5 – Semantics of the While language

Mark Armstrong, PhD Candidate, McMaster University

Fall, 2019

1 Preamble

1.1 Notable references

- Programming Languages and Operational Semantics: A Concise Overview, Fernández
  - Chapter 4 – Operational Semantics of Imperative Languages

1.2 Update history

Nov. 12
- A correction to the PDF version: missing prime symbols were added.
- Slight cleanup; added references and table of contents.

Nov. 4
- Original version posted

1.3 Table of contents

2 Introduction to operational semantics

An *operational semantics* describes the meaning of programs in terms of computation steps on some abstract machine.

In particular, our "abstract machine" for the *While* language will be a *relation*.

We consider two different semantics, each consisting of two relations.

- Small-step, or *reduction*, semantics,
  - which defines two *reduction relations*
    - `_⟶_ : Expr × Env × State → Expr × Env × State`
    - `_⟶_ : Stmt × Env × State → Stmt × Env × State`
  - These relations describe individual computation steps.
- Big-step, or *evaluation*, semantics,
  - which defines two *evaluation relations*
    - `_⇓_ : Expr × Env × State → Value`
      - where `Value` is the set of values of the appropriate types
    - `_⇓_ : Stmt × Env × State → Env × State`
  - These relations describe the final results of computation.

2.1 Why two different semantics?

The two semantics we present, big-step and small-step operational semantics, each serve a different purpose.

- The small-step semantics precisely describe the order in which expressions/statements are computed.
  - For instance, in the case of a binary operator, small-step semantics describe which operand is computed first.
- The big-step semantics provide a simpler view of computation, in that there are fewer rules.
  - Sometimes big-step semantics are called "natural" semantics.

Neither approach is "correct" or "incorrect"; they are complementary!

2.2 Syntactic correctness

We assume in these slides that every program we consider is *correct*, meaning it is

- syntactically correct, and in particular that they are
  - well typed,
  - well scoped, and
  - every variable is declared before it is used.

That is, we assume these syntactic/static semantic rules have been enforced

- by a context-free grammar
  - (given in notes 3; we repeat the relevant portions below)
- and an attribute grammars
  - (to be given as part of assignment 2).

2.3 Defining a relation via inference rules

To define our two semantic relations, we use *inference rules*.

An inference rule consists of

- a set of *premises* and
- a *conclusion*;
- we also give each inference rule a *name*.

Inference rules are written

$$\frac{\texttt{premise}_1 \quad \texttt{premise}_2 \quad \dots \quad \texttt{premise}_n}{\texttt{conclusion}}\ \texttt{name}$$

and such an inference rule can be read

- If $\texttt{premise}_1$, $\texttt{premise}_2$, ..., and $\texttt{premise}_n$ are true (satisfied), then `conclusion` is true (satisfied).

2.4 Example inference rules – defining the "less than" relation

You are familiar with the relation $\leq$ `:` `ℕ → ℕ`.

- It can be defined via two inference rules.
  - One "base case" rule with no premises, and
  - one "induction step" with one premise.

$$\frac{}{\texttt{0} \leq \texttt{n}}\ \texttt{Zero is least}$$

$$\frac{\texttt{m} \leq \texttt{n}}{\texttt{suc m} \leq \texttt{suc n}}\ \texttt{Less than successor}$$

which can be read as

- Zero is less than or equal to any natural number `n`

and

- If `m` is less than or equal to `n`, then `suc m` is less than or equal to `suc m`.

3 Semantics of ***Expr*** $_0$

Recall our language of expressions, *Expr*$_0$.

```
⟨expr⟩ ::= ⟨bexpr⟩ | ⟨iexpr⟩
⟨bexpr⟩ ::= true | false
        | ⟨expr⟩ == ⟨expr⟩ | ⟨expr⟩ \= ⟨expr⟩
        | ⟨expr⟩ =< ⟨expr⟩ | ⟨expr⟩ < ⟨expr⟩
        | ⟨expr⟩ >= ⟨expr⟩ | ⟨expr⟩ > ⟨expr⟩
⟨iexpr⟩ ::= number | var
        | ⟨iexpr⟩ + ⟨iexpr⟩ | ⟨iexpr⟩ - ⟨iexpr⟩
        | ⟨iexpr⟩ * ⟨iexpr⟩ | ⟨iexpr⟩ div ⟨iexpr⟩ | ⟨iexpr⟩ mod ⟨iexpr⟩
```

We'll define both the small-step and big-step semantics of expressions before moving on to statements.

3.1 Meta-variables

Throughout this section, the meta-variables

- `E`, $\texttt{E}_1$, $\texttt{E}_2$, `E'`, etc. vary over *expressions*,
- `c`, $\texttt{c}_1$, $\texttt{c}_2$, `c'`, etc. vary over *constant* expressions of either integer or boolean type,
- `x` is any variable,
- Σ is any environment, and
- σ is any state.

