Programming Languages
Lecture 9 – Operational Semantics

Ulf Norell

February 15, 2006

Syntax versus Semantics

- Syntax
  - Which are the elements of the language?
  - How can they be composed (to form programs or sentences)?
- Semantics
  - What does (a syntactically correct) program *mean*?
- Examples

Syntax	*Which are the keywords of the language?*
Syntax:	*What can you write to the left of an assignment?*
Type system:	*Can you use an integer as the condition in an if-statement?*
Semantics:	*In which order does* (+) *evaluate its arguments?*
Semantics:	*Does program A produce the same result as program B?*

Specifying Semantics

- There are three ways to specify semantics:
  - By giving an implementation
    - Not a good way.
    - Bugs in the compiler become part of the language.
  - Informally
    - Using natural language.
  - Formally
    - Using some mathematical notation.

Informal Semantics

- Example:

  To execute **while** e **do** s*, first evaluate* e*. If the result is* $true$*, execute* s*. Repeat this procedure until* e *no longer evaluates to* $true$*.*

- Advantages
  - Well-known notation.
- Disadvantages
  - Hard to get right.
  - Easy to misunderstand, due to unclear language. Different compilers might make different interpretations.

Formal Semantics

- Exact mathematical description of the language.
- Example:

$$\frac{\langle e, \sigma \rangle \Downarrow \mathit{true} \quad \langle s, \sigma \rangle \Downarrow \sigma' \quad \langle \textbf{while}\ e\ \textbf{do}\ s, \sigma' \rangle \Downarrow \sigma''}{\langle \textbf{while}\ e\ \textbf{do}\ s, \sigma \rangle \Downarrow \sigma''}$$

$$\frac{\langle e, \sigma \rangle \Downarrow \mathit{false}}{\langle \textbf{while}\ e\ \textbf{do}\ s, \sigma \rangle \Downarrow \sigma}$$

- Advantages
  - No ambiguities.
  - Programs can be executed by hand (e.g. to verify the compiler).
  - Can be used to prove properties of programs.
- Disadvantages
  - Yet another notation to learn.
  - A lot of work.
- Most larger languages does not have a formal semantics.
  - ML does. Haskell doesn't.

Different Types of Formal Semantics

- There are three major categories of formal semantics:
  - Operational semantics
    - How is the program executed?
    - What *operations* does it perform?
  - Denotational semantics
    - What does the program mean?
    - What mathematical object does it *denote*?
  - Axiomatic semantics
    - Which logical propositions hold for a program?
    - Example:

$$\{x = 4\}\, x := x + 1\, \{x = 5\}$$

      If $x = 4$ holds before executing $x := x + 1$ then $x = 5$ holds after.

- We will concentrate on operational semantics, and mostly ignore denotational and axiomatic semantics.

Operational Semantics

- Specifies *how* a program is executed.
- Defines an *abstract machine* (or abstract interpreter) that can run programs.
- We can do this in two different ways:
  - Small step semantics
    - Define one step of the abstract machine.
    - Example: $\langle 1 + (2 + 3), \sigma \rangle \rightarrow \langle 1 + 5, \sigma \rangle$
  - Big step semantics
    - Describe how the abstract machine computes the final result.
    - Example: $\langle 1 + (2 + 3), \sigma \rangle \Downarrow 6$.
- In the examples σ is the state of the abstract machine. It can contain, for instance, the values of the variables.

An Example Language

- Remember our small imperative language from lecture 6:

$$
\begin{array}{lcl}
e & ::= & x \mid n \mid b \mid e + e \\
s & ::= & \textbf{while}\ e\ \textbf{do}\ s \mid \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \\
 & \mid & x := e \mid s; s \mid \textbf{skip} \\
v & ::= & n \mid b
\end{array}
$$

- For reasons that will soon be revealed we add a statement **skip** that does nothing.
- We also add a category of *values* that represents expressions that cannot be further evaluated.
- To construct an abstract machine we first need to define its state:
  - The state is a set of variable-value pairs.
  - Looking up the value of a variable: $\sigma(x)$.
  - Updating the state: $\sigma[x \mapsto v]$

Small Step Semantics for Expressions

- For expressions we define a rewrite relation:

$$\langle e, \sigma \rangle \rightarrow \langle e', \sigma \rangle$$

  meaning that in the state σ, e can be rewritten to e' in one step.

