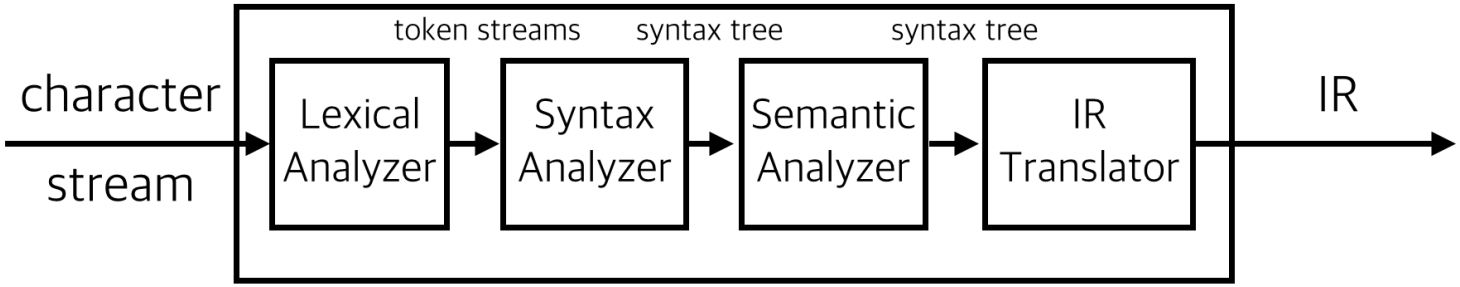
COSE312: Compilers

Lecture 8 — Operational Semantics

Hakjoo Oh
2023 Spring

Beyond Syntax Analysis




```
i = 0
sum = 0
while i < 10:
  sum = sum + i
  i += 1
print(sum)
```

```
 3 : .t1 = 0
 4 : i = .t1
 5 : .t2 = 0
 6 : sum = .t2
 7 : SKIP
 9 : .t4 = i
10 : .t5 = 10
11 : .t3 = .t4 < .t5
21 : iffalse .t3 goto 8
12 : .t7 = sum
13 : .t8 = i
14 : .t6 = .t7 + .t8
15 : sum = .t6
16 : .t10 = i
17 : .t11 = 1
18 : .t9 = .t10 + .t11
19 : i = .t9
```

Syntax vs. Semantics

A programming language is defined with syntax and semantics.

- The syntax is concerned with the grammatical structure of programs.
  - Context-free grammar
- The semantics is concerned with the meaning of grammatically correct programs.
  - Operational semantics: The meaning is specified by the computation steps executed on a machine. It is of intrest how it is obtained.
  - Denotational semantics: The meaning is modelled by mathematical objects that represent the effect of executing the program. It is of interest the effect, not how it is obtained.

The **While** Language: Abstract Syntax

n will range over numerals, **Num**
x will range over variables, **Var**
a will range over arithmetic expressions, **Aexp**
b will range over boolean expressions, **Bexp**
c, S will range over statements, **Stm**

$$
\begin{array}{rcl}
a & \rightarrow & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & \rightarrow & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c & \rightarrow & x := a \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if } b\ c_1\ c_2 \mid \texttt{while } b\ c
\end{array}
$$

Example

The factorial program:

```
y:=1; while ¬(x=1) do (y:=y⋆x; x:=x-1)
```

The abstract syntax tree:

Semantics of Arithmetic Expressions

- The meaning of an expression depends on the values bound to the variables that occur in the expression, e.g., $x + 3$.
- A state is a function from variables to values:

$$\mathbf{State} = \mathbf{Var} \to \mathbb{Z}$$

- The meaning of arithmetic expressions is a function:

$$\mathcal{A} : \mathbf{Aexp} \to \mathbf{State} \to \mathbb{Z}$$

$$\begin{aligned}
\mathcal{A}[\![\, a \,]\!] &: \mathbf{State} \to \mathbb{Z} \\
\mathcal{A}[\![\, n \,]\!](s) &= n \\
\mathcal{A}[\![\, x \,]\!](s) &= s(x) \\
\mathcal{A}[\![\, a_1 + a_2 \,]\!](s) &= \mathcal{A}[\![\, a_1 \,]\!](s) + \mathcal{A}[\![\, a_2 \,]\!](s) \\
\mathcal{A}[\![\, a_1 \star a_2 \,]\!](s) &= \mathcal{A}[\![\, a_1 \,]\!](s) \times \mathcal{A}[\![\, a_2 \,]\!](s) \\
\mathcal{A}[\![\, a_1 - a_2 \,]\!](s) &= \mathcal{A}[\![\, a_1 \,]\!](s) - \mathcal{A}[\![\, a_2 \,]\!](s)
\end{aligned}$$

