Homework 2: An Interpreter for SpartanLang

CS 152: Programming Language Paradigms
Prof. Thomas H. Austin
San José State University

1 Introduction

In this assignment, you will develop an interpreter for a small language, called SpartanLang. The valid expressions for SpartanLang are given in Figure 1.

SpartanLang supports *mutable references*. The state of these references is maintained in an *environment*, a mapping of variables to values. With mutable references, other language constructs become more useful, such as sequencing operations ($e_1; e_2$).

$e ::=$		*Expressions*
	v	values
	$x := e$	assignment
	$!x$	variable dereferencing (i.e. getting a variable's value)
	$e; e$	sequential expressions
	$e\ op\ e$	binary operations
	`if` e `then` e `else` e end	conditional expressions
	`while` e `do` e end	while expressions
$v ::=$		*Values*
	i	integer values
	b	boolean values
$op ::=$	$+ \mid - \mid * \mid > \mid >= \mid < \mid <=$	*Binary operators*

Figure 1: Expressions in SpartanLang

2 YOUR ASSIGNMENT

Create an interpreter for SpartanLang. You can assume that the parsing has been done for you – your job is to write an interpreter when given an abstract syntax tree (AST).

Download `interp.rkt` from the course website and implement the `evaluate` function, as well as any additional functions it uses. The file `test.rkt` provides some additional testing cases (though not necessarily a comprehensive set of tests). The files `parser.rkt` and `sl-run.rkt` allow you to run on programs written in text files; `test1.sparta` is provided as an example.

The input to evaluate consists of two arguments, an expression and the environment. The expression will be represented as a struct, which could be:

- `sp-val`, representing some constant (such as `42` or `true`).

- `sp-binop` represents a binary expression (e.g. $2 + 4$).
- `sp-if` represents a conditional expression (e.g. `if true then` 1 `else` 0 end).
- `sp-assign` represents an update to a variable (e.g. `x := 4`).
- `sp-var` represents getting the value of a variable, such as `!x`.
  Note: this is not the 'not' operator of languages like Java.
- **`sp-seq` represents a sequence of expressions, like `x:=1; y:=2`.**
- **`sp-while` represents a loop, like `while` $i < 10$ `do` $i := !i + 1$ end.**

The environment is an *immutable* map of variables to values.

The return value of evaluate is a *pair* of a value and a new environment. While we cannot mutate the map, we simulate imperative updates by creating a new environment based on the old environment[1].

Submit your modified `interp.rkt` to Canvas.

3 Operational Semantics

Evaluation Rules: $\boxed{e, \sigma \Downarrow v, \sigma'}$

[BS-VAR]
$$\frac{v = \sigma(x)}{x, \sigma \Downarrow v, \sigma}$$

[BS-VAL]
$$\frac{}{v, \sigma \Downarrow v, \sigma}$$

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

[BS-IF-TRUE]
$$\frac{e_1, \sigma \Downarrow \texttt{false}, \sigma_1 \qquad e_2, \sigma_1 \Downarrow v, \sigma'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \text{ end}, \sigma \Downarrow v, \sigma'}$$

[BS-SEQ]
$$\frac{e_1, \sigma \Downarrow v_1, \sigma_1 \qquad e_2, \sigma_1 \Downarrow v_2, \sigma'}{e_1 \ op \ e_2, \sigma \Downarrow v_2, \sigma'}$$

[BS-IF-FALSE]
$$\frac{e_1, \sigma \Downarrow \texttt{true}, \sigma_1 \qquad e_3, \sigma_1 \Downarrow v, \sigma'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \text{ end}, \sigma \Downarrow v, \sigma'}$$

[BS-OP]
$$\frac{e_1, \sigma \Downarrow v_1, \sigma_1 \qquad e_2, \sigma_1 \Downarrow v_2, \sigma' \qquad v = v_1 \ op \ v_2}{e_1 \ op \ e_2, \sigma \Downarrow v, \sigma'}$$

[BS-WHILE-FALSE]
$$\frac{e_1, \sigma \Downarrow \texttt{false}, \sigma'}{\texttt{while } e_1 \texttt{ do } e_2 \text{ end}, \sigma \Downarrow 0, \sigma'}$$

[BS-WHILE-TRUE]
$$\frac{e_1, \sigma \Downarrow \texttt{true}, \sigma_1 \qquad \texttt{while } e_1 \texttt{ do } e_2 \text{ end}, \sigma_1 \Downarrow v, \sigma'}{\texttt{while } e_1 \texttt{ do } e_2 \text{ end}, \sigma \Downarrow v, \sigma'}$$

Figure 2: Big-step semantics for SpartanLang

While the behavior of most of our constructs is straightforward, a formal semantics can be useful for clarifying the behavior of the language in corner cases.

Operational semantics allow us to specify the behavior of a language in concrete terms. In Figure 3, we are specifically using *big step* operational semantics. With big step, we define how to evaluate an expression

[1]This approach is similar to how Java deals with Strings. A String is immutable, but `String s2 = s.substring(2)` creates a new immutable String derived from the value of `s`.

e down to a value v in the context of an environment (or "store") σ. (For those not up on their Greek letters, this is "sigma".)

The big step relation is defined by

$$e, \sigma \Downarrow v, \sigma'$$

Note that this corresponds exactly with the specification for our `evaluate` method in Racket. Our input parameters are on the left hand side of the $\Downarrow$. We take an expression (e) and a store (σ) and evaluate them, returning a pair of a value (v) and a possible modified store (σ').

In each rule, the details above the lines are the *preconditions*. If these conditions hold, then the rule applies. For instance, if we were to translate the rule [BS-VAR] into English, we might say:

> If x maps to the value v in the store σ, then evaluating x in the context of store σ results in the value v and the unchanged store σ.

While these rules might be tricky to parse at first, your Racket code is essentially a straight translation of these rules.