Operational Semantics

Winskel, Ch. 2

Slonneger and Kurtz Ch 8.4, 8.5, 8.6
Semantics: How?

Operational semantics in the general mathematical sense: imagine an abstract machine

Some notion of machine state

Transition function: given the current machine state, what is the next machine state?

Goal: operational semantics of imperative language

machine state = program + contents of memory

transition = program simplification (rewriting) + changes to the memory

Think of this as a formal definition of an interpreter
Uses of Operational Semantics

Correctness: does this program have a run-time error?
Equivalence: given two programs, are they always semantically equivalent? Essential question for the correctness of compiler optimizations

Conditions for equivalence: given two programs, under what restrictions/conditions are they semantically equivalent? Needed to define compiler static analyses that prove these conditions before optimizations can be applied

Correctness of code generation: given any program and a translation algorithm to create low-level code (e.g., assembly code or Java bytecode), is the low-level program semantically equivalent to the original program? That is, can we prove the correctness of the translation algorithm?
IMP: Simple Imperative Language

\[ \textit{<c>} ::= \text{skip} \mid \text{id := } \textit{<ae>} \mid \textit{<c>} ; \textit{<c>} \]
\[ \mid \text{if } \textit{<be>} \text{ then } \textit{<c>} \text{ else } \textit{<c>} \]
\[ \mid \text{while } \textit{<be>} \text{ do } \textit{<c>} \]

\[ \textit{<ae>} ::= \text{id} \mid \text{const} \mid \textit{<ae>} + \textit{<ae>} \]
\[ \mid \textit{<ae>} - \textit{<ae>} \mid \textit{<ae>} \ast \textit{<ae>} \mid \textit{<ae>} / \textit{<ae>} \]

\[ \textit{<be>} ::= \text{true} \mid \text{false} \]
\[ \mid \textit{<ae>} = \textit{<ae>} \mid \textit{<ae>} < \textit{<ae>} \]
\[ \mid \neg \textit{<be>} \mid \textit{<be>} \land \textit{<be>} \]
\[ \mid \textit{<be>} \lor \textit{<be>} \]
Memory State (we will just say “State”)  

**State**: a function $\sigma$ from variable names to values  
An abstraction of the contents of the physical memory  
Example: program with two variables $x$ and $y$  

\[
\begin{align*}
\sigma(x) &= 9 \\
\sigma(y) &= 5
\end{align*}
\]

For simplicity, we will only consider integer variables  

$\sigma$: Variables $\rightarrow \{0, -1, 1, -2, 2, \ldots\}$  
“Variables” is the set of all variable names in the program
Evaluation for Arithmetic Expressions

Evaluation relation (4-way relation) for IMP arithmetic expressions

\(<ae,\sigma> \rightarrow <ae',\sigma>\) : the evaluation of \(ae\) from state \(\sigma\) is not completed; after one step, the remaining work is \(<ae',\sigma>\)

\(<ae,\sigma> \rightarrow <\text{const},\sigma>\) : the evaluation of \(ae\) from state \(\sigma\) has completed successfully with the value \(\text{const}\)

Full evaluation of \(ae_1\) from state \(\sigma\):

\(<ae_1,\sigma> \rightarrow <ae_2,\sigma> \rightarrow \ldots \rightarrow <ae_n,\sigma>\) where \(ae_n\) is \(\text{const}\)
Evaluation for Arithmetic Expressions

Inference rules for the evaluation relation

Syntax: \texttt{id} | \texttt{const} | <\texttt{ae}> + <\texttt{ae}> | ...

\[<\text{id}, \sigma> \rightarrow <\sigma(\text{id}), \sigma>\] axiom, applicable if \texttt{id} has a value in \(\sigma\)

\[<\text{const}_1 + \text{const}_2, \sigma> \rightarrow <\text{const}_3, \sigma>\] axiom, for any three integers such that \texttt{const}_3 is the sum of \texttt{const}_1 and \texttt{const}_2

\[<\text{ae}_1, \sigma> \rightarrow <\text{ae}_1', \sigma>\]

\[<\text{ae}_1 + \text{ae}_2, \sigma> \rightarrow <\text{ae}_1' + \text{ae}_2, \sigma>\] first evaluate the left operand

\[<\text{ae}_2, \sigma> \rightarrow <\text{ae}_2', \sigma>\] after the left operand is evaluated, evaluate the right one

\[<\text{const} + \text{ae}_2, \sigma> \rightarrow <\text{const} + \text{ae}_2', \sigma>\]
Example (parentheses used to clarify the associativity of +)