Semantics of Boolean Expressions

- The meaning of boolean expressions is a function:

$$\mathcal{B} : \mathbf{Bexp} \to \mathbf{State} \to \mathbf{T}$$

where $\mathbf{T} = \{true, false\}$.

$$\begin{aligned}
\mathcal{B}[\![\ b\ ]\!] &: \mathbf{State} \to \mathbf{T} \\
\mathcal{B}[\![\ \texttt{true}\ ]\!](s) &= true \\
\mathcal{B}[\![\ \texttt{false}\ ]\!](s) &= false \\
\mathcal{B}[\![\ a_1 = a_2\ ]\!](s) &= \mathcal{A}[\![\ a_1\ ]\!](s) = \mathcal{A}[\![\ a_2\ ]\!](s) \\
\mathcal{B}[\![\ a_1 \leq a_2\ ]\!](s) &= \mathcal{A}[\![\ a_1\ ]\!](s) \leq \mathcal{A}[\![\ a_2\ ]\!](s) \\
\mathcal{B}[\![\ \neg b\ ]\!](s) &= \mathcal{B}[\![\ b\ ]\!](s) = false \\
\mathcal{B}[\![\ b_1 \wedge b_2\ ]\!](s) &= \mathcal{B}[\![\ b_1\ ]\!](s) \wedge \mathcal{B}[\![\ b_2\ ]\!](s)
\end{aligned}$$

Free Variables

The free variables of an arithmetic expression a are defined to be the set of variables occurring in it:

$$\begin{aligned} FV(n) &= \emptyset \\ FV(x) &= \{x\} \\ FV(a_1 + a_2) &= FV(a_1) \cup FV(a_2) \\ FV(a_1 \star a_2) &= FV(a_1) \cup FV(a_2) \\ FV(a_1 - a_2) &= FV(a_1) \cup FV(a_2) \end{aligned}$$

Exercise) Define free variables of boolean expressions.

Substitution

- $a[y \mapsto a_0]$: the arithmetic expression that is obtained by replacing each occurrence of y in a by a_0.

$$
\begin{array}{rcl}
n[y \mapsto a_0] & = & n \\
x[y \mapsto a_0] & = & \begin{cases} a_0 & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\
(a_1 + a_2)[y \mapsto a_0] & = & (a_1[y \mapsto a_0]) + (a_2[y \mapsto a_0]) \\
(a_1 \star a_2)[y \mapsto a_0] & = & (a_1[y \mapsto a_0]) \star (a_2[y \mapsto a_0]) \\
(a_1 - a_2)[y \mapsto a_0] & = & (a_1[y \mapsto a_0]) - (a_2[y \mapsto a_0])
\end{array}
$$

- $s[y \mapsto v]$: the state s except that the value bound to y is v.

$$
(s[y \mapsto v])(x) = \begin{cases} v & \text{if } x = y \\ s(x) & \text{if } x \neq y \end{cases}
$$

Operational Semantics

Operational semantics is concerned about how to execute programs and not merely what the execution results are.

- *Big-step operational semantics* describes how the overall results of executions are obtained.
- *Small-step operational semantics* describes how the individual steps of the computations take place.

In both kinds, the semantics is specified by a transition system $(\mathbb{S}, \rightarrow)$ where $\mathbb{S}$ is the set of states (configurations) with two types:

- $\langle \boldsymbol{S}, \boldsymbol{s} \rangle$: a nonterminal state (i.e. the statement $\boldsymbol{S}$ is to be executed from the state $\boldsymbol{s}$)
- $\boldsymbol{s}$: a terminal state

The transition relation $(\rightarrow) \subseteq \mathbb{S} \times \mathbb{S}$ describes how the execution takes place. The difference between the two approaches are in the definitions of transition relation.