3.2 Small-step semantics of ***Expr*** $_0$

Let $\oplus$ be any of the binary, infix operators of the *Expr*$_0$ language.

- That is, $\oplus$ is a meta-operator standing in for `==`, `\=`, `+`, `*`, etc.

We make the (arbitrary) choice to always reduce the left subexpression first.

- An alternate small-step semantics could reduce the right subexpression, or give different orders for different operators.

Then the small-step semantics of *Expr*$_0$ can be given by just four inference rules.

$$\frac{\texttt{n = } \sigma(\Sigma(\texttt{x}))}{(\texttt{x}\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{n}\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{variable}$$

$$\frac{(\texttt{E}_1\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{E}_1\texttt{'}\ ,\ \Sigma\ ,\ \sigma)}{(\texttt{E}_1 \oplus \texttt{E}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{E}_1\texttt{'} \oplus \texttt{E}_2\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{operator-left}$$

$$\frac{(\texttt{E}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{E}_2\texttt{'}\ ,\ \Sigma\ ,\ \sigma)}{(\texttt{c} \oplus \texttt{E}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{c} \oplus \texttt{E}_2\texttt{'}\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{operator-right}$$

$$\frac{\texttt{c = c}_1 \oplus \texttt{c}_2}{(\texttt{c}_1 \oplus \texttt{c}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{c}\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{operator-apply}$$

3.2.1 Explaining the small-step semantic rules of $\boldsymbol{Expr_0}$ – constants

Let us examine each small-step semantic rule.

First, note there are no rules given for the expressions `true`, `false` and `number`.

- That is, there are no rules for constant expressions.
- This is because constants are *irreducible*; they cannot be simplified any further!
  - The presence of such rules would mean we could *infinitely* "reduce" expressions, which is undesirable.

3.2.2 Explaining the small-step semantic rules of $\boldsymbol{Expr_0}$ – variables

The first rule we give

$$\frac{\texttt{n = } \sigma(\Sigma(\texttt{x}))}{(\texttt{x}\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{n}\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{variable}$$

says that a variable reduces to the value stored at the variable in the current environment and state.

- Recall that we only have integer variables in $Expr_0$.

3.2.3 Explaining the small-step semantic rules of $Expr_0$ – operators

The three remaining rules

$$\frac{(\texttt{E}_1\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{E}_1\texttt{'}\ ,\ \Sigma\ ,\ \sigma)}{(\texttt{E}_1 \oplus \texttt{E}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{E}_1\texttt{'} \oplus \texttt{E}_2\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{operator-left}$$

$$\frac{(\texttt{E}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{E}_2\texttt{'}\ ,\ \Sigma\ ,\ \sigma)}{(\texttt{c} \oplus \texttt{E}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{c} \oplus \texttt{E}_2\texttt{'}\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{operator-right}$$

$$\frac{\texttt{c = c}_1 \oplus \texttt{c}_2}{(\texttt{c}_1 \oplus \texttt{c}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\texttt{c}\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{operator-apply}$$

tell us respectively:

- If the left subexpression can be reduced (it is non-constant), then reduce it.
- If the left subexpression is a constant, (expressed by the fact we use the variable `c`) and the right subexpression can be reduced, then reduce the right expression.
- If both subexpressions are constants, then perform the operation.
  - We abuse notation a small amount here; in the premise, $\texttt{c}_1 \oplus \texttt{c}_2$ refers to the application of the operator $\oplus$ to the constants, not an expression.

3.2.4 An example reduction sequence

Consider the expression

```
5 + 3 == 2 * x
```

and suppose in the current environment and state,

- Σ(x) = R and
- σ(R) = 4.

We can test out our reduction rules by trying to reduce this expression to its value, which should be `true`.

```
  5 + 3 == 2 * x
⟶⟨ "operator-left" with "operator-apply" ⟩
  8 == 2 * x
⟶⟨ "operator-right" with "operator-right" with "variable" with fact 4 = σ(Σ(x)) ⟩
  8 == 2 * 4
⟶⟨ "operator-right" with "operator-apply" ⟩
  8 == 8
⟶⟨ "operator-apply" ⟩
  true
```

For each reduction step, we state the inference rule used.

- If the inference rule has a premise involving $\longrightarrow$, we must further state the inference rule which justifies the premise.

3.3 Big-step semantics of *Expr*$_0$

Once again, let $\oplus$ be any of the binary, infix operators of the *Expr*$_0$ language.

