Computer Science 131, Fall 2001
Assignment 4: Interpreting an Imperative Language

Out: Saturday, October 13

Due: Friday, October 19, 5:00pm

1 Introduction

For this problem, you are given a formal definition of execution for a very simple subset of a language like C.

This language is divided into *expressions*, which yield values and *commands* which update variables and handle control flow. A grammar for the abstract syntax is shown in Figure 1. The only command that may not be self-explanatory is the `skip` command, which has no effect when executed. The abstract syntax for this command usually represented the absence of a command, as when the C programmer writes

```
for (i=0; i<10; i++) ;
```

One difference about the expressions from those seen in class is that the environment maps *every* variable to a memory location. Uses of variables automatically refer to the location or the contents of the associated memory location as appropriate, depending on whether the variable is being used as an l-value or an r-value.

A second difference is that pairs evaluate to a location in memory where the pair is stored, rather than to the pair itself. This lets the language build up "linked lists" in the store when evaluating expressions such as

$$\langle 1, \langle 2, \langle 3, \texttt{null} \rangle \rangle \rangle$$

The evaluation rules for expressions in this language are shown in Figure 2 and the rules for commands are shown in Figure 3.

For example, the following code computes the factorial of 4:

```
newvar n := 4
in newvar answer := 1
   in while (not (n = 0)) do
      (answer := answer * n; n := n - 1)
```

Similarly, the following code creates the reverse of a linked list:

```
newvar list := ⟨1,⟨2,⟨3,null⟩⟩⟩
in newvar out := null
   in while (not (isNull in)) do
      (out := <fst in, out>;
      in := snd in)
```

2 The Exercise

First, in a file named `assign5.sml`, define an interpreter for the language as specified. This will require:

- Defining an abstract syntax for expressions and commands;
- Implementing environments and stores, including choosing a representation for locations and a way to get fresh locations on demand;
- Writing the evaluation functions for expressions and commands

Define a variable `reverse` which contains the abstract syntax representation of the above list-reversing example. In a comment, explain clearly what is in the final store after this program is evaluated; this should be readable by the graders without knowing exactly how your implementation chose to represent stores.

Submit this file using `cs131submit` as usual.

Expressions	e ::=	n	integers
	|	`tt`	true
	|	`ff`	false
	|	`null`	null location
	|	l	other locations
	|	x	variables
	|	$\langle e_1, e_2 \rangle$	pair
	|	`fst` e	extract first component
	|	`snd` e	extract second component
	|	`not` e	boolean negation
	|	`isNull` e	test for `null`
	|	e`*`e	integer product
	|	e`=`e	integer equality
Commands	c ::=	`skip`	no-op
	|	x `:=` e	assignment
	|	c `;` c	sequence of commands
	|	`if` e `then` c	if-then
	|	`while` e `do` c	while loop
	|	`newvar` x `:=` e `in` c	variable declaration
Values	v ::=	n	
	|	`null`	
	|	l	
	|	`tt`	
	|	`ff`	

Figure 1: SIL Syntax

$$\frac{}{(\rho, \sigma, v) \Downarrow (\sigma, v)} \tag{1}$$

$$\frac{}{(\rho, \sigma, x) \Downarrow (\sigma, (\sigma(\rho(x))))} \tag{2}$$

$$\frac{(\rho, \sigma, e_1) \Downarrow (\sigma', n_1) \qquad (\rho, \sigma', e_2) \Downarrow (\sigma'', n_2)}{(\rho, \sigma, e_1\texttt{*}e_2) \Downarrow (\sigma'', n_1 n_2)} \tag{3}$$

$$\frac{(\rho, \sigma, e_1) \Downarrow (\sigma', n_1) \qquad (\rho, \sigma', e_2) \Downarrow (\sigma'', n_2)}{(\rho, \sigma, e_1\texttt{=}e_2) \Downarrow (\sigma'', n_1 \equiv n_2)} \tag{4}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', b)}{(\rho, \sigma, \texttt{not}\ e) \Downarrow (\sigma', \neg b)} \tag{5}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', v)}{(\rho, \sigma, \texttt{isNull}\ e) \Downarrow (\sigma', v \equiv \texttt{null})} \tag{6}$$

$$\frac{(\rho, \sigma, e_1) \Downarrow (\sigma', n_1) \qquad (\rho, \sigma', e_2) \Downarrow (\sigma'', n_2) \qquad l \notin \operatorname{dom} \sigma''}{(\rho, \sigma, \langle e_1, e_2 \rangle) \Downarrow ((\sigma'', l{=}\langle v_1, v_2 \rangle), l)} \tag{7}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', l) \qquad \sigma'(l) = \langle v_1, v_2 \rangle}{(\rho, \sigma, \texttt{fst}\ e) \Downarrow (\sigma', v_1)} \tag{8}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', l) \qquad \sigma'(l) = \langle v_1, v_2 \rangle}{(\rho, \sigma, \texttt{snd}\ e) \Downarrow (\sigma', v_2)} \tag{9}$$

Figure 2: Evaluation Rules for Expressions

$$\frac{}{(\rho, \sigma, \texttt{skip}) \Downarrow \sigma} \tag{10}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', v) \qquad \rho(x) = l}{(\rho, \sigma, x \texttt{ := } e) \Downarrow (\sigma', l{=}v)} \tag{11}$$

$$\frac{(\rho, \sigma, c_1) \Downarrow \sigma' \qquad (\rho, \sigma', c_2)) \Downarrow \sigma''}{(\rho, \sigma, (c_1 \texttt{ ; } c_2)) \Downarrow \sigma''} \tag{12}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', \texttt{tt}) \qquad (\rho, \sigma', c) \Downarrow \sigma''}{(\rho, \sigma, \texttt{if } e \texttt{ then } c) \Downarrow \sigma''} \tag{13}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', \texttt{ff})}{(\rho, \sigma, \texttt{if } e \texttt{ then } c) \Downarrow \sigma'} \tag{14}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', \texttt{tt}) \qquad (\rho, \sigma', c) \Downarrow \sigma'' \qquad (\rho, \sigma'', \texttt{while } e \texttt{ do } c) \Downarrow \sigma'''}{(\rho, \sigma, \texttt{while } e \texttt{ do } c) \Downarrow \sigma'''} \tag{15}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', \texttt{ff})}{(\rho, \sigma, \texttt{while } e \texttt{ do } c) \Downarrow \sigma'} \tag{16}$$

$$\frac{(\rho, \sigma, e) \Downarrow (\sigma', v) \qquad ((\rho, x{=}l), (\sigma', l{=}v), c) \Downarrow \sigma'' \qquad l \notin \operatorname{dom} \sigma}{(\rho, \sigma, \texttt{newvar } x \texttt{ := } e \texttt{ in } c) \Downarrow \sigma''} \tag{17}$$

Figure 3: Evaluation Rules for Commands