Big-Step Operational Semantics

The transition relation specifies the relationship between the initial state and the final state:

$$\langle S, s \rangle \rightarrow s'$$

Transition relation is defined with inference rules of the form: A rule has the general form

$$\frac{\langle S_1, s_1 \rangle \rightarrow s'_1, \ldots, \langle S_n, s_n \rangle \rightarrow s'_n}{\langle S, s \rangle \rightarrow s'} \text{ if } \ldots$$

- $S_1, \ldots, S_n$ are statements that constitute S.
- A rule has a number of premises and one conclusion.
- A rule may also have a number of conditions that have to be fulfilled whenever the rule is applied.
- Rules without premises are called axioms.

Big-Step Operational Semantics for **While**

$$\frac{}{\langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![\ a\ ]\!](s)]}$$

$$\frac{}{\langle \texttt{skip}, s\rangle \to s}$$

$$\frac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

$$\frac{\langle S_1, s\rangle \to s'}{\langle \texttt{if}\ b\ S_1\ S_2, s\rangle \to s'} \text{ if } \mathcal{B}[\![\ b\ ]\!](s) = \texttt{true}$$

$$\frac{\langle S_2, s\rangle \to s'}{\langle \texttt{if}\ b\ S_1\ S_2, s\rangle \to s'} \text{ if } \mathcal{B}[\![\ b\ ]\!](s) = \texttt{false}$$

$$\frac{\langle S, s\rangle \to s' \qquad \langle \texttt{while}\ b\ S, s'\rangle \to s''}{\langle \texttt{while}\ b\ S, s\rangle \to s''} \text{ if } \mathcal{B}[\![\ b\ ]\!](s) = \texttt{true}$$

$$\frac{}{\langle \texttt{while}\ b\ S, s\rangle \to s} \text{ if } \mathcal{B}[\![\ b\ ]\!](s) = \texttt{false}$$

Example

Consider the statement:

$$\texttt{(z:=x; x:=y); y:=z}$$

Let s_0 be the state that maps all variables except $\texttt{x}$ and $\texttt{y}$ and has $s_0(\texttt{x}) = 5$ and $s_0(\texttt{y}) = 7$.

$$\dfrac{\dfrac{\langle \texttt{z} := \texttt{x}, s_0\rangle \to s_1 \qquad \langle \texttt{x} := \texttt{y}, s_1\rangle \to s_2}{\langle \texttt{z} := \texttt{x}; \texttt{x} := \texttt{y}, s_0\rangle \to s_2} \qquad \langle \texttt{y} := \texttt{z}, s_2\rangle \to s_3}{\langle (\texttt{z} := \texttt{x}; \texttt{x} := \texttt{y}); \texttt{y} := \texttt{z}, s_0\rangle \to s_3}$$

where we have used the abbreviations:

$$\begin{aligned} s_1 &= s_0[z \mapsto 5] \\ s_2 &= s_1[x \mapsto 7] \\ s_3 &= s_2[y \mapsto 5] \end{aligned}$$

Exercise

Let s be a state with $s(x) = 3$. Find s' such that

$$(\texttt{y:=1; while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s) \rightarrow s'$$

Execution Types

We say the execution of a statement S on a state s

- *terminates* if and only if there is a state s' such that $\langle S, s\rangle \rightarrow s'$ and
- *loops* if and only if there is no state s' such that $\langle S, s\rangle \rightarrow s'$.

We say a statement S always terminates if its execution on a state s terminates for all states s, and always loops if its execution on a state s loops for all states s.

