

Operational Semantics

A programming language

- ▶ Syntax
- ▶ Semantics

Why formal semantics

Formal semantics gives an unambiguous definition of what a program written in the language should do.

- ▶ Understand the subtleties of the language
- ▶ Offer a formal reference and a correctness definition for implementors of tools (parsers, compilers, interpreters, debuggers, etc)
- ▶ Prove global properties of any program written in the language
- ▶ Verify programs against formal specifications
- ▶ Prove two different programs are equivalent/non-equivalent
- ▶ From a computer readable version of the semantics, an interpreter can be automatically generated (full compiler generation is not yet feasible)
- ▶ ...

Formal semantics of a programming language

- ▶ Operational semantics
- ▶ Denotational semantics
- ▶ Axiomatic semantics

Operational semantics

Operational semantics defines program executions:

- ▶ Sequence of steps, formulated as **transitions of an abstract machine**

Configurations of the abstract machine include:

- ▶ **Expression/statement** being evaluated/executed
- ▶ **States**: abstract description of registers, memory and other data structures involved in computation

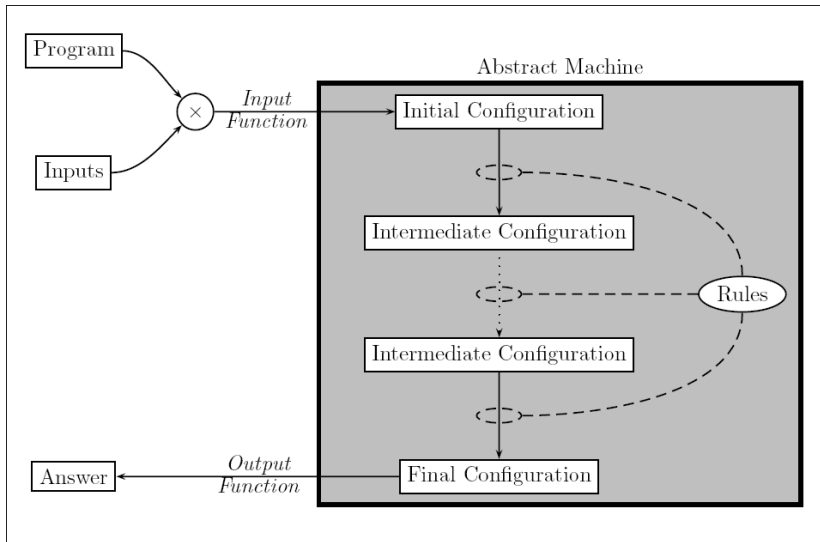


Figure taken from Franklyn Turbak and David Gifford's *Design Concepts in Programming Languages*.

Different approaches of operational semantics

- ▶ **Small-step semantics:**
Describe each *single step* of the execution
- ▶ **Big-step semantics:**
Describe the *overall result* of the execution

We will explain both in detail by examples.

After this class...

You should be able to:

- ▶ write down the evaluation/execution steps, if given the operational semantics rules
- ▶ formulate the operational semantics rule, if given the informal meaning of an expression/statement

Outline

Syntax of a Simple Imperative Language

Operational semantics

Small-step operational semantics

- Structural operational semantics (SOS)

- Extensions: going wrong, local variable declaration, heap

- Contextual semantics (a.k.a. reduction semantics)

Big-step operational semantics

Outline

Syntax of a Simple Imperative Language

Operational semantics

Small-step operational semantics

- Structural operational semantics (SOS)

- Extensions: going wrong, local variable declaration, heap

- Contextual semantics (a.k.a. reduction semantics)

Big-step operational semantics

Syntax

(IntExp) $e ::= \mathbf{n}$
 | x
 | $e + e \mid e - e \mid \dots$

(BoolExp) $b ::= \mathbf{true} \mid \mathbf{false}$
 | $e = e \mid e < e \mid e > e$
 | $\neg b \mid b \wedge b \mid b \vee b \mid \dots$

(Comm) $c ::= \mathbf{skip}$
 | $x := e$
 | $c ; c$
 | $\mathbf{if } b \mathbf{ then } c \mathbf{ else } c$
 | $\mathbf{while } b \mathbf{ do } c$

Syntax

$$(\text{IntExp}) \quad e ::= \mathbf{n} \mid x \mid e + e \mid e - e \mid \dots$$

Here \mathbf{n} ranges over the numerals $\mathbf{0}, \mathbf{1}, \mathbf{2}, \dots$.

We distinguish between numerals, written $\mathbf{n}, \mathbf{0}, \mathbf{1}, \mathbf{2}, \dots$, and the natural numbers, written $n, 0, 1, 2, \dots$. The natural numbers are the normal numbers that we use in everyday life, while the numerals are just syntax for describing these numbers.

We write $\llbracket \mathbf{n} \rrbracket$ to denote the meaning of \mathbf{n} . We assume that $\llbracket \mathbf{n} \rrbracket = n$, $\llbracket \mathbf{0} \rrbracket = 0$, $\llbracket \mathbf{1} \rrbracket = 1$, \dots .

The distinction is subtle, but important, because it is one manifestation of [the difference between syntax and semantics](#).

Syntax

	Syntax	Semantics $[\cdot]$
(IntExp) e	$::=$ n	n
	x	
	$e + e$	$+$
	$e - e$	$-$
	\dots	
(BoolExp) b	$::=$ true	<i>true</i>
	false	<i>false</i>
	$e = e$	$=$
	$e < e$	$<$
	$\neg b$	\neg
	$b \wedge b$	\wedge
	$b \vee b$	\vee
	\dots	

Outline

Syntax of a Simple Imperative Language

Operational semantics

- Small-step operational semantics

 - Structural operational semantics (SOS)

 - Extensions: going wrong, local variable declaration, heap

 - Contextual semantics (a.k.a. reduction semantics)

- Big-step operational semantics

States

To evaluate variables or update variables, we need to know the current state.

$$\text{(State)} \quad \sigma \in \text{Var} \rightarrow \text{Values}$$

What are Values? \mathbf{n} or n ?

Both are fine. Here we think Values are natural numbers, boolean values, etc.

States

(State) $\sigma \in \text{Var} \rightarrow \text{Values}$

For example, $\sigma_1 = \{(x, 2), (y, 3), (a, 10)\}$, which we will write as $\{x \rightsquigarrow 2, y \rightsquigarrow 3, a \rightsquigarrow 10\}$.

(For simplicity, here we assume that a state always contain all the variables that may be used in a program.)

States

(State) $\sigma \in \text{Var} \rightarrow \text{Values}$

For example, $\sigma_1 = \{(x, 2), (y, 3), (a, 10)\}$, which we will write as $\{x \rightsquigarrow 2, y \rightsquigarrow 3, a \rightsquigarrow 10\}$.

(For simplicity, here we assume that a state always contain all the variables that may be used in a program.)

Recall

$$\sigma\{x \rightsquigarrow n\} \stackrel{\text{def}}{=} \lambda z. \begin{cases} \sigma(z) & \text{if } z \neq x \\ n & \text{if } z = x \end{cases}$$

For example, $\sigma_1\{y \rightsquigarrow 7\} = \{x \rightsquigarrow 2, y \rightsquigarrow 7, a \rightsquigarrow 10\}$.

States

(State) $\sigma \in \text{Var} \rightarrow \text{Values}$

For example, $\sigma_1 = \{(x, 2), (y, 3), (a, 10)\}$, which we will write as $\{x \rightsquigarrow 2, y \rightsquigarrow 3, a \rightsquigarrow 10\}$.

(For simplicity, here we assume that a state always contain all the variables that may be used in a program.)

Recall

$$\sigma\{x \rightsquigarrow n\} \stackrel{\text{def}}{=} \lambda z. \begin{cases} \sigma(z) & \text{if } z \neq x \\ n & \text{if } z = x \end{cases}$$

For example, $\sigma_1\{y \rightsquigarrow 7\} = \{x \rightsquigarrow 2, y \rightsquigarrow 7, a \rightsquigarrow 10\}$.

Operational semantics will be defined using **configurations** of the forms (e, σ) , (b, σ) and (c, σ) .

Small-step structural operational semantics (SOS)

Systematic definition of operational semantics:

- ▶ The program syntax is inductively-defined
- ▶ So we can also define the semantics of a program in terms of the semantics of its parts
- ▶ “Structural”: syntax oriented and inductive

Examples:

- ▶ The state transition for $e_1 + e_2$ is described using the transition for e_1 and the transition for e_2 .
- ▶ The state transition for $c_1 ; c_2$ is described using the transition for c_1 and the transition for c_2 .