The big-step semantics of *Expr*$_0$ can be given by just three inference rules.

$$\frac{}{(\texttt{n}\ ,\ \Sigma\ ,\ \sigma)\Downarrow \texttt{n}}\ \texttt{constant}$$

$$\frac{\texttt{n} = \sigma(\Sigma(\texttt{x}))}{(\texttt{x}\ ,\ \Sigma\ ,\ \sigma)\Downarrow \texttt{n}}\ \texttt{variable}$$

$$\frac{(\texttt{E}_1\ ,\ \Sigma\ ,\ \sigma)\Downarrow \texttt{n}_1 \quad (\texttt{E}_2\ ,\ \Sigma\ ,\ \sigma)\Downarrow \texttt{n}_2 \quad \texttt{n} = \texttt{n}_1 \oplus \texttt{n}_2}{(\texttt{E}_1 \oplus \texttt{E}_2\ ,\ \Sigma\ ,\ \sigma)\Downarrow \texttt{n}}\ \texttt{operator}$$

3.3.1 Notes

- Recall that we did not give a reduction rule for constants; in contrast, we do give an evaluation rule for them.
  - Any expression can be evaluated; not all expressions can be reduced.
- The evaluation relation is between
  - expression, environment, state triples and
  - values;

  we do not "return" the environment and state because they never change.
  - Expressions are evaluated only for their value, not for a side effect.
- These evaluation rules "look similar" to the behaviour of the interpreter provided for assignment 1.
  - It is possible to write an interpreter following the small-step semantics.

3.3.2 An example evaluation

Let us consider once more the expression

```
5 + 3 == 2 * x
```

and an environment and state such that

- `Σ(x) = R` and
- `σ(R) = 4`.

Evaluation semantics do not define steps; to evaluate this expression, we will have to evaluate each subexpression in turn.

```
(1) (5 , Σ , σ) ⇓ 5,                by "constant".
(2) (3 , Σ , σ) ⇓ 3,                by "constant".
(3) (2 , Σ , σ) ⇓ 2,                by "constant".
(4) (x , Σ , σ) ⇓ 4,                by "variable" with fact 4 = σ(Σ(x)).
(5) (5 + 3 , Σ , σ) ⇓ 8,            by "expression" with (1) and (2)
```

```
                                         and fact 8 = 5 + 3.
(6) (2 + x , Σ , σ) ⇓ 8,               by "expression" with (3) and (4)
                                         and fact 8 = 2 * 4.
(7) (5 + 3 == 2 + x , Σ , σ) ⇓ true,   by "expression" with (5) and (6)
                                         and fact 8 == 8.
```

This proof could also be presented as a "derivation tree", but this presentation is unwieldly.

3.3.3 Derivation tree

(Pardon the poor rendering in LaTeX)

For completion's sake, here is the evaluation of expression

```
5 + 3 == 2 * x
```

presented as a derivation tree. We omit the names of the rules used for space.

```
                                                                                    ____________
                                                                                    4 = σ(Σ(x))
_______________   ______________   ____________   _______________   _____________
(5 , Σ , σ) ⇓ 5   (3 , Σ , σ) ⇓ 3   8 == 5 + 3     (2 , Σ , σ) ⇓ 2   (x , Σ , σ)
________________________________________________   ___________________________________
            (5 + 3 , Σ , σ) ⇓ 8                                    (2 * x , Σ , σ) ⇓
______________________________________________________________________________________
                              (5 + 3 == 2 + x , Σ , σ)  ⇓ true
```

4 Semantics of *While*

With the semantics of expressions defined, we are now prepared to give the semantics of *While* statements (programs).

Recall the grammar for *While*.

```
⟨stmt⟩ ::=
  skip
| local var in ⟨stmt⟩
| var := ⟨expr⟩
| ⟨stmt⟩ ⟨stmt⟩
| if ⟨bexpr⟩ then ⟨stmt⟩ else ⟨stmt⟩
| while ⟨bexpr⟩ do ⟨stmt⟩
```

- With expressions, we wanted to reduce/evaluate to a *constant*.
  - For statements, we will want to reduce to the simplest program: `skip`.
  - And we will evaluate statements for their side effects, meaning we return a (potentially updated) *state*.

4.1 Small-step semantics of *While*

4.1.1 Assignment, composition

$$\frac{(\mathtt{E}\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{E'}\ ,\ \Sigma\ ,\ \sigma)}{(\mathtt{x} := \mathtt{E}\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{x} := \mathtt{E'}\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{assign-reduce}$$

$$\frac{}{(\mathtt{x} := \mathtt{n}\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{skip}\ ,\ \Sigma\ ,\ \sigma[\Sigma(\mathtt{x}) := \mathtt{n}])}\ \texttt{assign-number}$$

$$\frac{(\mathtt{S}_1\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{S}_1'\ ,\ \Sigma\ ,\ \sigma')}{(\mathtt{S}_1\ \mathtt{S}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{S}_1'\ \mathtt{S}_2\ ,\ \Sigma\ ,\ \sigma')}\ \texttt{compose-reduce}$$

$$\frac{}{(\mathtt{skip}\ \mathtt{S}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{S}_2\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{compose-skip}$$