$ae_1$ is $(a+b)+11$, with $\sigma(a)=2$, $\sigma(b)=5$

Need: $<ae_1,\sigma> \rightarrow <ae_2,\sigma> \rightarrow ... \rightarrow <\text{const},\sigma>$

$ae_2$ is $(2+b)+11$

To get $<(a+b)+11,\sigma> \rightarrow <(2+b)+11,\sigma>$ we need $<a+b,\sigma> \rightarrow <2+b,\sigma>$ as a premise

To get $<a+b,\sigma> \rightarrow <2+b,\sigma>$ we need $<a,\sigma> \rightarrow <2,\sigma>$ as a premise

Derive $<a,\sigma> \rightarrow <2,\sigma>$ using axiom $<\text{id},\sigma> \rightarrow <\sigma(\text{id}),\sigma>$

$ae_3$ is $(2+5)+11$

To get $<(2+b)+11,\sigma> \rightarrow <(2+5)+11,\sigma>$ we need $<2+b,\sigma> \rightarrow <2+5,\sigma>$ as a premise

To get $<2+b,\sigma> \rightarrow <2+5,\sigma>$ we need $<b,\sigma> \rightarrow <5,\sigma>$ as a premise

Derive $<b,\sigma> \rightarrow <5,\sigma>$ using axiom $<\text{id},\sigma> \rightarrow <\sigma(\text{id}),\sigma>$

$ae_4$ is $7+11$

To get $<(2+5)+11,\sigma> \rightarrow <7+11,\sigma>$ we need $<2+5,\sigma> \rightarrow <7,\sigma>$ as a premise

Derive $<2+5,\sigma> \rightarrow <7,\sigma>$ using axiom $<\text{const}_1+\text{const}_2,\sigma> \rightarrow <\text{const}_3,\sigma>$

$ae_5$ is 18

Derive $<7+11,\sigma> \rightarrow <8,\sigma>$ using axiom $<\text{const}_1+\text{const}_2,\sigma> \rightarrow <\text{const}_3,\sigma>$
Run-time Errors

When can the abstract machine “get stuck”?

\[ <\text{id},\sigma> \rightarrow <\sigma(\text{id}),\sigma> \]  
axiom, applicable if \text{id} has a value in \sigma

But what if \text{id} does not have a value in \sigma? We are trying to read an uninitialized variable

\[ <\text{const}_1/\text{const}_2,\sigma> \rightarrow <\text{const}_3,\sigma> \]  
axiom, for any integers such that \text{const}_3 is \text{const}_1/\text{const}_2  
(integer division)

But what if \text{const}_2 is zero? Integer division of \text{const}_1/0 is not defined

Note: our semantics assumes infinite-precision arithmetic (no overflow/underflow); not true for real languages
Evaluation for Boolean Expressions

Goal: \(<be_1, \sigma> \rightarrow <be_2, \sigma> \rightarrow \ldots \rightarrow <be_n, \sigma>\) where \(be_n\) is true or false

Syntax: true | false | \(<ae> = <ae>\) | \(<ae> < <ae>\) | \(\neg <be>\) | \(<be> \land <be>\) | \(<be> \lor <be>\)

\(<\text{const} = \text{const}, \sigma> \rightarrow <\text{true}, \sigma>\) \hspace{1cm} \text{axiom}

\(<\text{const}_1 = \text{const}_2, \sigma> \rightarrow <\text{false}, \sigma>\) \hspace{1cm} \text{axiom, for distinct integers}