Small-step SOS for expression evaluation

Recall

$$\text{(IntExp)} \quad e ::= \mathbf{n} \mid x \mid e + e \mid e - e \mid \dots$$

Below we define $(e, \sigma) \longrightarrow (e', \sigma')$. We'll start from addition.

Small-step SOS for expression evaluation

Recall

(IntExp) $e ::= \mathbf{n} \mid x \mid e + e \mid e - e \mid \dots$

Below we define $(e, \sigma) \longrightarrow (e', \sigma')$. We'll start from addition.

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 + e_2, \sigma) \longrightarrow (e'_1 + e_2, \sigma)}$$

$$\frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} + e_2, \sigma) \longrightarrow (\mathbf{n} + e'_2, \sigma)}$$

Small-step SOS for expression evaluation

Recall

(IntExp) $e ::= \mathbf{n} \mid x \mid e + e \mid e - e \mid \dots$

Below we define $(e, \sigma) \longrightarrow (e', \sigma')$. We'll start from addition.

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 + e_2, \sigma) \longrightarrow (e'_1 + e_2, \sigma)} \qquad \frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} + e_2, \sigma) \longrightarrow (\mathbf{n} + e'_2, \sigma)}$$

$$\frac{[\mathbf{n}_1] \quad [+]\quad [\mathbf{n}_2] = [\mathbf{n}]}{(\mathbf{n}_1 + \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

Small-step SOS for expression evaluation

Recall

(IntExp) $e ::= \mathbf{n} \mid x \mid e + e \mid e - e \mid \dots$

Below we define $(e, \sigma) \longrightarrow (e', \sigma')$. We'll start from addition.

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 + e_2, \sigma) \longrightarrow (e'_1 + e_2, \sigma)} \qquad \frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} + e_2, \sigma) \longrightarrow (\mathbf{n} + e'_2, \sigma)}$$

$$\frac{[\mathbf{n}_1] \quad [+]\quad [\mathbf{n}_2] = [\mathbf{n}]}{(\mathbf{n}_1 + \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

Example: $((\mathbf{10} + \mathbf{12}) + (\mathbf{13} + \mathbf{20}), \sigma)$

Small-step SOS for expression evaluation

It is important to note that the order of evaluation is fixed by the small-step semantics.

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 + e_2, \sigma) \longrightarrow (e'_1 + e_2, \sigma)}$$

$$\frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} + e_2, \sigma) \longrightarrow (\mathbf{n} + e'_2, \sigma)}$$

It is different from the following.

$$\frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(e_1 + e_2, \sigma) \longrightarrow (e_1 + e'_2, \sigma)}$$

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 + \mathbf{n}, \sigma) \longrightarrow (e_1 + \mathbf{n}, \sigma)}$$

Next: subtraction.

Small-step SOS for expression evaluation

Transitions for subtraction:

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 - e_2, \sigma) \longrightarrow (e'_1 - e_2, \sigma)}$$

$$\frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} - e_2, \sigma) \longrightarrow (\mathbf{n} - e'_2, \sigma)}$$

$$\frac{[\mathbf{n}_1] \quad [-] \quad [\mathbf{n}_2] = [\mathbf{n}]}{(\mathbf{n}_1 - \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

Next: variables.

Small-step SOS for expression evaluation

Recall

(State) $\sigma \in \text{Var} \rightarrow \text{Values}$

Transitions for evaluating variables:

$$\frac{\sigma(x) = \lfloor \mathbf{n} \rfloor}{(x, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

Summary: small-step SOS for expression evaluation

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 + e_2, \sigma) \longrightarrow (e'_1 + e_2, \sigma)}$$

$$\frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} + e_2, \sigma) \longrightarrow (\mathbf{n} + e'_2, \sigma)}$$

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 - e_2, \sigma) \longrightarrow (e'_1 - e_2, \sigma)}$$

$$\frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} - e_2, \sigma) \longrightarrow (\mathbf{n} - e'_2, \sigma)}$$

$$\frac{\lfloor \mathbf{n}_1 \rfloor \quad \lfloor + \rfloor \quad \lfloor \mathbf{n}_2 \rfloor = \lfloor \mathbf{n} \rfloor}{(\mathbf{n}_1 + \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

$$\frac{\lfloor \mathbf{n}_1 \rfloor \quad \lfloor - \rfloor \quad \lfloor \mathbf{n}_2 \rfloor = \lfloor \mathbf{n} \rfloor}{(\mathbf{n}_1 - \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

$$\frac{\sigma(x) = \lfloor \mathbf{n} \rfloor}{(x, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

Example: Suppose $\sigma(x) = 10$ and $\sigma(y) = 42$.

$$(x + y, \sigma) \longrightarrow (\mathbf{10} + y, \sigma) \longrightarrow (\mathbf{10} + \mathbf{42}, \sigma) \longrightarrow (\mathbf{52}, \sigma)$$

Small-step SOS for boolean expressions

Recall

$$\begin{aligned} (\text{BoolExp}) \quad b &::= \mathbf{true} \mid \mathbf{false} \\ &\mid e = e \mid e < e \mid e > e \\ &\mid \neg b \mid b \wedge b \mid b \vee b \mid \dots \end{aligned}$$

We overload the symbol \longrightarrow .

Transitions for comparisons:

Small-step SOS for boolean expressions

Recall

$$\begin{array}{l} \text{(BoolExp)} \quad b ::= \mathbf{true} \mid \mathbf{false} \\ \quad \quad \quad \mid e = e \mid e < e \mid e > e \\ \quad \quad \quad \mid \neg b \mid b \wedge b \mid b \vee b \mid \dots \end{array}$$

We overload the symbol \longrightarrow .

Transitions for comparisons:

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 = e_2, \sigma) \longrightarrow (e'_1 = e_2, \sigma)}$$

$$\frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} = e_2, \sigma) \longrightarrow (\mathbf{n} = e'_2, \sigma)}$$

$$\frac{[n_1] \ [=] \ [n_2]}{(\mathbf{n}_1 = \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{true}, \sigma)}$$

$$\frac{\neg([n_1] \ [=] \ [n_2])}{(\mathbf{n}_1 = \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{false}, \sigma)}$$

Next: negation.

Small-step SOS for boolean expressions

Transitions for negation:

$$\frac{(b, \sigma) \longrightarrow (b', \sigma)}{(\neg b, \sigma) \longrightarrow (\neg b', \sigma)}$$

$$\overline{(\neg \mathbf{true}, \sigma) \longrightarrow (\mathbf{false}, \sigma)}$$

$$\overline{(\neg \mathbf{false}, \sigma) \longrightarrow (\mathbf{true}, \sigma)}$$

Next: conjunction.

Small-step SOS for boolean expressions

Transitions for conjunction:

$$\frac{(b_1, \sigma) \longrightarrow (b'_1, \sigma)}{(b_1 \wedge b_2, \sigma) \longrightarrow (b'_1 \wedge b_2, \sigma)}$$

$$\frac{(b_2, \sigma) \longrightarrow (b'_2, \sigma)}{(\mathbf{true} \wedge b_2, \sigma) \longrightarrow (\mathbf{true} \wedge b'_2, \sigma)} \quad \frac{(b_2, \sigma) \longrightarrow (b'_2, \sigma)}{(\mathbf{false} \wedge b_2, \sigma) \longrightarrow (\mathbf{false} \wedge b'_2, \sigma)}$$

$$\frac{}{(\mathbf{true} \wedge \mathbf{true}, \sigma) \longrightarrow (\mathbf{true}, \sigma)} \quad \frac{}{(\mathbf{true} \wedge \mathbf{false}, \sigma) \longrightarrow (\mathbf{false}, \sigma)}$$

$$\frac{}{(\mathbf{false} \wedge \mathbf{true}, \sigma) \longrightarrow (\mathbf{false}, \sigma)} \quad \frac{}{(\mathbf{false} \wedge \mathbf{false}, \sigma) \longrightarrow (\mathbf{false}, \sigma)}$$

Small-step SOS for boolean expressions

Different transitions for conjunction – short-circuit calculation:

$$\frac{(b_1, \sigma) \longrightarrow (b'_1, \sigma)}{(b_1 \wedge b_2, \sigma) \longrightarrow (b'_1 \wedge b_2, \sigma)}$$

$$\overline{(\mathbf{true} \wedge b_2, \sigma) \longrightarrow (b_2, \sigma)}$$

$$\overline{(\mathbf{false} \wedge b_2, \sigma) \longrightarrow (\mathbf{false}, \sigma)}$$

Remember that the order of evaluation is fixed by the small-step semantics.