- The rules:

$$\frac{}{\langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle} \text{ (VAR)}$$

$$\frac{\langle e_1, \sigma \rangle \rightarrow \left\langle e_1', \sigma \right\rangle}{\langle e_1 + e_2, \sigma \rangle \rightarrow \left\langle e_1' + e_2, \sigma \right\rangle} \text{ (ADD.1)} \qquad \frac{\langle e, \sigma \rangle \rightarrow \langle e', \sigma \rangle}{\langle v + e, \sigma \rangle \rightarrow \langle v + e', \sigma \rangle} \text{ (ADD.2)}$$

$$\frac{v \text{ is the sum of } v_1 \text{ and } v_2}{\langle v_1 + v_2, \sigma \rangle \rightarrow \langle v, \sigma \rangle} \text{ (ADD.3)}$$

- Remarks:
  - Rewriting stops when it reaches a value.
  - It isn't obvious from the form of the rules that expressions don't change the state. We have to check the individual rules to verify this.

Small Step Semantics for Statements

- Statements are allowed to change the state: $\langle s, \sigma \rangle \longrightarrow \langle s', \sigma' \rangle$
- The rules:

$$\frac{\langle e, \sigma \rangle \rightarrow \langle e', \sigma \rangle}{\langle x := e, \sigma \rangle \rightarrow \langle x := e', \sigma \rangle}\ (\textsc{Assign.1}) \qquad \frac{}{\langle x := v, \sigma \rangle \rightarrow \langle \mathbf{skip}, \sigma[x \mapsto v] \rangle}\ (\textsc{Assign.2})$$

$$\frac{\langle e, \sigma \rangle \rightarrow \langle e', \sigma \rangle}{\langle \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \sigma \rangle \rightarrow \langle \mathbf{if}\ e'\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \sigma \rangle}\ (\textsc{If})$$

$$\frac{}{\langle \mathbf{if}\ \mathit{true}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \sigma \rangle \rightarrow \langle s_1, \sigma \rangle}\ (\textsc{IfTrue})$$

$$\frac{}{\langle \mathbf{if}\ \mathit{false}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \sigma \rangle \rightarrow \langle s_2, \sigma \rangle}\ (\textsc{IfFalse})$$

$$\frac{\langle s_1, \sigma \rangle \rightarrow \left\langle s_1', \sigma' \right\rangle}{\langle s_1; s_2, \sigma \rangle \rightarrow \left\langle s_1'; s_2, \sigma' \right\rangle}\ (\textsc{Seq}) \qquad \frac{}{\langle \mathbf{skip}; s, \sigma \rangle \rightarrow \langle s, \sigma \rangle}\ (\textsc{SeqSkip})$$

$$\frac{}{\langle \mathbf{while}\ e\ \mathbf{do}\ s, \sigma \rangle \rightarrow \langle \mathbf{if}\ e\ \mathbf{then}\ s; \mathbf{while}\ e\ \mathbf{do}\ s\ \mathbf{else}\ \mathbf{skip}, \sigma \rangle}\ (\textsc{While})$$

- A program terminates when it reaches $\langle \mathbf{skip}, \sigma \rangle$.

Evaluation Contexts

- We had lots of rules of the form

$$\frac{\langle t, \sigma \rangle \to \langle t', \sigma' \rangle}{\langle \ldots t \ldots, \sigma \rangle \to \langle \ldots t' \ldots, \sigma' \rangle}$$

- These are called context rules, and control the order of evaluation.
- A more compact way of writing these rules is by defining valid evaluation contexts:
  - An evaluation context is a term with a hole ($\bullet$) in it.
  - Example: $\textbf{if} \ \bullet \ \textbf{then} \ x := 0 \ \textbf{else} \ x := 1$
  - The hole tells you where it is allowed to do rewriting.
- Evaluation contexts for our language:

$$\begin{array}{lcl} E & ::= & \bullet \ | \ E + e \ | \ v + E \\ S & ::= & \bullet \ | \ \textbf{if} \ E \ \textbf{then} \ s \ \textbf{else} \ s \ | \ x := E \ | \ S; s \end{array}$$