\(<ae_1, \sigma> \rightarrow <ae_1', \sigma>\)
\(<ae_1 = ae_2, \sigma> \rightarrow <ae_1' = ae_2, \sigma>\)

first evaluate the left operand

\(<ae_2, \sigma> \rightarrow <ae_2', \sigma>\)
\(<\text{const} = ae_2, \sigma> \rightarrow <\text{const} = ae_2', \sigma>\)

after the left operand is evaluated, evaluate the right one

Similar rules for \(<ae> < <ae>\)
### Evaluation for Boolean Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt; \neg true, \sigma &gt; ) → (&lt; false, \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; be, \sigma &gt; ) → (&lt; be', \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; \neg be, \sigma &gt; ) → (&lt; \neg be', \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; true \land true, \sigma &gt; ) → (&lt; true, \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; false \land true, \sigma &gt; ) → (&lt; false, \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; true \land false, \sigma &gt; ) → (&lt; false, \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; false \land false, \sigma &gt; ) → (&lt; false, \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; be_1, \sigma &gt; ) → (&lt; be_1', \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; be_1 \land be_2, \sigma &gt; ) → (&lt; be_1' \land be_2, \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; be_2, \sigma &gt; ) → (&lt; be_2', \sigma &gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt; true \land be_2, \sigma &gt; ) → (&lt; true \land be_2', \sigma &gt;)</td>
<td></td>
</tr>
</tbody>
</table>

Similar rule for \(false\)
Evaluation for Boolean Expressions

**Goal:** change the rules to allow short-circuit evaluation

\[
<\text{false} \land \text{be}, \sigma> \rightarrow <\text{false}, \sigma>
\]

\[
<\text{true} \land \text{be}, \sigma> \rightarrow <\text{be}, \sigma>
\]

\[
<\text{be}_1, \sigma> \rightarrow <\text{be}_1', \sigma>
\]

\[
<\text{be}_1 \land \text{be}_2, \sigma> \rightarrow <\text{be}_1', \land \text{be}_2, \sigma>
\]

How about the short-circuit rules for “or”?
Execution of Statements

**Expression:** produces a value; does not change the memory \( \sigma \) (the evaluation does not have **side effects** on the memory)

**Note:** in imperative languages, some expressions can have side effects (e.g. in C: \( x++ \) or \( f() \) if function \( f \) changes some global var)

**Statement:** does not produce a value; changes the memory \( \sigma \); so, we evaluate an expression but we execute a statement

**Syntax:**  
\[
\mathcal{C} ::= \text{skip} \mid \text{id} := \langle \mathcal{A} \rangle \mid \langle \mathcal{C} \rangle ; \langle \mathcal{C} \rangle \\
| \text{if } \langle \mathcal{B} \rangle \text{ then } \langle \mathcal{C} \rangle \text{ else } \langle \mathcal{C} \rangle | \text{while } \langle \mathcal{B} \rangle \text{ do } \langle \mathcal{C} \rangle
\]

**Semantics:**  
\[
\langle \mathcal{C}, \sigma \rangle \xrightarrow{\text{more work left}} \langle \mathcal{C}', \sigma' \rangle \text{ or } \langle \mathcal{C}, \sigma \rangle \xrightarrow{\text{no work left}} \sigma'
\]

**Full execution:**  
\[
\langle \mathcal{C}_1, \sigma_1 \rangle \xrightarrow{\text{more work left}} \langle \mathcal{C}_2, \sigma_2 \rangle \xrightarrow{\text{...}} \langle \mathcal{C}_{n-1}, \sigma_{n-1} \rangle \xrightarrow{\text{no work left}} \sigma_n
\]

where \( \mathcal{C}_1 \) is the program

**Idea:** we rewrite the program and change the state, until the program is reduced to nothing and we have the final state \( \sigma_n \)
Assignments and Skip