Small-step SOS for statements

Recall

(Comm) $c ::=$ **skip**
 | $x := e$
 | $c ; c$
 | **if** b **then** c **else** c
 | **while** b **do** c

Next we define the semantics for statements. Again we will overload the symbol \longrightarrow .

The statement execution relation has the form of $(c, \sigma) \longrightarrow (c', \sigma')$ or $(c, \sigma) \longrightarrow \sigma'$.

Small-step SOS for **skip**

$$\overline{(\mathbf{skip}, \sigma) \longrightarrow \sigma}$$

Small-step SOS for assignment

$$\frac{(e, \sigma) \longrightarrow (e', \sigma)}{(x := e, \sigma) \longrightarrow (x := e', \sigma)}$$

$$\frac{}{(x := \mathbf{n}, \sigma) \longrightarrow \sigma\{x \rightsquigarrow \lfloor \mathbf{n} \rfloor\}}$$

Small-step SOS for assignment

$$\frac{(e, \sigma) \longrightarrow (e', \sigma)}{(x := e, \sigma) \longrightarrow (x := e', \sigma)} \qquad \frac{}{(x := \mathbf{n}, \sigma) \longrightarrow \sigma\{x \rightsquigarrow \lfloor \mathbf{n} \rfloor\}}$$

Example:

$$(x := \mathbf{10} + \mathbf{12}, \sigma) \longrightarrow (x := \mathbf{22}, \sigma) \longrightarrow \sigma\{x \rightsquigarrow 22\}$$

Another example:

$$(x := x + \mathbf{1}, \sigma') \longrightarrow (x := \mathbf{22} + \mathbf{1}, \sigma') \longrightarrow (x := \mathbf{23}, \sigma') \longrightarrow \sigma'\{x \rightsquigarrow 23\}$$

Next: sequential composition.

Small-step SOS for sequential composition

$$\frac{(c_0, \sigma) \longrightarrow (c'_0, \sigma')}{(c_0 ; c_1, \sigma) \longrightarrow (c'_0 ; c_1, \sigma')}$$

$$\frac{(c_0, \sigma) \longrightarrow \sigma'}{(c_0 ; c_1, \sigma) \longrightarrow (c_1, \sigma')}$$

Small-step SOS for sequential composition

$$\frac{(c_0, \sigma) \longrightarrow (c'_0, \sigma')}{(c_0 ; c_1, \sigma) \longrightarrow (c'_0 ; c_1, \sigma')}$$

$$\frac{(c_0, \sigma) \longrightarrow \sigma'}{(c_0 ; c_1, \sigma) \longrightarrow (c_1, \sigma')}$$

Example:

$$\begin{aligned} & (x := \mathbf{10} + \mathbf{12} ; x := x + \mathbf{1}, \sigma) \\ & \longrightarrow (x := \mathbf{22} ; x := x + \mathbf{1}, \sigma) \\ & \longrightarrow (x := x + \mathbf{1}, \sigma\{x \rightsquigarrow 22\}) \\ & \longrightarrow (x := \mathbf{22} + \mathbf{1}, \sigma\{x \rightsquigarrow 22\}) \\ & \longrightarrow (x := \mathbf{23}, \sigma\{x \rightsquigarrow 22\}) \\ & \longrightarrow \sigma\{x \rightsquigarrow 23\} \end{aligned}$$

Next: if-then-else.

Small-step SOS for **if**

$$\frac{(b, \sigma) \longrightarrow (b', \sigma)}{(\text{if } b \text{ then } c_0 \text{ else } c_1, \sigma) \longrightarrow (\text{if } b' \text{ then } c_0 \text{ else } c_1, \sigma)}$$

$$\frac{}{(\text{if true then } c_0 \text{ else } c_1, \sigma) \longrightarrow (c_0, \sigma)}$$

$$\frac{}{(\text{if false then } c_0 \text{ else } c_1, \sigma) \longrightarrow (c_1, \sigma)}$$

Incorrect semantics for **while**

$$\frac{(b, \sigma) \longrightarrow (b', \sigma)}{(\mathbf{while\ } b \ \mathbf{do\ } c, \sigma) \longrightarrow (\mathbf{while\ } b' \ \mathbf{do\ } c, \sigma)}$$

$$\overline{(\mathbf{while\ false\ do\ } c, \sigma) \longrightarrow \sigma}$$

$$\overline{(\mathbf{while\ true\ do\ } c, \sigma) \longrightarrow ?}$$

Incorrect semantics for **while**

$$\frac{(b, \sigma) \longrightarrow (b', \sigma)}{(\mathbf{while} \ b \ \mathbf{do} \ c, \sigma) \longrightarrow (\mathbf{while} \ b' \ \mathbf{do} \ c, \sigma)}$$

$$\overline{(\mathbf{while} \ \mathbf{false} \ \mathbf{do} \ c, \sigma) \longrightarrow \sigma}$$

$$\overline{(\mathbf{while} \ \mathbf{true} \ \mathbf{do} \ c, \sigma) \longrightarrow ?}$$

Actually we want to evaluate b every time we go through the loop. So, when we evaluate it the first time, it is vital that we don't throw away the original b .

In fact we can give a single rule for **while** using the **if** statement.

Small-step SOS for **while**

(while b do c, σ) \longrightarrow (if b then (c ; while b do c) else skip, σ)

Zero-or-multiple steps

We define \longrightarrow^* as the *reflexive transitive closure* of \longrightarrow .

For instance,

$$\frac{}{(c, \sigma) \longrightarrow^* (c, \sigma)} \quad \frac{(c, \sigma) \longrightarrow (c', \sigma') \quad (c', \sigma') \longrightarrow^* (c'', \sigma'')}{(c, \sigma) \longrightarrow^* (c'', \sigma'')}$$

n -step transitions:

$$\frac{}{(c, \sigma) \longrightarrow^0 (c, \sigma)} \quad \frac{(c, \sigma) \longrightarrow (c', \sigma') \quad (c', \sigma') \longrightarrow^n (c'', \sigma'')}{(c, \sigma) \longrightarrow^{n+1} (c'', \sigma'')}$$

We have $(c, \sigma) \longrightarrow^* (c', \sigma')$ iff $\exists n. (c, \sigma) \longrightarrow^n (c', \sigma')$.

What about $(c, \sigma) \longrightarrow^* \sigma'$?

Example

Compute the factorial of x and store the result in variable a :

$$c \stackrel{\text{def}}{=} \begin{array}{l} y := x; a := 1; \\ \mathbf{while} (y > 0) \mathbf{do} \\ \quad (a := a \times y; \\ \quad \quad y := y - 1) \end{array}$$

Let $\sigma = \{x \rightsquigarrow 3, y \rightsquigarrow 2, a \rightsquigarrow 9\}$. It should be the case that

$$(c, \sigma) \longrightarrow^* \sigma'$$

where $\sigma' = \{x \rightsquigarrow 3, y \rightsquigarrow 0, a \rightsquigarrow 6\}$.

Let's check that it is correct.

Remark

- ▶ As you can see, this kind of calculation is horrible to do by hand. It can, however, be automated to give a simple *interpreter* for the language, based directly on the semantics.
- ▶ It is also formal and precise, with no disputes about what should happen at any given point.
- ▶ Finally, it does compute the right answer!

Some facts about \longrightarrow

Theorem (Determinism)

For all $c, \sigma, c', \sigma', c'', \sigma''$, if $(c, \sigma) \longrightarrow (c', \sigma')$ and $(c, \sigma) \longrightarrow (c'', \sigma'')$, then $(c', \sigma') = (c'', \sigma'')$.

Corollary (Confluence)

For all $c, \sigma, c', \sigma', c'', \sigma''$, if $(c, \sigma) \longrightarrow^ (c', \sigma')$ and $(c, \sigma) \longrightarrow^* (c'', \sigma'')$, then there exist c''' and σ''' such that $(c', \sigma') \longrightarrow^* (c''', \sigma''')$ and $(c'', \sigma'') \longrightarrow^* (c''', \sigma''')$.*

Analogous results hold for the transitions on (e, σ) and (b, σ) .

