

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. For the sake of brevity, these rules use *evaluation contexts* (C), which specify which *redex* will be evaluated next. The evaluation rules then apply to the “hole” (\bullet) in this context.

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.

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 remove the contexts ($C[\dots]$) and to use evaluation order rules instead. Submit both your L^AT_EX source and the generated PDF file.

Extend your semantics with features to handle boolean values. 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
$\text{op} ::=$	$+$ $-$ $*$ $/$ $>$ $>=$ $<$ $<=$	<i>Binary operators</i>

Figure 1: The WHILE language

Runtime Syntax:

$C \in \text{Context} \quad ::= \quad C; e \mid C \text{ op } e \mid v \text{ op } C \mid x := C \mid \text{if } C \text{ then } e_1 \text{ else } e_2 \mid \bullet$
 $\sigma \in \text{Store} \quad = \quad \text{variable} \rightarrow v$

Evaluation Rules:

$e, \sigma \rightarrow e', \sigma'$

[SS-VAR]	$\frac{x \in \text{domain}(\sigma) \quad \sigma(x) = v}{C[x], \sigma \rightarrow C[v], \sigma}$
[SS-ASSIGN]	$\overline{C[x := v], \sigma \rightarrow C[v], \sigma[x := v]}$
[SS-OP]	$\frac{v = v_1 \text{ op } v_2}{C[v_1 \text{ op } v_2], \sigma \rightarrow C[v], \sigma}$
[SS-SEQ]	$\overline{C[v; e], \sigma \rightarrow C[e], \sigma}$
[SS-IFTRUE]	$\overline{C[\text{if true then } e_1 \text{ else } e_2], \sigma \rightarrow C[e_1], \sigma}$
[SS-IFFALSE]	$\overline{C[\text{if false then } e_1 \text{ else } e_2], \sigma \rightarrow C[e_2], \sigma}$
[SS-WHILE]	$\overline{C[\text{while } (e_1) \ e_2], \sigma \rightarrow C[\text{if } e_1 \text{ then } e_2; \text{while } (e_1) \ e_2 \text{ else false}], \sigma}$

Figure 2: Small-step semantics for WHILE