Syntax: \texttt{id} := \langle \texttt{ae} \rangle

\[
\langle \texttt{id}:=\texttt{const},\sigma \rangle \rightarrow \sigma[\texttt{const}/\texttt{id}]
\]

\[
\langle \texttt{ae},\sigma \rangle \rightarrow \langle \texttt{ae}',\sigma \rangle
\]

\[
\langle \texttt{id}:=\texttt{ae},\sigma \rangle \rightarrow \langle \texttt{id}:=\texttt{ae}',\sigma \rangle
\]

\[
\sigma[m/x] \text{ is the same state as } \sigma \text{ except for } x; \ x \text{ now has the value } m
\]

\[
\sigma[m/x](y) = \sigma(y) \text{ if } y \text{ is not } x
\]

\[
\sigma[m/x](x) = m
\]
Sequences and Ifs

Syntax: \(<c> ; <c> \mid \textbf{if} \ <\text{be}> \ \textbf{then} \ <c> \ \textbf{else} \ <c>\)

\[
\begin{align*}
<\text{c}_1,\sigma> & \rightarrow <\text{c}_1',\sigma'> \\
<\text{c}_1;\text{c}_2, \sigma> & \rightarrow <\text{c}_1';\text{c}_2, \sigma'>
\end{align*}
\]

To execute a sequence of commands, start by executing the first element of the sequence

\[
<\text{if true then } \text{c}_1 \ \text{else } \text{c}_2, \sigma> \rightarrow <\text{c}_1,\sigma>
\]

\[
<\text{if false then } \text{c}_1 \ \text{else } \text{c}_2, \sigma> \rightarrow <\text{c}_2,\sigma>
\]

\[
<\text{be},\sigma> \rightarrow <\text{be}',\sigma>
\]

\[
<\text{if be then } \text{c}_1 \ \text{else } \text{c}_2, \sigma> \rightarrow <\text{if be}' \ \text{then } \text{c}_1 \ \text{else } \text{c}_2, \sigma>
\]

To execute an if statement, first evaluate its boolean expression
While Statements

Syntax: **while** <be> **do** <c>

\[
<\text{while} \; \text{be} \; \text{do} \; \text{c} \; , \; \sigma> \rightarrow \\
<\text{if} \; \text{be} \; \text{then} \; (\text{c} ; \; \text{while} \; \text{be} \; \text{do} \; \text{c}) \; \text{else} \; \text{skip} \; , \; \sigma>
\]

To execute a **while** statement, treat it as an if statement:
- evaluate its boolean expression
- if the expression evaluates to **true**, execute the loop body and then consider the loop again
- if the expression evaluates to **false**, reduce the entire loop to **skip** (i.e., complete the execution of the loop)
Example (parentheses added for readability)

Goal: \(<c_1, \sigma_1> \rightarrow <c_2, \sigma_2> \rightarrow \ldots \rightarrow \sigma_n\)

Suppose \(c_1\) is while \((x<3)\) do \(x:=x+1\) and \(\sigma_1(x)=1\)

\(c_2\) is if \((x<3)\) then \((x:=x+1; \text{while } (x<3) \text{ do } x:=x+1)\) else skip

\(\sigma_2\) is same as \(\sigma_1\)

\(c_3\) is if \((1<3)\) then \((x:=x+1; \text{while } (x<3) \text{ do } x:=x+1)\) else skip

\(\sigma_3\) is same as \(\sigma_1\)

To get \(<c_2, \sigma_1> \rightarrow <c_3, \sigma_1>\) we need \(<x<3, \sigma_1> \rightarrow <1<3, \sigma_1>\)

For that, we need \(<x, \sigma_1> \rightarrow <1, \sigma_1>\) (axiom)

\(c_4\) is if true then \((x:=x+1; \text{while } (x<3) \text{ do } x:=x+1)\) else skip

\(\sigma_4\) is same as \(\sigma_1\)

To get \(<c_3, \sigma_1> \rightarrow <c_4, \sigma_1>\) we use axiom \(<1<3, \sigma_1> \rightarrow <\text{true}, \sigma_1>\)
Example (parentheses added for readability)

\(c_5\) is \(x:=x+1;\) while \((x<3)\) do \(x:=x+1\) \(\quad \sigma_5\) is same as \(\sigma_1\)

\(c_6\) is \(x:=1+1;\) while \((x<3)\) do \(x:=x+1\) \(\quad \sigma_6\) is same as \(\sigma_1\)
To get \(<c_5,\sigma_1> \rightarrow <c_6,\sigma_1>\) we need \(<x:=x+1,\sigma_1> \rightarrow <x:=1+1,\sigma_1>\)
To get that, we need \(<x+1,\sigma_1> \rightarrow <1+1,\sigma_1>\)
To get that, we need \(<x,\sigma_1> \rightarrow <1,\sigma_1>\) (axiom)