Complete Small Step Semantics

- Evaluation contexts:

$$
\begin{array}{lcl}
E & ::= & \bullet \mid E + e \mid v + E \\
S & ::= & \bullet \mid \textbf{if } E \textbf{ then } s \textbf{ else } s \mid x := E \mid S; s
\end{array}
$$

- Rules:

$$\frac{}{\langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle}\ (\textsc{Var}) \qquad \frac{v \text{ is the sum of } v_1 \text{ and } v_2}{\langle v_1 + v_2, \sigma \rangle \rightarrow \langle v, \sigma \rangle}\ (\textsc{Add})$$

$$\frac{}{\langle x := v, \sigma \rangle \rightarrow \langle \textbf{skip}, \sigma[x \mapsto v] \rangle}\ (\textsc{Assign}) \qquad \frac{}{\langle \textbf{skip}; s, \sigma \rangle \rightarrow \langle s, \sigma \rangle}\ (\textsc{Seq})$$

$$\frac{}{\langle \textbf{if } \mathit{true} \textbf{ then } s_1 \textbf{ else } s_2, \sigma \rangle \rightarrow \langle s_1, \sigma \rangle}\ (\textsc{IfTrue})$$

$$\frac{}{\langle \textbf{if } \mathit{false} \textbf{ then } s_1 \textbf{ else } s_2, \sigma \rangle \rightarrow \langle s_2, \sigma \rangle}\ (\textsc{IfFalse})$$

$$\frac{}{\langle \textbf{while } e \textbf{ do } s, \sigma \rangle \rightarrow \langle \textbf{if } e \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip}, \sigma \rangle}\ (\textsc{While})$$

Small Step Example

$$
\begin{array}{lll}
 & & \left\langle \begin{array}{l} \textbf{while}\ x\ \textbf{do}\ x := \mathit{false}; \\ y := 0 \end{array}, \ \{x \mapsto \mathit{true}\} \right\rangle \\
(\textsc{While}) & \to & \left\langle \begin{array}{l} \textbf{if}\ x \quad \textbf{then}\ \ x := \mathit{false};\ \textbf{while}\ x\ \textbf{do}\ x := \mathit{false} \\ \qquad\quad\ \ \textbf{else}\ \ \ \textbf{skip}; \\ y := 0 \end{array}, \ \{x \mapsto \mathit{true}\} \right\rangle \\
(\textsc{Var}) & \to & \left\langle \begin{array}{l} \textbf{if}\ \mathit{true} \quad \textbf{then}\ \ x := \mathit{false};\ \textbf{while}\ x\ \textbf{do}\ x := \mathit{false} \\ \qquad\qquad\ \ \textbf{else}\ \ \ \textbf{skip}; \\ y := 0 \end{array}, \ \{x \mapsto \mathit{true}\} \right\rangle \\
(\textsc{IfTrue}) & \to & \left\langle \begin{array}{l} x := \mathit{false}; \\ \textbf{while}\ x\ \textbf{do}\ x := \mathit{false}; \\ y := 0 \end{array}, \ \{x \mapsto \mathit{true}\} \right\rangle \\
(\textsc{Assign}), (\textsc{Seq}) & \to^* & \left\langle \begin{array}{l} \textbf{while}\ x\ \textbf{do}\ x := \mathit{false}; \\ y := 0 \end{array}, \ \{x \mapsto \mathit{false}\} \right\rangle \\
(\textsc{While}), (\textsc{Var}), (\textsc{IfFalse}) & \to^* & \langle \textbf{skip};\ y := 0, \ \{x \mapsto \mathit{false}\} \rangle \\
(\textsc{Seq}) & \to & \langle y := 0, \ \{x \mapsto \mathit{false}\} \rangle \\
(\textsc{Assign}) & \to & \langle \textbf{skip}, \ \{x \mapsto \mathit{false}, y \mapsto 0\} \rangle
\end{array}
$$

Big Step Semantics

- Focus: the relation between the start state and the final state.
- We write
$$\langle P, \sigma \rangle \Downarrow \sigma'$$
to say that the program P terminates in the final state σ' when run in the state σ.
- Corresponds to $\langle P, \sigma \rangle \rightarrow^* \langle \mathbf{skip}, \sigma' \rangle$ in the small step semantics.
- For expressions we say
$$\langle e, \sigma \rangle \Downarrow v$$
meaning that the expression e evaluates to v in the state σ (corresponding to $\langle e, \sigma \rangle \rightarrow^* \langle v, \sigma \rangle$).
- We skip all the bothersome intermediate steps and jump directly to the conclusion (hence the name big step).