Some facts about \longrightarrow

Normalization: There are no infinite sequences of configurations $(e_1, \sigma_1), (e_2, \sigma_2), \dots$ such that, for all i , $(e_i, \sigma_i) \longrightarrow (e_{i+1}, \sigma_{i+1})$. That is, every evaluation path eventually reaches a *normal form*.

Normal forms:

- ▶ For expressions, the normal forms are (\mathbf{n}, σ) for numeral \mathbf{n} .
- ▶ For booleans, the normal forms are (\mathbf{true}, σ) and (\mathbf{false}, σ) .

Facts: The transition relations on (e, σ) and (b, σ) are normalizing.

But!! The transition relation on (c, σ) is *not* normalizing.

Some facts about \longrightarrow

The transition relation on (c, σ) is *not* normalizing.

Specifically, we can have infinite loops. For example, the program **while true do skip** loops forever.

Theorem

For any state σ , there is no σ' such that
(while true do skip, σ) \longrightarrow^* σ'

Proof?

Next: we will see some variations of the current small-step semantics.

Note when we modify the semantics, we define a different language.

Variation I

Assignment:

$$\frac{\llbracket e \rrbracket_{intexp} \sigma = n}{(x := e, \sigma) \longrightarrow \sigma\{x \rightsquigarrow n\}}$$

Here

$$\llbracket e \rrbracket_{intexp} \sigma = n \text{ iff } (e, \sigma) \longrightarrow^* (\mathbf{n}, \sigma) \text{ and } n = \lfloor \mathbf{n} \rfloor$$

Compared to the original version:

$$\frac{(e, \sigma) \longrightarrow (e', \sigma)}{(x := e, \sigma) \longrightarrow (x := e', \sigma)} \qquad \frac{}{(x := \mathbf{n}, \sigma) \longrightarrow \sigma\{x \rightsquigarrow \lfloor \mathbf{n} \rfloor\}}$$

Earlier example: $(x := \mathbf{10 + 12}, \sigma) \longrightarrow (x := \mathbf{22}, \sigma) \longrightarrow \sigma\{x \rightsquigarrow 22\}$

Variation I

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \text{true}}{(\text{if } b \text{ then } c_0 \text{ else } c_1, \sigma) \longrightarrow (c_0, \sigma)}$$

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \text{false}}{(\text{if } b \text{ then } c_0 \text{ else } c_1, \sigma) \longrightarrow (c_1, \sigma)}$$

Compared to the original version:

$$\frac{(b, \sigma) \longrightarrow (b', \sigma)}{(\text{if } b \text{ then } c_0 \text{ else } c_1, \sigma) \longrightarrow (\text{if } b' \text{ then } c_0 \text{ else } c_1, \sigma)}$$

$$\frac{}{(\text{if true then } c_0 \text{ else } c_1, \sigma) \longrightarrow (c_0, \sigma)}$$

$$\frac{}{(\text{if false then } c_0 \text{ else } c_1, \sigma) \longrightarrow (c_1, \sigma)}$$

Variation I

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \text{true}}{(\mathbf{while } b \mathbf{ do } c, \sigma) \longrightarrow (c ; \mathbf{while } b \mathbf{ do } c, \sigma)}$$

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \text{false}}{(\mathbf{while } b \mathbf{ do } c, \sigma) \longrightarrow \sigma}$$

Compared to the original version:

$$\overline{(\mathbf{while } b \mathbf{ do } c, \sigma) \longrightarrow (\mathbf{if } b \mathbf{ then } (c ; \mathbf{while } b \mathbf{ do } c) \mathbf{ else skip}, \sigma)}$$

Variation II

Assignment:

$$\frac{\llbracket e \rrbracket_{intexp} \sigma = n}{(x := e, \sigma) \longrightarrow (\mathbf{skip}, \sigma\{x \rightsquigarrow n\})}$$

Here **skip** is overloaded as a flag for termination.
(So there is no rule for (\mathbf{skip}, σ)).

Variation II

Assignment:

$$\frac{\llbracket e \rrbracket_{intexp} \sigma = n}{(x := e, \sigma) \longrightarrow (\mathbf{skip}, \sigma\{x \rightsquigarrow n\})}$$

Here **skip** is overloaded as a flag for termination.
(So there is no rule for (\mathbf{skip}, σ)).

Sequential composition:

$$\frac{(c_0, \sigma) \longrightarrow (c'_0, \sigma')}{(c_0 ; c_1, \sigma) \longrightarrow (c'_0 ; c_1, \sigma')}$$

$$\frac{}{(\mathbf{skip} ; c_1, \sigma) \longrightarrow (c_1, \sigma)}$$

Variation II

$$\frac{\llbracket e \rrbracket_{intexp} \sigma = n}{(x := e, \sigma) \longrightarrow (\mathbf{skip}, \sigma\{x \rightsquigarrow n\})}$$

$$\frac{(c_0, \sigma) \longrightarrow (c'_0, \sigma')}{(c_0 ; c_1, \sigma) \longrightarrow (c'_0 ; c_1, \sigma')}$$

$$\frac{}{(\mathbf{skip} ; c_1, \sigma) \longrightarrow (c_1, \sigma)}$$

One more identity step is introduced after every command:
consider $x := x + 1 ; y := y + 2$.

Compared to the earlier rules:

$$\frac{\llbracket e \rrbracket_{intexp} \sigma = n}{(x := e, \sigma) \longrightarrow \sigma\{x \rightsquigarrow n\}}$$

$$\frac{}{(\mathbf{skip}, \sigma) \longrightarrow \sigma}$$

$$\frac{(c_0, \sigma) \longrightarrow (c'_0, \sigma')}{(c_0 ; c_1, \sigma) \longrightarrow (c'_0 ; c_1, \sigma')}$$

$$\frac{(c_0, \sigma) \longrightarrow \sigma'}{(c_0 ; c_1, \sigma) \longrightarrow (c_1, \sigma')}$$

Variation II

Why?

Sometimes it is more convenient.

The earlier versions have two forms of transitions for statements.

$$(c, \sigma) \longrightarrow (c', \sigma') \qquad (c, \sigma) \longrightarrow \sigma'$$

When defining or proving properties of \longrightarrow , we need to consider both cases.

But, this is not a big deal.

Variation II – all rules

$$\frac{\llbracket e \rrbracket_{\text{intexp}} \sigma = n}{(x := e, \sigma) \longrightarrow (\mathbf{skip}, \sigma\{x \rightsquigarrow n\})}$$

$$\frac{(c_0, \sigma) \longrightarrow (c'_0, \sigma')}{(c_0 ; c_1, \sigma) \longrightarrow (c'_0 ; c_1, \sigma')} \qquad \frac{}{(\mathbf{skip} ; c_1, \sigma) \longrightarrow (c_1, \sigma)}$$

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \text{true}}{(\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma) \longrightarrow (c_0, \sigma)}$$

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \text{false}}{(\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma) \longrightarrow (c_1, \sigma)}$$

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \text{true}}{(\mathbf{while } b \mathbf{ do } c, \sigma) \longrightarrow (c ; \mathbf{while } b \mathbf{ do } c, \sigma)}$$

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \text{false}}{(\mathbf{while } b \mathbf{ do } c, \sigma) \longrightarrow (\mathbf{skip}, \sigma)}$$

Next: we will extend “Variation II” with the following language features.

- ▶ Going wrong
- ▶ Local variable declaration
- ▶ Dynamically-allocated data

Going wrong

We introduce another configuration: **abort**.

The following will lead to **abort**:

- ▶ Divide by 0
- ▶ Access non-existing data
- ▶ ...

abort cannot step anymore.