\(c_7\) is \(x:=2;\) while \((x<3)\) do \(x:=x+1\) \(\quad \sigma_7\) is same as \(\sigma_1\)
To get \(<c_6,\sigma_1> \rightarrow <c_7,\sigma_1>\) we need \(<x:=1+1,\sigma_1> \rightarrow <x:=2,\sigma_1>\)
To get that, we need \(<1+1,\sigma_1> \rightarrow <2,\sigma_1>\) (axiom)

\(c_8\) is while \((x<3)\) do \(x:=x+1\) \(\quad \sigma_8\) is \(\sigma_1[2/x]\)
To get \(<c_7,\sigma_1> \rightarrow <c_8,\sigma_8>\) we need \(<x:=2,\sigma_1> \rightarrow \sigma_8\) (axiom)

Note: \(c_8\) is the same as \(c_1\) (the original program), just the state has changed
Example (parentheses added for readability)

\[ c_9 \text{ is if } (x<3) \text{ then } (x:=x+1; \text{ while } (x<3) \text{ do } x:=x+1) \text{ else skip} \]

\[ c_{10} \text{ is if } (2<3) \text{ then } (x:=x+1; \text{ while } (x<3) \text{ do } x:=x+1) \text{ else skip} \]

\[ c_{11} \text{ is if true then } (x:=x+1; \text{ while } (x<3) \text{ do } x:=x+1) \text{ else skip} \]

\[ c_{12} \text{ is } x:=x+1; \text{ while } (x<3) \text{ do } x:=x+1 \]

\[ c_{13} \text{ is } x:=2+1; \text{ while } (x<3) \text{ do } x:=x+1 \]

\[ c_{14} \text{ is } x:=3; \text{ while } (x<3) \text{ do } x:=x+1 \]

\[ c_{15} \text{ is } \text{while } (x<3) \text{ do } x:=x+1 \quad \sigma_{15} \text{ is } \sigma_8[3/x] \]

Note: \( c_{15} \) is the same as \( c_1 \) (the original program), just the state has changed

\[ c_{16} \text{ is if } (x<3) \text{ then } (x:=x+1; \text{ while } (x<3) \text{ do } x:=x+1) \text{ else skip} \]

\[ c_{17} \text{ is if } (3<3) \text{ then } (x:=x+1; \text{ while } (x<3) \text{ do } x:=x+1) \text{ else skip} \]

\[ c_{18} \text{ is if false then } (x:=x+1; \text{ while } (x<3) \text{ do } x:=x+1) \text{ else skip} \]

\[ c_{19} \text{ is } \text{skip} \text{ and } \sigma_{19} \text{ is } \sigma_8[3/x] \text{ which is same as } \sigma_1[3/x] \]

Final state: \( \sigma_1[3/x] \); conclusion: the loop terminates; the final value of \( x \) is 3; the values of all other variables are unchanged
Another Example

c_1 is while true do skip

c_2 is if true then (skip; while true do skip) else skip

c_3 is skip; while true do skip

c_4 is while true do skip

c_4 is same as c_1 and we will never be able to derive anything else of the form while <be> ...; we have non-terminating loop

This is not the same as being “stuck” due to a run-time error (division by zero, or use of an uninitialized variable): a stuck program is still considered a terminating program
Properties of This Operational Semantics

**Determinism:** suppose a given program $c$ terminates when executed from initial state $\sigma$. There exists exactly one longest sequence $<c,\sigma> \rightarrow <c_2,\sigma_2> \rightarrow \ldots$

- If this sequence ends with a state $\sigma_n$, the program terminated successfully
- If this sequence ends with some $<c_n,\sigma_n>$, the program terminated with a run-time error (e.g., $<x:=5/0,\sigma_n>$)

Note: if $c$ does not terminate when executed from $\sigma$, the infinite sequence $<c,\sigma> \rightarrow <c_2,\sigma_2> \rightarrow \ldots$ is also unique
Properties of This Operational Semantics