Examples:

- `while true do skip`
- `while ¬(x=1) do (y:=y⋆x; x:=x-1)`

Semantic Equivalence

We say S_1 and S_2 are semantically equivalent, denoted $S_1 \equiv S_2$, if the following is true for all states s and s':

$$\langle S_1, s \rangle \rightarrow s' \quad \text{if and only if} \quad \langle S_2, s \rangle \rightarrow s'$$

Example:

$$\texttt{while } b \texttt{ do } S \equiv \texttt{if } b \texttt{ then } (S\texttt{; while } b \texttt{ do } S\texttt{) else skip}$$

Semantic Function for Statements

The semantic function for statements is the partial function:

$$\mathcal{S}_b : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

$$\mathcal{S}_b[\![\ S\ ]\!](s) = \begin{cases} s' & \text{if } \langle S, s\rangle \to s' \\ \mathbf{undef} & \text{otherwise} \end{cases}$$

Examples:

- $\mathcal{S}_b[\![$ `y:=1; while ¬(x=1) do (y:=y⋆x; x:=x-1)` $]\!](s[x \mapsto 3])$
- $\mathcal{S}_b[\![$ `while true do skip` $]\!]$(s)

Summary

The syntax is defined by the grammar:

$$
\begin{array}{rcl}
a & \rightarrow & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & \rightarrow & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c & \rightarrow & x := a \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if } b\ c_1\ c_2 \mid \texttt{while } b\ c
\end{array}
$$

The semantics is defined by the functions:

$$
\begin{array}{rcl}
\mathcal{A}[\![\ a\ ]\!] & : & \mathbf{State} \rightarrow \mathbb{Z} \\
\mathcal{B}[\![\ b\ ]\!] & : & \mathbf{State} \rightarrow \mathbf{T} \\
\mathcal{S}_b[\![\ c\ ]\!] & : & \mathbf{State} \hookrightarrow \mathbf{State}
\end{array}
$$

Implementing Big-Step Interpreter in OCaml

```
type var = string

type aexp =
  | Int of int
  | Var of var
  | Plus of aexp * aexp
  | Mult of aexp * aexp
  | Minus of aexp * aexp

type bexp =
  | True
  | False
  | Eq of aexp * aexp
  | Le of aexp * aexp
  | Neg of bexp
  | Conj of bexp * bexp

type cmd =
  | Assign of var * aexp
  | Skip
  | Seq of cmd * cmd
  | If of bexp * cmd * cmd
  | While of bexp * cmd
```

Implementing Big-Step Interpreter

```
let fact =
  Seq (Assign ("y", Int 1),
    While (Neg (Eq (Var "x", Int 1)),
      Seq (Assign("y", Mult(Var "y", Var "x")),
           Assign("x", Minus(Var "x", Int 1)))
     )
  )

module State = struct
  type t = (var * int) list
  let empty = []
  let rec lookup s x =
    match s with
    | [] -> raise (Failure (x ^ "is not bound in state"))
    | (y,v)::s' -> if x = y then v else lookup s' x
  let update s x v = (x,v)::s
end

let init_s = update empty "x" 3
```

Implementing Big-Step Interpreter

```
let rec eval_a : aexp -> State.t -> int
=fun a s ->
  match a with
  | Int n -> n
  | Var x -> State.lookup s x
  | Plus (a1, a2) -> (eval_a a1 s) + (eval_a a2 s)
  | Mult (a1, a2) -> (eval_a a1 s) * (eval_a a2 s)
  | Minus (a1, a2) -> (eval_a a1 s) - (eval_a a2 s)

let rec eval_b : bexp -> State.t -> bool
=fun b s ->
  match b with
  | True -> true
  | False -> false
  | Eq (a1, a2) -> (eval_a a1 s) = (eval_a a2 s)
  | Le (a1, a2) -> (eval_a a1 s) <= (eval_a a2 s)
  | Neg b' -> not (eval_b b' s)
  | Conj (b1, b2) -> (eval_b b1 s) && (eval_b b2 s)
```

Implementing Big-Step Interpreter

```
let rec eval_c : cmd -> State.t -> State.t
=fun c s ->
  match c with
  | Assign (x, a) -> State.update s x (eval_a a s)
  | Skip -> s
  | Seq (c1, c2) -> eval_c c2 (eval_c c1 s)
  | If (b, c1, c2) -> eval_c (if eval_b b s then c1 else c2) s
  | While (b, c) ->
    if eval_b b s then eval_c (While (b,c)) (eval_c c s)
    else s

let _ =
    print_int (State.lookup (eval_c fact init_s) "y");
    print_newline ()
```

Small-Step Operational Semantics

The individual computation steps are described by the transition relation of the form:

$$\langle S, s \rangle \Rightarrow \gamma$$

where γ either is non-terminal state $\langle S', s' \rangle$ or terminal state s'. The transition expresses the first step of the execution of S from state s.