Our Example—Expressions

- The rules for expressions:

$$\frac{}{\langle n,\sigma\rangle \Downarrow n}\ (\textsc{Int}) \qquad \frac{}{\langle b,\sigma\rangle \Downarrow b}\ (\textsc{Bool}) \qquad \frac{}{\langle x,\sigma\rangle \Downarrow \sigma(x)}\ (\textsc{Var})$$

$$\frac{\langle e_1,\sigma\rangle \Downarrow v_1 \quad \langle e_2,\sigma\rangle \Downarrow v_2 \quad v \text{ is the sum of } v_1 \text{ and } v_2}{\langle e_1 + e_2,\sigma\rangle \Downarrow v}\ (\textsc{Plus})$$

- We need rules for integers and booleans, saying that they evaluate to themselves.
- It's not clear in which order the arguments to (+) are evaluated (in truth it doesn't matter, since expressions don't have side effects).

Our Example—Statements

- The rules for statements:

$$\frac{\langle e, \sigma \rangle \Downarrow v}{\langle x := e, \sigma \rangle \Downarrow \sigma[x \mapsto v]} \text{ (ASSIGN)} \qquad \frac{\langle s_1, \sigma \rangle \Downarrow \sigma' \quad \langle s_2, \sigma' \rangle \Downarrow \sigma''}{\langle s_1; s_2, \sigma \rangle \Downarrow \sigma''} \text{ (SEQ)}$$

$$\frac{\langle e, \sigma \rangle \Downarrow \mathit{true} \quad \langle s_1, \sigma \rangle \Downarrow \sigma'}{\langle \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2, \sigma \rangle \Downarrow \sigma'} \text{ (IFTRUE)} \qquad \frac{\langle e, \sigma \rangle \Downarrow \mathit{false} \quad \langle s_2, \sigma \rangle \Downarrow \sigma'}{\langle \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2, \sigma \rangle \Downarrow \sigma'} \text{ (IFFALSE)}$$

$$\frac{\langle e, \sigma \rangle \Downarrow \mathit{true} \quad \langle s, \sigma \rangle \Downarrow \sigma' \quad \langle \textbf{while } e \textbf{ do } s, \sigma' \rangle \Downarrow \sigma''}{\langle \textbf{while } e \textbf{ do } s, \sigma \rangle \Downarrow \sigma''} \text{ (WHILETRUE)}$$

$$\frac{\langle e, \sigma \rangle \Downarrow \mathit{false}}{\langle \textbf{while } e \textbf{ do } s, \sigma \rangle \Downarrow \sigma} \text{ (WHILEFALSE)} \qquad \frac{}{\langle \textbf{skip}, \sigma \rangle \Downarrow \sigma} \text{ (SKIP)}$$

- Note that the (WHILETRUE)-rule is recursive (not surprisingly).

Example Derivation

- Let's revisit the example from the small step semantics

$$
(1)\quad \dfrac{\dfrac{\overline{\langle 0, \{x \mapsto \mathit{false}\}\rangle \Downarrow 0}}{\langle y := 0, \{x \mapsto \mathit{false}\}\rangle \Downarrow \{x \mapsto \mathit{false}, y \mapsto 0\}}}{\left\langle \begin{array}{l}\textbf{while } x \textbf{ do } x := \mathit{false};\\ y := 0\end{array}, \{x \mapsto \mathit{true}\}\right\rangle \Downarrow \{x \mapsto \mathit{false}, y \mapsto 0\}}
$$

$$
(1)\quad \dfrac{\overline{\langle x, \{x \mapsto \mathit{true}\}\rangle \Downarrow \mathit{true}} \quad \dfrac{\overline{\langle \mathit{false}, \{x \mapsto \mathit{true}\}\rangle \Downarrow \mathit{false}}}{\langle x := \mathit{false}, \{x \mapsto \mathit{true}\}\rangle \Downarrow \{x \mapsto \mathit{false}\}} \quad (2)}{\left\langle \begin{array}{l}\textbf{while } x \textbf{ do}\\ \quad x := \mathit{false}\end{array}, \{x \mapsto \mathit{true}\}\right\rangle \Downarrow \{x \mapsto \mathit{false}\}}
$$