Semantic equivalence: two programs c and c’ are equivalent if, for every initial state σ, one of the following is true:

1. \(<c,\sigma> \text{ terminates successfully with } \sigma_n \text{ and } <c’,\sigma> \text{ terminates successfully with the same } \sigma_n\)

2. \(<c,\sigma> \text{ terminates with a run-time error and } <c’,\sigma> \text{ terminates with a run-time error}\)

3. \(<c,\sigma> \text{ does not terminate and } <c’,\sigma> \text{ does not terminate}\)
Simple Example (parentheses used to clarify the order of +)

Initial program: \[ x := ((a_1 + a_2) + a_3) \ldots + a_{100} \]

Modified: \[ x := ((a_1 + a_2) \ldots + a_{50}) + ((a_{51} + a_{52}) \ldots + a_{100}) \]

Why: parallelization – compute the sum of the first 50 numbers in one thread; in parallel, compute the second sum in another thread; add up the two results

Simplified version: is \( x := (a+b) + c \) equivalent to \( x := a + (b+c) \)?

Intuitively, seems obviously true, but we can prove it formally using the operational semantics (details not shown)

Key assumption of the proof: if we have values \( v_1, v_2, v_3 \) then \((v_1 + v_2) + v_3\) is the same value as \( v_1 + (v_2 + v_3) \)

In our semantics they are the same, but not in real languages.
Example (parentheses used to show the higher precedence of ++) 

Initial program: \( x := a + (a++) \) 
Modified: \( x := (a++) + a \) 

First, need to define the semantics of \texttt{id++} 

\[
\langle \texttt{id++}, \sigma \rangle \rightarrow \langle \sigma(\texttt{id}), \sigma[\text{const/id}] \rangle
\]

axiom, \texttt{const} is equal to \(1 + \sigma(\texttt{id})\) 

Suppose we have some \( \sigma \) with \( \sigma(a) = v_1 \) 

\( x := a + (a++) \) 
\( x := v_1 + (a++) \) 
\( x := v_1 + v_1 \) with a new state mapping \( a \) to \(1 + v_1\) 
\( x := v_2 \) where \( v_2 \) is equal to the sum of \( v_1 \) and \( v_1 \) 

final state maps \( x \) to \( v_2 \)
Example (parentheses used to show the higher precedence of ++)  

Modified:  \( x := (a++) + a \)

\( x := v_1 + a \) with a new state mapping \( a \) to \( 1 + v_1 \)
\( x := v_1 + v_3 \) where \( v_3 \) is equal to \( 1 + v_1 \)
\( x := v_4 \) where \( v_4 \) is equal to the sum of \( v_1 \) and \( v_3 \)

final state maps \( x \) to \( v_4 \)

Clearly the programs are not semantically equivalent

Note: the order of evaluation of operands of operator + is **not** defined in C, but is fixed in our semantics (left operand is evaluated first); thus, in C the modified program could produce \( 2 * v_1 \) or \( 2 * v_1 + 1 \) at the discretion of the compiler

Note: **order of side effects** is also not specified in C: e.g., \( i = ++i + 1; \) and \( a[i++] = i; \) have several possible behaviors

\( ++i \) is two things: “produce value \( 1 + \sigma(i) \)” and “change \( \sigma(i) \)” [which is side effect of evaluation]; but when does this side effect happen?
Small-step vs Big-step Semantics

So far: a form of operational semantics which shows one “elementary” step of execution: small-step semantics

**Alternative:** big-step (a.k.a. “natural”) operational semantics to show directly the final result of evaluation/execution

If the state is $\sigma$ and expression $e$ is evaluated fully without run-time error, what is the resulting value?

- $\langle ae, \sigma \rangle \rightarrow const$ for arithmetic expression
- $\langle be, \sigma \rangle \rightarrow true$ or $false$ for boolean expressions

If the state is $\sigma$ and command $c$ is executed to termination without a run-time error, what is the resulting state?