Going wrong

Expressions:

$$e ::= \dots \mid e/e$$

Expression evaluation:

$$\frac{\mathbf{n}_2 \neq \mathbf{0} \quad \llbracket \mathbf{n}_1 \rrbracket \llbracket / \rrbracket \llbracket \mathbf{n}_2 \rrbracket = \llbracket \mathbf{n} \rrbracket}{(\mathbf{n}_1/\mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

$$\frac{}{(\mathbf{n}_1/\mathbf{0}, \sigma) \longrightarrow \mathbf{abort}}$$

Going wrong

Assignment:

$$\frac{\llbracket e \rrbracket_{intexp} \sigma = n}{(x := e, \sigma) \longrightarrow (\mathbf{skip}, \sigma\{x \rightsquigarrow n\})}$$

$$\frac{\llbracket e \rrbracket_{intexp} \sigma = \perp}{(x := e, \sigma) \longrightarrow \mathbf{abort}}$$

Here

$$\llbracket e \rrbracket_{intexp} \sigma = n \quad \text{iff} \quad (e, \sigma) \longrightarrow^* (\mathbf{n}, \sigma) \text{ and } n = \lfloor \mathbf{n} \rfloor$$

$$\llbracket e \rrbracket_{intexp} \sigma = \perp \quad \text{iff} \quad (e, \sigma) \longrightarrow^* \mathbf{abort}$$

Going wrong

Add new rules:

$$\frac{(c_0, \sigma) \longrightarrow \mathbf{abort}}{(c_0 ; c_1, \sigma) \longrightarrow \mathbf{abort}}$$

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \perp}{(\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma) \longrightarrow \mathbf{abort}}$$

$$\frac{\llbracket b \rrbracket_{\text{boolexp}} \sigma = \perp}{(\mathbf{while } b \mathbf{ do } c, \sigma) \longrightarrow \mathbf{abort}}$$

Going wrong

We distinguish “going wrong” from “getting stuck”.

We say c *gets stuck* at the state σ iff there's no c', σ' such that $(c, \sigma) \longrightarrow (c', \sigma')$.

In the semantics “Version II”, **skip** gets stuck at any state.

Note both notions are language-dependent.

Next extension: local variable declaration.

Local variable declaration

Statements:

$c ::= \dots \mid \mathbf{newvar} \ x := e \ \mathbf{in} \ c$

An unsatisfactory attempt:

$$\frac{\sigma \ x = \lfloor \mathbf{n} \rfloor}{(\mathbf{newvar} \ x := e \ \mathbf{in} \ c, \sigma) \longrightarrow (x := e ; c ; x := \mathbf{n}, \sigma)}$$

Unsatisfactory because the value of local variable x could be exposed to external observers while c is executing.

This is a problem when we have concurrency.

Semantics for **newvar**

Solution (due to Eugene Fink):

$$\frac{n = \llbracket e \rrbracket_{intexp} \sigma \quad (c, \sigma\{x \rightsquigarrow n\}) \longrightarrow (c', \sigma') \quad \sigma' x = \lfloor n' \rfloor}{(\mathbf{newvar} \ x := e \ \mathbf{in} \ c, \ \sigma) \longrightarrow (\mathbf{newvar} \ x := n' \ \mathbf{in} \ c', \ \sigma'\{x \rightsquigarrow \sigma x\})}$$

$$\overline{(\mathbf{newvar} \ x := e \ \mathbf{in} \ \mathbf{skip}, \ \sigma) \longrightarrow (\mathbf{skip}, \ \sigma)}$$

$$\frac{\llbracket e \rrbracket_{intexp} \sigma = \perp}{(\mathbf{newvar} \ x := e \ \mathbf{in} \ c, \ \sigma) \longrightarrow \mathbf{abort}}$$

$$\frac{n = \llbracket e \rrbracket_{intexp} \sigma \quad (c, \sigma\{x \rightsquigarrow n\}) \longrightarrow \mathbf{abort}}{(\mathbf{newvar} \ x := e \ \mathbf{in} \ c, \ \sigma) \longrightarrow \mathbf{abort}}$$

Heap for dynamically-allocated data

(States) $\sigma ::= (s, h)$

(Stores) $s \in \text{Var} \rightarrow \text{Values}$

(Heaps) $h \in \text{Loc} \rightarrow_{\text{fin}} \text{Values}$

(Values) $v \in \text{Int} \cup \text{Bool} \cup \text{Loc}$

Here \rightarrow_{fin} represents a partial mapping.

A simple language with heap manipulation

Statements:

c	$::=$	\dots	
		$x := \mathbf{alloc}(e)$	allocation
		$y := [x]$	lookup
		$[x] := e$	mutation
		$\mathbf{free}(x)$	deallocation

Configurations: $(c, (s, h))$

Operational semantics for **alloc**

$$\frac{l \notin \text{dom}(h) \quad \llbracket e \rrbracket_{\text{intexp}} s = n}{(x := \mathbf{alloc}(e), (s, h)) \longrightarrow (\mathbf{skip}, (s\{x \rightsquigarrow l\}, h \uplus \{l \rightsquigarrow n\}))}$$

Operational semantics for **free**

$$\frac{s x = l \quad l \in \text{dom}(h)}{(\mathbf{free}(x), (s, h)) \longrightarrow (\mathbf{skip}, (s, h \setminus \{l\}))}$$

Operational semantics for lookup and mutation

$$\frac{s x = l \quad h l = n}{(y := [x], (s, h)) \longrightarrow (\mathbf{skip}, (s\{y \rightsquigarrow n\}, h))}$$

$$\frac{s x = l \quad l \in \text{dom}(h) \quad \llbracket e \rrbracket_{\text{intexp}} s = n}{([x] := e, (s, h)) \longrightarrow (\mathbf{skip}, (s, h\{l \rightsquigarrow n\}))}$$

Summary of small-step structural operational semantics

Form of transition rules:

$$\frac{P_1 \quad \dots \quad P_n}{(c, \sigma) \longrightarrow (c', \sigma')}$$

P_1, \dots, P_n are the conditions that must hold for the transition to go through. Also called the premises for the rule. They could be

- ▶ Other transitions corresponding to the sub-terms.
- ▶ Side conditions: predicates that must be true.

Next: small-step contextual semantics (a.k.a. reduction semantics)

Review of small-step SOS

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 + e_2, \sigma) \longrightarrow (e'_1 + e_2, \sigma)}$$

$$\frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} + e_2, \sigma) \longrightarrow (\mathbf{n} + e'_2, \sigma)}$$

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 - e_2, \sigma) \longrightarrow (e'_1 - e_2, \sigma)}$$

$$\frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} - e_2, \sigma) \longrightarrow (\mathbf{n} - e'_2, \sigma)}$$

$$\frac{\lfloor \mathbf{n}_1 \rfloor \quad \lfloor + \rfloor \quad \lfloor \mathbf{n}_2 \rfloor = \lfloor \mathbf{n} \rfloor}{(\mathbf{n}_1 + \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

$$\frac{\lfloor \mathbf{n}_1 \rfloor \quad \lfloor - \rfloor \quad \lfloor \mathbf{n}_2 \rfloor = \lfloor \mathbf{n} \rfloor}{(\mathbf{n}_1 - \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

$$\frac{\sigma(x) = \lfloor \mathbf{n} \rfloor}{(x, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

Observe that the top four rules look similar. We may combine them into a **single** rule:

$$\frac{(e, \sigma) \longrightarrow (e', \sigma)}{(\mathcal{E}[e], \sigma) \longrightarrow (\mathcal{E}[e'], \sigma)}$$

Here $\mathcal{E} ::= [] + e \mid [] - e \mid \mathbf{n} + [] \mid \mathbf{n} - []$

A quick feel of contextual semantics

$$\frac{(r, \sigma) \rightarrow (e', \sigma)}{(\mathcal{E}[r], \sigma) \rightarrow (\mathcal{E}[e'], \sigma)}$$

Here $r ::= x \mid \mathbf{n} + \mathbf{n} \mid \mathbf{n} - \mathbf{n}$

$$\frac{\llbracket \mathbf{n}_1 \rrbracket \llbracket + \rrbracket \llbracket \mathbf{n}_2 \rrbracket = \llbracket \mathbf{n} \rrbracket}{(\mathbf{n}_1 + \mathbf{n}_2, \sigma) \rightarrow (\mathbf{n}, \sigma)} \quad \frac{\llbracket \mathbf{n}_1 \rrbracket \llbracket - \rrbracket \llbracket \mathbf{n}_2 \rrbracket = \llbracket \mathbf{n} \rrbracket}{(\mathbf{n}_1 - \mathbf{n}_2, \sigma) \rightarrow (\mathbf{n}, \sigma)} \quad \frac{\sigma(x) = \llbracket \mathbf{n} \rrbracket}{(x, \sigma) \rightarrow (\mathbf{n}, \sigma)}$$

And $\mathcal{E} ::= [] \mid \mathcal{E} + e \mid \mathcal{E} - e \mid \mathbf{n} + \mathcal{E} \mid \mathbf{n} - \mathcal{E}$

r is called a **redex**.

\mathcal{E} is called an **evaluation context** (or a reduction context).

Contextual semantics

An alternative presentation of small-step operational semantics using **redex** and **evaluation contexts**.