4.1.2 Branch, loop

$$\frac{(\mathtt{E}\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{E'}\ ,\ \Sigma\ ,\ \sigma)}{(\mathtt{if}\ \mathtt{E}\ \mathtt{then}\ \mathtt{S}_1\ \mathtt{else}\ \mathtt{S}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{if}\ \mathtt{E'}\ \mathtt{then}\ \mathtt{S}_1\ \mathtt{else}\ \mathtt{S}_2\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{branch-reduce}$$

$$\frac{}{(\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ \mathtt{S}_1\ \mathtt{else}\ \mathtt{S}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{S}_1\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{branch-left}$$

$$\frac{}{(\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ \mathtt{S}_1\ \mathtt{else}\ \mathtt{S}_2\ ,\ \Sigma\ ,\ \sigma) \longrightarrow (\mathtt{S}_2\ ,\ \Sigma\ ,\ \sigma)}\ \texttt{branch-right}$$

$$\frac{}{\texttt{(while E do S , }\Sigma\texttt{ , }\sigma\texttt{)} \longrightarrow \texttt{(if E then (S (while E do S)) else skip , }\Sigma\texttt{ , }\sigma\texttt{)}}\ \texttt{loop-unfo}$$

4.1.3 Local variables

$$\frac{\texttt{(S , }\Sigma\texttt{[x := nextref(x,}\Sigma\texttt{)] , }\sigma\texttt{)} \longrightarrow \texttt{(S , }\Sigma\texttt{[x := nextref(x,}\Sigma\texttt{)] , }\sigma\texttt{')}}{\texttt{(local x in S , }\Sigma\texttt{ , }\sigma\texttt{)} \longrightarrow \texttt{(local x in S' , }\Sigma\texttt{ , }\sigma\texttt{')}}\ \texttt{local-reduce}$$

$$\frac{}{\texttt{(local x in skip , }\Sigma\texttt{ , }\sigma\texttt{)} \longrightarrow \texttt{(skip , }\Sigma\texttt{ , }\sigma\texttt{[nextref(x,}\Sigma\texttt{) := undefined])}}\ \texttt{local-skip}$$

4.2 Big-step semantics of *While*

4.2.1 skip, assignment, composition

$$\frac{}{\texttt{(skip , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \sigma}\ \texttt{skip}$$

$$\frac{\texttt{(E , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \texttt{n}}{\texttt{(x := E , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \sigma\texttt{[}\Sigma\texttt{(x) := n]}}\ \texttt{assign}$$

$$\frac{\texttt{(S}_1\texttt{ , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \sigma\texttt{'} \quad \texttt{(S}_2\texttt{ , }\Sigma\texttt{ , }\sigma\texttt{')} \Downarrow \sigma\texttt{''}}{\texttt{(S}_1\texttt{ S}_2\texttt{ , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \sigma\texttt{''}}\ \texttt{composition}$$

4.2.2 Branch, loop

$$\frac{\texttt{(E , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \texttt{true} \quad \texttt{(S}_1\texttt{ , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \sigma\texttt{'}}{\texttt{(if E then S}_1\texttt{ else S}_2\texttt{ , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \sigma\texttt{'}}\ \texttt{branch-left}$$

$$\texttt{(E , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \texttt{false} \quad \texttt{(S}_2\texttt{ , }\Sigma\texttt{ , }\sigma\texttt{)} \Downarrow \sigma\texttt{'}$$

$$\frac{}{(\texttt{if E then } S_1 \texttt{ else } S_2 \ , \ \Sigma \ , \ \sigma) \Downarrow \sigma'} \ \texttt{branch-right}$$

$$\frac{(\texttt{E} \ , \ \Sigma \ , \ \sigma) \Downarrow \texttt{true} \quad (\texttt{S (while E do S)} \ , \ \Sigma \ , \ \sigma) \Downarrow \sigma'}{(\texttt{while E do S} \ , \ \Sigma \ , \ \sigma) \Downarrow \sigma'} \ \texttt{loop-do}$$

$$\frac{(\texttt{E} \ , \ \Sigma \ , \ \sigma) \Downarrow \texttt{false}}{(\texttt{while E do S} \ , \ \Sigma \ , \ \sigma) \Downarrow \sigma} \ \texttt{loop-skip}$$

4.2.3 Local variables

$$\frac{(\texttt{S} \ , \ \Sigma[\texttt{x} \coloneqq \texttt{nextref(x,}\Sigma\texttt{)}] \ , \ \sigma) \Downarrow \sigma'}{(\texttt{local x in S} \ , \ \Sigma \ , \ \sigma) \Downarrow \sigma'[\texttt{nextref(x,}\Sigma\texttt{)} \coloneqq \texttt{undefined}]} \ \texttt{local}$$