$$
(2)\quad \dfrac{\overline{\langle x, \{x \mapsto \mathit{false}\}\rangle \Downarrow \mathit{false}}}{\left\langle \begin{array}{l}\textbf{while } x \textbf{ do}\\ \quad x := \mathit{false}\end{array}, \{x \mapsto \mathit{false}\}\right\rangle \Downarrow \{x \mapsto \mathit{false}\}}
$$

- This is less readable (and harder to write down) than the small step derivation.

Abstract Interpretations

- Let's change notation slightly:

$$\sigma \vdash e \Downarrow v \;\; \text{instead of} \;\; \langle e, \sigma \rangle \Downarrow v$$

- Looks familiar?
- Compare the typing rule for addition with the evaluation rule:

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \qquad \frac{\sigma \vdash e_1 \Downarrow v_1 \quad \sigma \vdash e_2 \Downarrow v_2}{\sigma \vdash e_1 + e_2 \Downarrow v_1 + v_2}$$

- The only difference is how precise the rule is (remember when I said that the type system could be seen as an approximation of what would happen when a program was executed?).
- These kinds of rules are called *abstract interpretations*.

Interpreters

- The judgement
$$\langle P, \sigma \rangle \Downarrow \sigma'$$
means that running the program P in state σ terminates in the state σ'.
- We can see the $\Downarrow$ relation as a function
$$\Downarrow \in \mathit{Program} \times \mathit{State} \rightarrow \mathit{State}$$
- If we implement this function we get an interpreter for our language!
- This is possible because the semantics is deterministic.
  - There is at most one rule applicable at any given time.
  - Moreover, we can tell which one by looking at the structure of the program.
  - This was not the case for some of our advanced type systems.

Implementing an Interpreter

- First we have to define the abstract syntax:

```
type Var   = String
data Expr  = Var Var | Int Int | Bool Bool | Plus Expr Expr
data Stmt  = Assign Var Expr
           | If Expr Stmt Stmt
           | While Expr Stmt
           | Seq Stmt Stmt
           | Skip
data Value = VInt Int | VBool Bool
```

- Next we define the state:

```
type State = [(Var, Value)]
```

$lookupVar$ :: State → Var → Value — $lookupVar\ \sigma\ x\ =\ \sigma(x)$

$updateState$ :: State → Var → Value → State — $updateState\ \sigma\ x\ v\ =\ \sigma[x \mapsto v]$

- We need two interpretation functions:

$eval$ :: Expr → State → Value — $eval\ e\ \sigma\ =\ v$ if $\langle e, \sigma \rangle \Downarrow v$

$exec$ :: Stmt → State → State — $exec\ s\ \sigma\ =\ \sigma'$ if $\langle s, \sigma \rangle \Downarrow \sigma'$

Implementing *eval*

- Evaluating expressions

$$eval\ (\texttt{Var}\ x)\ \sigma = lookupVar\ \sigma\ x \qquad \frac{}{\langle x, \sigma\rangle \Downarrow \sigma(x)}$$

$$eval\ (\texttt{Int}\ n)\ \sigma = \texttt{VInt}\ n \qquad \frac{}{\langle n, \sigma\rangle \Downarrow n}$$

$$eval\ (\texttt{Bool}\ b)\ \sigma = \texttt{VBool}\ b \qquad \frac{}{\langle b, \sigma\rangle \Downarrow b}$$

$$\begin{array}{l} eval\ (\texttt{Plus}\ e_1\ e_2)\ \sigma = plus\ (eval\ e_1\ \sigma)\ (eval\ e_2\ \sigma) \\ \quad \textbf{where}\ plus\ (\texttt{VInt}\ n_1)\ (\texttt{VInt}\ n_2) = \texttt{VInt}\ (n_1 + n_2) \end{array} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow v_1 \quad \langle e_2, \sigma\rangle \Downarrow v_2}{\langle e_1 + e_2, \sigma\rangle \Downarrow v_1 + v_2}$$

- Note the differences between the syntactic and semantic plusses (`Plus` vs. *plus*).