Specified in two parts:

- ▶ What evaluation rules (for redex) to apply?
 - ▶ What is an *atomic reduction* step?
- ▶ Where can we apply them?
 - ▶ Where should we apply the next atomic reduction step?

Redex

A **redex** is a syntactic expression or command that can be reduced (transformed) in one atomic step.

For brevity, below we mix expression and command redexes.

```
(Redex)  r ::= x
          |  n + n
          |  x := n
          |  skip ; c
          |  if true then c else c
          |  if false then c else c
          |  while b do c
          |  ...
```

Example: $(1 + 3) + 2$ is not a redex, but $1 + 3$ is.

Local reduction rules

One rule for each redex: $(r, \sigma) \longrightarrow (t, \sigma')$.

$$\frac{\sigma(x) = \lfloor \mathbf{n} \rfloor}{(x, \sigma) \longrightarrow (\mathbf{n}, \sigma)} \qquad \frac{\lfloor \mathbf{n}_1 \rfloor \lfloor + \rfloor \lfloor \mathbf{n}_2 \rfloor = \lfloor \mathbf{n} \rfloor}{(\mathbf{n}_1 + \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

$$\frac{}{(x := \mathbf{n}, \sigma) \longrightarrow (\mathbf{skip}, \sigma\{x \rightsquigarrow \lfloor \mathbf{n} \rfloor\})}$$

$$\frac{}{(\mathbf{skip}; c_1, \sigma) \longrightarrow (c_1, \sigma)}$$

$$\frac{}{(\mathbf{if true then } c_0 \mathbf{ else } c_1, \sigma) \longrightarrow (c_0, \sigma)}$$

$$\frac{}{(\mathbf{while } b \mathbf{ do } c, \sigma) \longrightarrow (\mathbf{if } b \mathbf{ then } (c; \mathbf{while } b \mathbf{ do } c) \mathbf{ else skip}, \sigma)}$$

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, σ)** \longrightarrow **(10, σ)**

Next: global reduction rules

Consider

- ▶ **($x := 1 + (2 + 8), \sigma$)**
- ▶ **(while false do $x := 1 + (2 + 8), \sigma$)**

Should we reduce **2 + 8** in these cases?

Evaluation contexts

An evaluation context is a term with a “hole” in the place of a sub-term

- ▶ Location of the hole indicates the next place for evaluation
- ▶ If \mathcal{E} is a context, then $\mathcal{E}[r]$ is the expression obtained by replacing redex r for the hole in context \mathcal{E}
- ▶ Now, if $(r, \sigma) \longrightarrow (t, \sigma')$, then $(\mathcal{E}[r], \sigma) \longrightarrow (\mathcal{E}[t], \sigma')$.

Example: $x := \mathbf{1} + []$

- ▶ Filling hole with $\mathbf{2} + \mathbf{8}$ yields $\mathcal{E}[\mathbf{2} + \mathbf{8}] = (x := \mathbf{1} + (\mathbf{2} + \mathbf{8}))$
- ▶ Or filling with $\mathbf{10}$ yields $\mathcal{E}[\mathbf{10}] = (x := \mathbf{1} + \mathbf{10})$

Evaluation contexts

(Ctx) $\mathcal{E} ::=$ $[\]$
| $\mathcal{E} + e$
| $\mathbf{n} + \mathcal{E}$
| $x := \mathcal{E}$
| $\mathcal{E}; c$
| **if** \mathcal{E} **then** c **else** c
| ...

Examples:

- ▶ $x := \mathbf{1} + [\]$
- ▶ NOT: **while false do** $x := \mathbf{1} + [\]$
- ▶ NOT: **if** b **then** c **else** $[\]$

Evaluation contexts

- ▶ \mathcal{E} has exactly one hole
- ▶ \mathcal{E} uniquely identifies the next redex to be evaluated

Consider $e = e_1 + e_2$ and its decomposition as $\mathcal{E}[r]$.

- ▶ If $e_1 = \mathbf{n}_1$ and $e_2 = \mathbf{n}_2$, then $r = \mathbf{n}_1 + \mathbf{n}_2$ and $\mathcal{E} = []$
- ▶ If $e_1 = \mathbf{n}_1$ and e_2 is not \mathbf{n}_2 , then $e_2 = \mathcal{E}_2[r]$ and $\mathcal{E} = \mathbf{n}_1 + \mathcal{E}_2$
- ▶ If e_1 is not \mathbf{n}_1 , then $e_1 = \mathcal{E}_1[r]$ and $\mathcal{E} = \mathcal{E}_1 + e_2$

In the last two cases the decomposition is done recursively.

In each case the solution is unique.

Evaluation contexts

Consider $c = (c_1 ; c_2)$ and its decomposition as $\mathcal{E}[r]$.

- ▶ If $c_1 = \mathbf{skip}$, then $r = (\mathbf{skip} ; c_2)$ and $\mathcal{E} = []$
- ▶ If $c_1 \neq \mathbf{skip}$, then $c_1 = \mathcal{E}_1[r]$ and $\mathcal{E} = (\mathcal{E}_1 ; c_2)$

Consider $c = (\mathbf{if } b \mathbf{ then } c_1 \mathbf{ else } c_2)$ and its decomposition as $\mathcal{E}[r]$.

- ▶ If $b = \mathbf{true}$ or $b = \mathbf{false}$, then $r = (\mathbf{if } b \mathbf{ then } c_1 \mathbf{ else } c_2)$ and $\mathcal{E} = []$
- ▶ Otherwise, $b = \mathcal{E}_0[r]$ and $\mathcal{E} = (\mathbf{if } \mathcal{E}_0 \mathbf{ then } c_1 \mathbf{ else } c_2)$

Evaluation contexts

Decomposition theorem:

- ▶ If $c \neq \mathbf{skip}$, then there exist unique \mathcal{E} and r such that $c = \mathcal{E}[r]$
- ▶ If $e \neq \mathbf{n}$, then there exist unique \mathcal{E} and r such that $e = \mathcal{E}[r]$

“exists” \Rightarrow progress

“unique” \Rightarrow determinism

Global reduction rule

General idea of the contextual semantics:

- ▶ Decompose the current term into
 - ▶ the next redex r
 - ▶ and an evaluation context \mathcal{E} (the remaining program).
- ▶ Reduce the redex r to some other term t .
- ▶ Put t back into the original context, yielding $\mathcal{E}[t]$.

Formalized as a small-step rule:

$$\frac{(r, \sigma) \longrightarrow (t, \sigma')}{(\mathcal{E}[r], \sigma) \longrightarrow (\mathcal{E}[t], \sigma')}$$

Contextual semantics rules =

Global reduction rule + Local reduction rules for individual r

Examples

$x := \mathbf{1} + (\mathbf{2} + \mathbf{8})$

- ▶ Decompose it into an evaluation context \mathcal{E} and a redex r
 - ▶ $r = (\mathbf{2} + \mathbf{8})$
 - ▶ $\mathcal{E} = (x := \mathbf{1} + [\])$
 - ▶ $\mathcal{E}[r] = (x := \mathbf{1} + (\mathbf{2} + \mathbf{8}))$ (original command)
- ▶ By local reduction rule, $(\mathbf{2} + \mathbf{8}, \sigma) \longrightarrow (\mathbf{10}, \sigma)$
- ▶ By global reduction rule, $(\mathcal{E}[\mathbf{2} + \mathbf{8}], \sigma) \longrightarrow (\mathcal{E}[\mathbf{10}], \sigma)$;
or equivalently $(x := \mathbf{1} + (\mathbf{2} + \mathbf{8}), \sigma) \longrightarrow (x := \mathbf{1} + \mathbf{10}, \sigma)$

Examples

$x := 1; x := x + 1$ in the initial state $\{x \rightsquigarrow 0\}$

Configuration	Redex	Context
$(x := 1; x := x + 1, \{x \rightsquigarrow 0\})$	$x := 1$	$[]; x := x + 1$
$(\mathbf{skip}; x := x + 1, \{x \rightsquigarrow 1\})$	$\mathbf{skip}; x := x + 1$	$[]$
$(x := x + 1, \{x \rightsquigarrow 1\})$	x	$x := [] + 1$
$(x := 1 + 1, \{x \rightsquigarrow 1\})$	$1 + 1$	$x := []$
$(x := 2, \{x \rightsquigarrow 1\})$	$x := 2$	$[]$
$(\mathbf{skip}, \{x \rightsquigarrow 2\})$		