- $\langle c, \sigma \rangle \rightarrow \sigma'$
Evaluation for Arithmetic Expressions

Syntax: \texttt{id} | \texttt{const} | \texttt{<ae> + <ae>} | ... \\

\begin{align*}
\langle \texttt{const}, \sigma \rangle & \rightarrow \texttt{const} \\
\langle \texttt{id}, \sigma \rangle & \rightarrow \sigma(\texttt{id}) \quad \text{axiom, applicable if \texttt{id} has a value in} \ \sigma \\
\langle \texttt{ae}_1, \sigma \rangle & \rightarrow \texttt{const}_1 \quad \langle \texttt{ae}_2, \sigma \rangle \rightarrow \texttt{const}_2 \\
\hline
\langle \texttt{ae}_1 + \texttt{ae}_2, \sigma \rangle & \rightarrow \texttt{const}_3 \\
\hline
\end{align*}

\texttt{const}_3 \text{ is the sum of } \texttt{const}_1 \text{ and } \texttt{const}_2

With small-step semantics, we enforced that the left operand is evaluated first, before the second operand is touched. Here we do not have any constraints: nothing in the semantic rules tells us in which order the operands will be evaluated. In fact, their evaluation could be interleaved – do a bit of work for \texttt{ae}_1 then do a bit of work for \texttt{ae}_2 then go again to \texttt{ae}_1 etc. (or even evaluate them in parallel)
Evaluation for Boolean Expressions

Syntax: true | false | \(<ae>=<ae>\) | \(<ae><ae>\) | \(\neg be\) | \(be\wedge be\) | \(be\vee be\)

\(<true, \sigma> \rightarrow true\)
\(<false, \sigma> \rightarrow false\)

\(<ae_1, \sigma> \rightarrow const_1\)
\(<ae_2, \sigma> \rightarrow const_2\)

\(<ae_1=ae_2, \sigma> \rightarrow true\)

if const_1 and const_2 are equal (and a similar rule for const_1 \(\neq\) const_2)

Also, similar rules for \(<ae> < <ae>\)

\(<be, \sigma> \rightarrow true\)
\(<be, \sigma> \rightarrow false\)

\(<\neg be, \sigma> \rightarrow false\)
\(<\neg be, \sigma> \rightarrow true\)

\(<be_1, \sigma> \rightarrow true\)
\(<be_2, \sigma> \rightarrow true\)

\(<be_1 \wedge be_2, \sigma> \rightarrow true\)

Also, similar rules for \(<be> \vee <be>\)

and three more similar rules, for true/false, false/true, false/false
**Short-Circuit Evaluation**

<table>
<thead>
<tr>
<th>$&lt;\text{be}_1, \sigma&gt; \rightarrow \text{true}$</th>
<th>$&lt;\text{be}_2, \sigma&gt; \rightarrow \text{true}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;\text{be}_1 \land \text{be}_2, \sigma&gt; \rightarrow \text{true}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$&lt;\text{be}_1, \sigma&gt; \rightarrow \text{true}$</th>
<th>$&lt;\text{be}_2, \sigma&gt; \rightarrow \text{false}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;\text{be}_1 \land \text{be}_2, \sigma&gt; \rightarrow \text{false}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$&lt;\text{be}_1, \sigma&gt; \rightarrow \text{false}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;\text{be}_1 \land \text{be}_2, \sigma&gt; \rightarrow \text{false}$</td>
</tr>
</tbody>
</table>

How about the rules for $\text{be} \lor \text{be}$?
Statements

\[
\begin{align*}
\langle ae, \sigma \rangle & \rightarrow \text{const} \\
\langle \text{id}:=ae, \sigma \rangle & \rightarrow \sigma[\text{const/id}] \\
\langle c_1, \sigma \rangle & \rightarrow \sigma' \quad \langle c_2, \sigma' \rangle \rightarrow \sigma'' \\
\langle c_1; c_2, \sigma \rangle & \rightarrow \sigma'' \\
\langle be, \sigma \rangle & \rightarrow \text{true} \quad \langle c_1, \sigma \rangle \rightarrow \sigma' \\
\langle \text{if be then } c_1 \text{ else } c_2, \sigma \rangle & \rightarrow \sigma' \\
\langle be, \sigma \rangle & \rightarrow \text{false} \quad \langle c_2, \sigma \rangle \rightarrow \sigma' \\
\langle \text{if be then } c_1 \text{ else } c_2, \sigma \rangle & \rightarrow \sigma'
\end{align*}
\]
What happens with infinite loops? We will not be able to create a derivation tree (e.g., no tree for `while true do skip`)