- If $\gamma = \langle S', s' \rangle$, then the execution of S from s is not completed and the remaining computation continues with $\langle S', s' \rangle$.
- If $\gamma = s'$, then the execution of S from s has terminated and the final state is s'.

We say $\langle S, s \rangle$ is stuck if there is no γ such that $\langle S, s \rangle \Rightarrow \gamma$ (no stuck state for **While**).

Small-Step Operational Semantics for **While**

$$\frac{}{\langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![\ a\ ]\!](s)]}$$

$$\frac{}{\langle \texttt{skip}, s\rangle \Rightarrow s}$$

$$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

$$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$\frac{}{\langle \texttt{if}\ b\ S_1\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle}\ \text{if}\ \mathcal{B}[\![\ b\ ]\!](s) = \texttt{true}$$

$$\frac{}{\langle \texttt{if}\ b\ S_1\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle}\ \text{if}\ \mathcal{B}[\![\ b\ ]\!](s) = \texttt{false}$$

$$\frac{}{\langle \texttt{while}\ b\ S, s\rangle \Rightarrow \langle \texttt{if}\ \ b\ \ (S;\ \texttt{while}\ \ b\ \ S)\ \ \texttt{skip}, s\rangle}$$

Derivation Sequence

A *derivation sequence* of a statement S starting in state s is either

- A finite sequence

  $$\gamma_0, \gamma_1, \gamma_2, \cdots, \gamma_k$$

  which is sometimes written

  $$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_k$$

  such that

  $$\gamma_0 = \langle S, s \rangle, \quad \gamma_i \Rightarrow \gamma_{i+1} \text{ for } 0 \leq i \leq k$$

  and γ_k is either a terminal configuration or a stuck configuration.
- An infinite sequence

  $$\gamma_0, \gamma_1, \gamma_2, \cdots$$

  which is sometimes written

  $$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots$$

  consisting of configurations satisfying $\gamma_0 = \langle S, s \rangle$ and $\gamma_i \Rightarrow \gamma_{i+1}$ for $0 \leq i$.

Example

Consider the statement:

$$\texttt{(z:=x; x:=y); y:=z}$$

Let s_0 be the state that maps all variables except x and y and has $s_0(\mathrm{x}) = 5$ and $s_0(\mathrm{y}) = 7$. We then have the derivation sequence:

$$\begin{aligned}
&\langle (\mathrm{z} := \mathrm{x}; \mathrm{x} := \mathrm{y}); \mathrm{y} := \mathrm{z}, s_0 \rangle \\
&\Rightarrow \langle \mathrm{x} := \mathrm{y}; \mathrm{y} := \mathrm{z}, s_0[z \mapsto 5] \rangle \\
&\Rightarrow \langle \mathrm{y} := \mathrm{z}, s_0[z \mapsto 5, x \mapsto 7] \rangle \\
&\Rightarrow s_0[z \mapsto 5, x \mapsto 7, y \mapsto 5]
\end{aligned}$$

Each step has a derivation tree explaining why it takes place, e.g.,

$$\dfrac{\dfrac{\langle \mathrm{z} := \mathrm{x}, s_0 \rangle \Rightarrow s_0[z \mapsto 5]}{\langle \mathrm{z} := \mathrm{x}; \mathrm{x} := \mathrm{y}, s_0 \rangle \Rightarrow \langle \mathrm{x} := \mathrm{y}, s_0[z \mapsto 5] \rangle}}{\langle (\mathrm{z} := \mathrm{x}; \mathrm{x} := \mathrm{y}); \mathrm{y} := \mathrm{z}, s_0 \rangle \Rightarrow \langle \mathrm{x} := \mathrm{y}; \mathrm{y} := \mathrm{z}, s_0[z \mapsto 5] \rangle}$$