Contextual semantics for boolean expressions

Normal evaluation of \wedge :

define the following contexts, redexes, and local rules

$$\mathcal{E} ::= \dots \mid \mathcal{E} \wedge b \mid \mathbf{true} \wedge \mathcal{E} \mid \mathbf{false} \wedge \mathcal{E}$$
$$r ::= \dots \mid \mathbf{true} \wedge \mathbf{true} \mid \mathbf{true} \wedge \mathbf{false} \mid \mathbf{false} \wedge \mathbf{true} \mid \mathbf{false} \wedge \mathbf{false}$$
$$(\mathbf{true} \wedge \mathbf{true}, \sigma) \longrightarrow (\mathbf{true}, \sigma) \quad \dots$$

Contextual semantics for boolean expressions

Normal evaluation of \wedge :

define the following contexts, redexes, and local rules

$$\mathcal{E} ::= \dots \mid \mathcal{E} \wedge b \mid \mathbf{true} \wedge \mathcal{E} \mid \mathbf{false} \wedge \mathcal{E}$$

$$r ::= \dots \mid \mathbf{true} \wedge \mathbf{true} \mid \mathbf{true} \wedge \mathbf{false} \mid \mathbf{false} \wedge \mathbf{true} \mid \mathbf{false} \wedge \mathbf{false}$$

$$(\mathbf{true} \wedge \mathbf{true}, \sigma) \longrightarrow (\mathbf{true}, \sigma) \quad \dots$$

Short-circuit evaluation of \wedge :

define the following contexts, redexes, and local rules

$$\mathcal{E} ::= \dots \mid \mathcal{E} \wedge b$$

$$r ::= \dots \mid \mathbf{true} \wedge b \mid \mathbf{false} \wedge b$$

$$(\mathbf{true} \wedge b, \sigma) \longrightarrow (b, \sigma) \quad (\mathbf{false} \wedge b, \sigma) \longrightarrow (\mathbf{false}, \sigma)$$

The local reduction kicks in before b is evaluated.

Summary of contextual semantics

Think of a hole as representing a program counter

The rules for advancing holes are non-trivial

- ▶ Must decompose entire command at every step
- ▶ So, inefficient to implement contextual semantics directly

Major advantage of contextual semantics is that it allows a mix of global and local reduction rules

- ▶ Global rules indicate next redex to be evaluated (defined by the grammar of the context)
- ▶ Local rules indicate how to perform the reduction one for each redex

Big-Step Semantics

Different approaches of operational semantics:

- ▶ We have discussed **small-step semantics**, which describes each *single step* of the execution.
 - ▶ Structural operational semantics
 - ▶ Contextual semantics

$$(c, \sigma) \longrightarrow (c', \sigma')$$

$$(e, \sigma) \longrightarrow (e', \sigma)$$

- ▶ Next: **big-step semantics** (a.k.a. natural semantics), which describes the *overall result* of the execution

$$(c, \sigma) \Downarrow \sigma'$$

$$(e, \sigma) \Downarrow n$$

Big-Step Semantics

$$\frac{}{(n, \sigma) \Downarrow [n]} \qquad \frac{\sigma x = n}{(x, \sigma) \Downarrow n}$$

$$\frac{(e_1, \sigma) \Downarrow n_1 \quad (e_2, \sigma) \Downarrow n_2}{(e_1 + e_2, \sigma) \Downarrow n_1 [+] n_2}$$

The last rule can be generalized to:

$$\frac{(e_1, \sigma) \Downarrow n_1 \quad (e_2, \sigma) \Downarrow n_2}{(e_1 \mathbf{op} e_2, \sigma) \Downarrow n_1 [\mathbf{op}] n_2}$$

Big-Step Semantics

$$\frac{(e_1, \sigma) \Downarrow n_1 \quad (e_2, \sigma) \Downarrow n_2}{(e_1 \text{ op } e_2, \sigma) \Downarrow n_1 [\text{op}] n_2}$$

Compared to small-step SOS:

$$\frac{(e_1, \sigma) \longrightarrow (e'_1, \sigma)}{(e_1 \text{ op } e_2, \sigma) \longrightarrow (e'_1 \text{ op } e_2, \sigma)} \quad \frac{(e_2, \sigma) \longrightarrow (e'_2, \sigma)}{(\mathbf{n} \text{ op } e_2, \sigma) \longrightarrow (\mathbf{n} \text{ op } e'_2, \sigma)}$$

$$\frac{[\mathbf{n}_1] \ [\text{op}] \ [\mathbf{n}_2] = [\mathbf{n}]}{(\mathbf{n}_1 \text{ op } \mathbf{n}_2, \sigma) \longrightarrow (\mathbf{n}, \sigma)}$$

Examples

$$\frac{\overline{(\mathbf{3}, \sigma) \Downarrow 3} \quad \frac{\overline{(\mathbf{2}, \sigma) \Downarrow 2} \quad \overline{(\mathbf{1}, \sigma) \Downarrow 1}}{\overline{(\mathbf{2} + \mathbf{1}, \sigma) \Downarrow 3}}}{\overline{(\mathbf{3} + (\mathbf{2} + \mathbf{1}), \sigma) \Downarrow 6}}$$

Examples

$$\frac{\frac{\overline{(\mathbf{3}, \sigma) \Downarrow 3}}{\quad} \quad \frac{\frac{\overline{(\mathbf{2}, \sigma) \Downarrow 2} \quad \overline{(\mathbf{1}, \sigma) \Downarrow 1}}{\overline{(\mathbf{2} + \mathbf{1}, \sigma) \Downarrow 3}}}{\overline{(\mathbf{3} + (\mathbf{2} + \mathbf{1}), \sigma) \Downarrow 6}}}{\quad}}$$

Compared to small-step version:

$$(\mathbf{3} + (\mathbf{2} + \mathbf{1}), \sigma) \longrightarrow (\mathbf{3} + \mathbf{3}, \sigma) \longrightarrow (\mathbf{6}, \sigma)$$

Big-step semantics more closely models a recursive interpreter.

Examples

$$\frac{\frac{\overline{(4, \sigma) \Downarrow 4} \quad \overline{(3, \sigma) \Downarrow 3}}{(4 + 3, \sigma) \Downarrow 7} \quad \frac{\overline{(2, \sigma) \Downarrow 2} \quad \overline{(1, \sigma) \Downarrow 1}}{(2 + 1, \sigma) \Downarrow 3}}{((4 + 3) + (2 + 1), \sigma) \Downarrow 10}$$

Compared to small-step version:

$$((4 + 3) + (2 + 1), \sigma) \longrightarrow (7 + (2 + 1), \sigma) \longrightarrow (7 + 3, \sigma) \longrightarrow (10, \sigma)$$

The “boring” rules of small-step semantics specify the order of evaluation.

Some facts about \Downarrow

Theorem (Determinism)

For all e, σ, n, n' , if $(e, \sigma) \Downarrow n$ and $(e, \sigma) \Downarrow n'$, then $n = n'$.

Theorem (Totality)

For all e, σ , there exists n such that $(e, \sigma) \Downarrow n$.

Theorem (Equivalence to small-step semantics)

$(e, \sigma) \Downarrow \lfloor \mathbf{n} \rfloor$ iff $(e, \sigma) \longrightarrow^* (\mathbf{n}, \sigma)$

Big-step semantics for boolean expressions

$$\overline{(\mathbf{true}, \sigma) \Downarrow \mathit{true}}$$

$$\overline{(\mathbf{false}, \sigma) \Downarrow \mathit{false}}$$

Normal evaluation of \wedge :

$$\frac{(b_1, \sigma) \Downarrow \mathit{false} \quad (b_2, \sigma) \Downarrow \mathit{true}}{(b_1 \wedge b_2, \sigma) \Downarrow \mathit{false}} \quad \dots$$

Big-step semantics for boolean expressions

$$\overline{(\mathbf{true}, \sigma) \Downarrow \mathit{true}}$$

$$\overline{(\mathbf{false}, \sigma) \Downarrow \mathit{false}}$$

Normal evaluation of \wedge :

$$\frac{(b_1, \sigma) \Downarrow \mathit{false} \quad (b_2, \sigma) \Downarrow \mathit{true}}{(b_1 \wedge b_2, \sigma) \Downarrow \mathit{false}} \quad \dots$$