What happens with run-time errors? Similarly, we will not be able to create a derivation tree (e.g., no tree for `x:=y+1` if `y` is not initialized)
Properties of Big-Step Operational Semantics

**Determinism:** suppose a given program $c$ terminates normally (without a run-time error or infinite loop) when executed from initial state $\sigma$. Then there exists a unique state $\sigma'$ such that $<c,\sigma> \rightarrow \sigma'$

Note: If there is a run-time error or infinite loop, it is impossible to derive $<c,\sigma> \rightarrow \sigma'$

**Semantic equivalence:** programs $c$ and $c'$ are equivalent if, for every initial state $\sigma$, $<c,\sigma> \rightarrow \sigma'$ if and only if $<c',\sigma> \rightarrow \sigma'$

Note: If for some $\sigma$ program $c$ terminates normally but $c'$ does not (or vice versa), they are no equivalent. Either both succeed, or both fail.
Example of Semantic Equivalence

Loop peeling: transform `while be do c`

Modified: `if be then (c; while be do c) else skip`

- Take the first iteration out of the loop
- Common compiler optimization: enables a variety of other optimizations

Can we prove that this transformation is semantics-preserving: for every initial state $\sigma$, $<c, \sigma> \rightarrow \sigma'$ if and only if $<c', \sigma> \rightarrow \sigma'$
First Half of the Proof (the other half is similar)

If $<\text{while...},\sigma> \rightarrow \sigma'$ is derivable, so is $<\text{if...},\sigma> \rightarrow \sigma'$

If we can derive $<\text{while...},\sigma> \rightarrow \sigma'$ there are only two possibilities: there must be some derivation w/ premises

1. $<\text{be},\sigma> \rightarrow \text{false}$ ($\sigma$ and $\sigma'$ are the same state), or
2. $<\text{be},\sigma> \rightarrow \text{true}$. Combine $<\text{c},\sigma> \rightarrow \sigma_{\text{interm}}$ and $<\text{while be do c},\sigma_{\text{interm}}> \rightarrow \sigma'$ into $<\text{c;while...},\sigma> \rightarrow \sigma'$

Case 1: $<\text{be},\sigma> \rightarrow \text{false}$. Use $<\text{skip},\sigma> \rightarrow \sigma'$ (axiom, since $\sigma$ and $\sigma'$ are the same) to derive $<\text{if be then ... else skip},\sigma> \rightarrow \sigma'$

Case 2: $<\text{be},\sigma> \rightarrow \text{true}$. Combine $<\text{c},\sigma> \rightarrow \sigma_{\text{interm}}$ and $<\text{while be do c},\sigma_{\text{interm}}> \rightarrow \sigma'$ into $<\text{c;while...},\sigma> \rightarrow \sigma'$ and then derive $<\text{if be then c;while... else skip}, \sigma> \rightarrow \sigma'$
Other Examples (in their general form, advanced compiler optimizations)

Partial redundancy elimination

if be then \{ x:=e_1 \} else \{ y:=e_2 \}; x:=e_1 \text{ transformed to }
if be then \{ x:=e_1 \} else \{ y:=e_2; x:=e_1 \}

Under what conditions are these two programs equivalent?

Movement of loop-invariant code

Example: while be do \{ x:=1+1; y:=y+x \} is it equivalent to
\[ \boxed{x:=1+1; \text{ while be do } \{ y:=y+x \}} \]

Example: do \{ x:=1+1; y:=y+x \} while be is it equivalent to
\[ \boxed{x:=1+1; \text{ do } \{ y:=y+x \} \text{ while be}} \]

Example: do \{ y:=y+x; \boxed{x:=1+1} \} while be is it equivalent to
\[ \boxed{x:=1+1; \text{ do } \{ y:=y+x \} \text{ while be}} \]