

Homework 2: Operational Semantics for WHILE

CS 252: Advanced Programming Languages
Prof. Thomas H. Austin
San José State University

1 Introduction

For this assignment, you will implement the semantics for a small imperative language, named WHILE.

The language for WHILE is given in Figure 1. Unlike the Bool* language we discussed previously, WHILE supports *mutable references*. The state of these references is maintained in a *store*, a mapping of references to values. (“Store” can be thought of as a synonym for heap.) Once we have mutable references, other language constructs become more useful, such as sequencing operations $(e_1; e_2)$.

2 Small-step semantics

The small-step semantics for WHILE are given in Figure 3. Most of these rules are fairly straightforward, but there are a couple of points to note with the [SS-WHILE] rule. First of all, this is the only rule that makes a more complex expression when it has finished. (This rule is much cleaner when specified with the big-step operational semantics.)

Secondly, note the final value of this expression once the while loop completes. It will *always* be **false** when it completes. We could have created a special value, such as **null**, or we could have made the while loop a statement that returns no value. Both choices, however, would complicate our language needlessly.

3 YOUR ASSIGNMENT

Part 1: Rewrite the operational semantic rules for WHILE in L^AT_EX to use big-step operational semantics instead. Submit both your L^AT_EX source and the generated PDF file.

Extend your semantics with features to handle boolean values. **Do not treat these as binary operators.** Specifically, add support for:

- **and**
- **or**
- **not**

The exact behavior of these new features is up to you, but should seem reasonable to most programmers.

Part 2: Once you have your semantics defined, download `WhileInterp.hs` and implement the `evaluate` function, as well as any additional functions you need. Your implementation must be consistent with your operational semantics, *including your extensions for **and**, **or**, and **not***. Also, you may not change any type signatures provided in the file.

Finally, implement the interpreter to match your semantics.

Zip all files together into `hw2.zip` and submit to Canvas.

$e ::=$	x v $x := e$ $e; e$ $e \text{ op } e$ $\text{if } e \text{ then } e \text{ else } e$ $\text{while } (e) \ e$	<i>Expressions</i> variables/addresses values assignment sequential expressions binary operations conditional expressions while expressions
$v ::=$	i b	<i>Values</i> integer values boolean values
$op ::=$	$+$ $-$ $*$ $/$ $>$ $>=$ $<$ $<=$	<i>Binary operators</i>

Figure 1: The WHILE language

Runtime Syntax:		$\sigma \in \text{Store} = \text{variable} \rightarrow v$
Evaluation Rules:		$\boxed{e, \sigma \rightarrow e', \sigma'}$
[SS-SEQCTX]	$\frac{e_1, \sigma \rightarrow e'_1, \sigma'}{e_1; e_2, \sigma \rightarrow e'_1; e_2, \sigma'}$	[SS-VAR] $\frac{x \in \text{domain}(\sigma) \quad \sigma(x) = v}{x, \sigma \rightarrow v, \sigma}$
[SS-SEQ]	$\frac{}{v; e, \sigma \rightarrow e, \sigma}$	[SS-ASSIGNCTX] $\frac{e_1, \sigma \rightarrow e'_1, \sigma'}{x := e, \sigma \rightarrow x := e', \sigma'}$
[SS-OPCTX1]	$\frac{e_1, \sigma \rightarrow e'_1, \sigma'}{e_1 \text{ op } e_2, \sigma \rightarrow e'_1 \text{ op } e_2, \sigma'}$	[SS-ASSIGN] $\frac{}{x := v, \sigma \rightarrow v, \sigma[x := v]}$
[SS-OPCTX2]	$\frac{e_2, \sigma \rightarrow e'_2, \sigma'}{v_1 \text{ op } e_2, \sigma \rightarrow v_1 \text{ op } e'_2, \sigma'}$	[SS-IFTRUE] $\frac{}{\text{if true then } e_1 \text{ else } e_2, \sigma \rightarrow e_1, \sigma}$
[SS-OP]	$\frac{v = v_1 \text{ op } v_2}{v_1 \text{ op } v_2, \sigma \rightarrow v, \sigma}$	[SS-IFFALSE] $\frac{}{\text{if false then } e_1 \text{ else } e_2, \sigma \rightarrow e_2, \sigma}$
[SS-IFCTX]	$\frac{e_1, \sigma \rightarrow e'_1, \sigma'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3, \sigma \rightarrow \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, \sigma'}$	
[SS-WHILE]	$\frac{}{\text{while } (e_1) \ e_2, \sigma \rightarrow \text{if } e_1 \text{ then } e_2; \text{while } (e_1) \ e_2 \text{ else false}, \sigma}$	

Figure 2: Small-step semantics for WHILE

Runtime Syntax:

$$\sigma \in Store = variable \rightarrow v$$

Evaluation Rules:

$$\boxed{e, \sigma \Downarrow e', \sigma'}$$

[B-VALUE]

$$\frac{}{v, \sigma \Downarrow v, \sigma}$$

[B-VAR]

$$\frac{x \in domain(\sigma) \quad \sigma(x) = v}{x, \sigma \Downarrow v, \sigma}$$

[B-ASSIGN]

$$\frac{e, \sigma \Downarrow v, \sigma'}{x := e, \sigma \Downarrow v, \sigma' [x := v]}$$

[B-SEQ]

$$\frac{\begin{array}{c} e_1, \sigma \Downarrow v_1, \sigma' \\ e_2, \sigma' \Downarrow v_2, \sigma'' \end{array}}{e_1; e_2, \sigma \Downarrow v_2, \sigma''}$$

[B-OP]

$$\frac{\begin{array}{c} e_1, \sigma \Downarrow v_1, \sigma' \\ e_2, \sigma' \Downarrow v_2, \sigma'' \\ v = apply(op, v_1, v_2) \end{array}}{e_1 \ op \ e_2, \sigma \Downarrow v, \sigma''}$$

[B-IFTRUE]

$$\frac{\begin{array}{c} e_1, \sigma \Downarrow \mathbf{true}, \sigma' \\ e_2, \sigma' \Downarrow v, \sigma'' \end{array}}{\mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3, \sigma \Downarrow v, \sigma''}$$

[B-IFFALSE]

$$\frac{\begin{array}{c} e_1, \sigma \Downarrow \mathbf{false}, \sigma' \\ e_3, \sigma' \Downarrow v, \sigma'' \end{array}}{\mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3, \sigma \Downarrow v, \sigma''}$$

[B-WHILETRUE]

$$\frac{\begin{array}{c} e_1, \sigma \Downarrow \mathbf{true}, \sigma' \\ e_2, \sigma' \Downarrow v, \sigma'' \end{array}}{\mathbf{while} \ (e_1) \ e_2, \sigma \Downarrow v, \sigma''}$$

[B-WHILEFALSE]

$$\frac{e_1, \sigma \Downarrow \mathbf{false}, \sigma'}{\mathbf{while} \ (e_1) \ e_2, \sigma \Downarrow \mathbf{false}, \sigma'}$$

Figure 3: Big-step semantics for WHILE