Example: Factorial

Assume that $s(x) = 3$.

$$
\begin{aligned}
&\langle \texttt{y:=1; while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s\rangle \\
&\Rightarrow \langle \texttt{while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s[y \mapsto 1]\rangle \\
&\Rightarrow \langle \texttt{if } \neg\texttt{(x=1) then ((y:=y}\star\texttt{x; x:=x-1);while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1))} \\
&\quad \texttt{else skip}, s[y \mapsto 1]\rangle \\
&\Rightarrow \langle \texttt{((y:=y}\star\texttt{x; x:=x-1);while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s[y \mapsto 1]\rangle \\
&\Rightarrow \langle \texttt{x:=x-1;while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s[y \mapsto 3]\rangle \\
&\Rightarrow \langle \texttt{while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s[y \mapsto 3][x \mapsto 2]\rangle \\
&\Rightarrow \langle \texttt{if } \neg\texttt{(x=1) then ((y:=y}\star\texttt{x; x:=x-1);while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1))} \\
&\quad \texttt{else skip}, s[y \mapsto 3][x \mapsto 2]\rangle \\
&\Rightarrow \langle \texttt{((y:=y}\star\texttt{x; x:=x-1);while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s[y \mapsto 3][x \mapsto 2]\rangle \\
&\Rightarrow \langle \texttt{x:=x-1;while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s[y \mapsto 6][x \mapsto 2]\rangle \\
&\Rightarrow \langle \texttt{while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s[y \mapsto 6][x \mapsto 1]\rangle \\
&\Rightarrow s[y \mapsto 6][x \mapsto 1]
\end{aligned}
$$

Other Notations

- We write $\gamma_0 \Rightarrow^i \gamma_i$ to indicate that there are i steps in the execution from γ_0 to γ_i.
- We write $\gamma_0 \Rightarrow^* \gamma_i$ to indicate that there are a finite number of steps.
- We say that the execution of a statement S on a state s terminates if and only if there is a finite derivation sequence starting with $\langle S, s \rangle$.
- The execution loops if and only if there is an infinite derivation sequence starting with $\langle S, s \rangle$.

Semantic Function

The semantic function $\mathcal{S}_s$ for small-step semantics:

$$\mathcal{S}_s : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

$$\mathcal{S}_s[\![\ S\ ]\!](s) = \begin{cases} s' & \text{if } \langle S, s\rangle \Rightarrow^* s' \\ \mathbf{undef} & \end{cases}$$

Implementing Small-Step Interpreter

```
type conf =
  | NonTerminated of cmd * State.t
  | Terminated of State.t

let rec next : conf -> conf
=fun conf ->
  match conf with
  | Terminated _ -> raise (Failure "Must not happen")
  | NonTerminated (c, s) ->
    match c with
    | Assign (x, a) -> Terminated (State.update s x (eval_a a s))
    | Skip -> Terminated s
    | Seq (c1, c2) -> (
       match  (next (NonTerminated (c1,s))) with
       | NonTerminated (c', s') -> NonTerminated (Seq (c', c2), s')
       | Terminated s' -> NonTerminated (c2, s')
      )
    | If (b, c1, c2) ->
      if eval_b b s then NonTerminated (c1, s) else NonTerminated (c2, s)
    | While (b, c) -> NonTerminated (If (b, Seq (c, While (b,c)), Skip), s)
```

Implementing Small-Step Interpreter

```
let rec next_trans : conf -> State.t
=fun conf ->
  match conf with
  | Terminated s -> s
  | _ -> next_trans (next conf)

let _ =
  print_int (State.lookup (next_trans (NonTerminated (fact,init_s))) "y");
  print_newline ()
```

Summary

We have defined the operational semantics of **While**.

- *Big-step operational semantics* describes how the overall results of executions are obtained.
- *Small-step operational semantics* describes how the individual steps of the computations take place.

The big-step and small-step operational semantics are equivalent:

Theorem

For every statement S *of* **While**, *we have* $\mathcal{S}_b[\![S]\!] = \mathcal{S}_s[\![S]\!]$.