c = H'[r] \) where \( H' = \text{if} \ H \ \text{then} \ c_1 \ \text{else} \ c_2 \)

• Decomposition theorem: If \( c \) is not “skip” then there exist unique \( H \) and \( r \) such that \( c \) is \( H[r] \).
  - \( \Rightarrow \) Progress and determinism.
Summary of Operational Semantics

• Precise specification of dynamic semantics:
  - order of evaluation (or that it doesn’t matter)
  - error conditions (sometimes implicitly, by rule applicability)
• Simple and abstract (cf. implementations)
  - no low-level details such as stack and memory management, data layout, etc.
• Often not compositional (as for while)
• Basis for some proofs about languages
• Basis for some reasoning about particular programs
• Point of reference for other semantics