Implementing *exec*

$$exec\ (\texttt{Assign}\ x\ e)\ \sigma = updateState\ \sigma\ x\ (eval\ e\ \sigma)$$

$$\frac{\langle e, \sigma\rangle \Downarrow v}{\langle x := e, \sigma\rangle \Downarrow \sigma[x \mapsto v]}$$

$$\begin{array}{l} exec\ (\texttt{If}\ e\ s_1\ s_2)\ \sigma = \\ \quad \textbf{case}\ eval\ e\ \sigma\ \textbf{of} \\ \quad\quad \texttt{VBool True} \rightarrow exec\ s_1\ \sigma \\ \quad\quad \texttt{VBool False} \rightarrow exec\ s_2\ \sigma \end{array}$$

$$\frac{\langle e, \sigma\rangle \Downarrow true \quad \langle s_1, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle e, \sigma\rangle \Downarrow false \quad \langle s_2, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \sigma\rangle \Downarrow \sigma'}$$

$$\begin{array}{l} exec\ (\texttt{While}\ e\ s)\ \sigma = \\ \quad \textbf{case}\ eval\ e\ \sigma\ \textbf{of} \\ \quad\quad \texttt{VBool True} \rightarrow exec\ (\texttt{While}\ e\ s) \\ \quad\quad\quad\quad\quad\quad\quad\quad (exec\ s\ \sigma) \\ \quad\quad \texttt{VBool False} \rightarrow \sigma \end{array}$$

$$\frac{\langle e, \sigma\rangle \Downarrow true \quad \langle s, \sigma\rangle \Downarrow \sigma' \quad \langle \textbf{while}\ e\ \textbf{do}\ s, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while}\ e\ \textbf{do}\ s, \sigma\rangle \Downarrow \sigma''}$$

$$\frac{\langle e, \sigma\rangle \Downarrow false}{\langle \textbf{while}\ e\ \textbf{do}\ s, \sigma\rangle \Downarrow \sigma}$$

$$exec\ (\texttt{Seq}\ s_1\ s_2)\ \sigma = exec\ s_2\ (exec\ s_1\ \sigma)$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \sigma' \quad \langle s_2, \sigma'\rangle \Downarrow \sigma''}{\langle s_1 ; s_2, \sigma\rangle \Downarrow \sigma''}$$

$$exec\ \texttt{Skip}\ \sigma = \sigma$$

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \Downarrow \sigma}$$

Input/Output

- There are basically two ways of handling I/O in the semantics.
- Model it in the state
  - $\mathit{State} = \mathit{Input} \times \mathit{Output} \times \mathcal{P}(\mathit{Var} \times \mathit{Value})$
  - Example:

$$\frac{\langle e, \langle \iota, o, \sigma \rangle \rangle \Downarrow v}{\langle \mathbf{print}\ e, \langle \iota, o, \sigma \rangle \rangle \Downarrow \langle \iota, o \cdot v, \sigma \rangle}\ (\textsc{Print}) \qquad \frac{}{\langle \mathbf{read}\ x, \langle v \cdot \iota, o, \sigma \rangle \rangle \Downarrow \langle \iota, o, \sigma[x \mapsto v] \rangle}\ (\textsc{Read})$$

- Labelled transition systems
  - Most often used with small step semantics (works for big step as well).
  - Each rewrite step can be *labelled* with an action.
  - Example:

$$\frac{}{\langle \mathbf{print}\ v, \sigma \rangle \xrightarrow{\mathbf{out}(v)} \langle \mathbf{skip}, \sigma \rangle}\ (\textsc{Print}) \qquad \frac{}{\langle \mathbf{read}\ x, \sigma \rangle \xrightarrow{\mathbf{in}(v)} \langle \mathbf{skip}, \sigma[x \mapsto v] \rangle}\ (\textsc{Read})$$

Summary

- Three kinds of formal semantics
  - Operational semantics
    - How to execute a program.
  - Denotational semantics
    - What a program means.
  - Axiomatic semantics
    - What properties a program satisfies.
- Operational semantics
  - Small step
    - Step-by-step rewriting rules.
  - Big step
    - Corresponds to an interpreter.