Short-circuit evaluation of \wedge :

$$\frac{(b_1, \sigma) \Downarrow \mathit{false}}{(b_1 \wedge b_2, \sigma) \Downarrow \mathit{false}} \quad \dots$$

Big-step semantics for statements

$$\frac{(e, \sigma) \Downarrow n}{(x := e, \sigma) \Downarrow \sigma\{x \rightsquigarrow n\}}$$

$$\frac{}{(\mathbf{skip}, \sigma) \Downarrow \sigma}$$

Big-step semantics for statements

$$\frac{(e, \sigma) \Downarrow n}{(x := e, \sigma) \Downarrow \sigma\{x \rightsquigarrow n\}}$$

$$\frac{}{(\mathbf{skip}, \sigma) \Downarrow \sigma}$$

$$\frac{(c_0, \sigma) \Downarrow \sigma' \quad (c_1, \sigma') \Downarrow \sigma''}{(c_0 ; c_1, \sigma) \Downarrow \sigma''}$$

Big-step semantics for statements

$$\frac{(e, \sigma) \Downarrow n}{(x := e, \sigma) \Downarrow \sigma\{x \rightsquigarrow n\}}$$

$$\frac{}{(\mathbf{skip}, \sigma) \Downarrow \sigma}$$

$$\frac{(c_0, \sigma) \Downarrow \sigma' \quad (c_1, \sigma') \Downarrow \sigma''}{(c_0 ; c_1, \sigma) \Downarrow \sigma''}$$

$$\frac{(b, \sigma) \Downarrow \mathit{true} \quad (c_0, \sigma) \Downarrow \sigma'}{(\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma) \Downarrow \sigma'}$$

$$\frac{(b, \sigma) \Downarrow \mathit{false} \quad (c_1, \sigma) \Downarrow \sigma'}{(\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma) \Downarrow \sigma'}$$

Big-step semantics for statements

$$\frac{(e, \sigma) \Downarrow n}{(x := e, \sigma) \Downarrow \sigma\{x \rightsquigarrow n\}}$$

$$\frac{}{(\mathbf{skip}, \sigma) \Downarrow \sigma}$$

$$\frac{(c_0, \sigma) \Downarrow \sigma' \quad (c_1, \sigma') \Downarrow \sigma''}{(c_0 ; c_1, \sigma) \Downarrow \sigma''}$$

$$\frac{(b, \sigma) \Downarrow \mathit{true} \quad (c_0, \sigma) \Downarrow \sigma'}{(\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma) \Downarrow \sigma'}$$

$$\frac{(b, \sigma) \Downarrow \mathit{false} \quad (c_1, \sigma) \Downarrow \sigma'}{(\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma) \Downarrow \sigma'}$$

$$\frac{(b, \sigma) \Downarrow \mathit{false}}{(\mathbf{while } b \mathbf{ do } c, \sigma) \Downarrow \sigma}$$

$$\frac{(b, \sigma) \Downarrow \mathit{true} \quad (c, \sigma) \Downarrow \sigma' \quad (\mathbf{while } b \mathbf{ do } c, \sigma') \Downarrow \sigma''}{(\mathbf{while } b \mathbf{ do } c, \sigma) \Downarrow \sigma''}$$

Example

$(x := 5 ; \text{if } x > 3 \text{ then } y := 1 \text{ else } y := 2, \{x \rightsquigarrow 0, y \rightsquigarrow 0\})$
 $\Downarrow \{x \rightsquigarrow 5, y \rightsquigarrow 1\}$

Divergence (non-termination)

If (c, σ) does not terminate, then there does not exist σ' such that $(c, \sigma) \Downarrow \sigma'$.

Can we apply the inductive rule to **(while true do skip, σ)**?

$$\frac{(b, \sigma) \Downarrow \text{true} \quad (c, \sigma) \Downarrow \sigma' \quad (\mathbf{while\ } b \ \mathbf{do\ } c, \sigma') \Downarrow \sigma''}{(\mathbf{while\ } b \ \mathbf{do\ } c, \sigma) \Downarrow \sigma''}$$

Big-Step Semantics

$$\frac{(e, \sigma) \Downarrow n \quad (c, \sigma\{x \rightsquigarrow n\}) \Downarrow \sigma'}{(\mathbf{newvar} \ x := e \ \mathbf{in} \ c, \sigma) \Downarrow \sigma'\{x \rightsquigarrow \sigma x\}}$$

Compared to the small-step semantics:

$$\frac{n = \llbracket e \rrbracket_{intexp} \sigma \quad (c, \sigma\{x \rightsquigarrow n\}) \longrightarrow (c', \sigma') \quad \sigma' x = \lfloor n' \rfloor}{(\mathbf{newvar} \ x := e \ \mathbf{in} \ c, \sigma) \longrightarrow (\mathbf{newvar} \ x := n' \ \mathbf{in} \ c', \sigma'\{x \rightsquigarrow \sigma x\})}$$

$$\frac{}{(\mathbf{newvar} \ x := e \ \mathbf{in} \ \mathbf{skip}, \sigma) \longrightarrow (\mathbf{skip}, \sigma)}$$

Big-Step Semantics

Also, we could add rules to handle the **abort** case. For instance,

$$\frac{(e, \sigma) \Downarrow \mathbf{abort}}{(x := e, \sigma) \Downarrow \mathbf{abort}}$$

$$\frac{(c_0, \sigma) \Downarrow \mathbf{abort}}{(c_0 ; c_1, \sigma) \Downarrow \mathbf{abort}}$$

Equivalence between big-step and small-step semantics

For all c and σ ,

- ▶ $(c, \sigma) \Downarrow \mathbf{abort}$ iff $(c, \sigma) \longrightarrow^* \mathbf{abort}$
- ▶ $(c, \sigma) \Downarrow \sigma'$ iff $(c, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$

Small-step vs. big-step

- ▶ Small-step can clearly model more complex features, like concurrency, divergence, and runtime errors.
- ▶ Although one-step-at-a-time evaluation is useful for proving certain properties, in some cases it is unnecessary work to talk about each small step.
- ▶ Big-step semantics more closely models a recursive interpreter.
- ▶ Big-steps may make it quicker to prove things, because there are fewer rules. The “boring” rules of the small-step semantics that specify order of evaluation are folded in big-step rules.
- ▶ Big-step: all programs without final configurations (infinite loops, getting stuck) look the same. So you sometimes can't prove things related to these kinds of configurations.

Summary of operational semantics

- ▶ Precise specification of dynamic semantics
- ▶ Simple and abstract (compared to implementations)
 - ▶ No low-level details such as memory management, data layout, etc
- ▶ Often not compositional (e.g. **while**)
- ▶ Basis for some proofs about languages
- ▶ Basis for some reasoning about particular programs
- ▶ Point of reference for other semantics

Recall lambda calculus

Syntax

(Term) $M, N ::= x \mid \lambda x. M \mid MN$

Small-step SOS (reduction rules):

$$\frac{}{(\lambda x. M) N \longrightarrow M[N/x]} \qquad \frac{M \longrightarrow M'}{\lambda x. M \longrightarrow \lambda x. M'}$$
$$\frac{M \longrightarrow M'}{MN \longrightarrow M' N} \qquad \frac{N \longrightarrow N'}{MN \longrightarrow M N'}$$

This semantics is non-deterministic.

Can we have contextual semantics and big-step semantics?

More on lambda calculus

Syntax

$$\text{(Term)} \quad M, N ::= x \mid \lambda x. M \mid MN$$

Contextual semantics (still non-deterministic):

$$\text{(Redex)} \quad r ::= (\lambda x. M) N$$

$$\text{(Context)} \quad \mathcal{E} ::= [] \mid \lambda x. \mathcal{E} \mid \mathcal{E} N \mid M \mathcal{E}$$

Local reduction rule:

$$\overline{(\lambda x. M) N \longrightarrow M[N/x]}$$

Global reduction rule:

$$\frac{r \longrightarrow M}{\mathcal{E}[r] \longrightarrow \mathcal{E}[M]}$$

More on lambda calculus

Syntax

(Term) $M, N ::= x \mid \lambda x. M \mid MN$

Big-step semantics:

$$\frac{}{x \Downarrow x} \qquad \frac{M \Downarrow M'}{\lambda x. M \Downarrow \lambda x. M'}$$
$$\frac{M \Downarrow \lambda x. M' \quad N \Downarrow N' \quad M'[N'/x] \Downarrow P}{MN \Downarrow P}$$

Is this equivalent to the small-step semantics?