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

```
⟨1, ⟨2, ⟨3, null⟩⟩⟩
```

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:
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 |
| | $\text{fst } e$ | extract first component |
| | $\text{snd } e$ | extract second component |
| | $\text{not } e$ | boolean negation |
| | $\text{isNull } e$ | test for null |
| | $e*e$ | integer product |
| | $e=e$ | integer equality |

| Commands | $c ::= \text{skip}$ | no-op |
| | $x := e$ | assignment |
| | $c ; c$ | sequence of commands |
| | $\text{if } e \text{ then } c$ | if-then |
| | $\text{while } e \text{ do } c$ | while loop |
| | $\text{newvar } x := e \text{ in } c$ | variable declaration |

| Values | $v ::= n$ |
| | $null$ |
| | $l$ |
| | $tt$ |
| | $ff$ |

Figure 1: SIL Syntax
\[
\begin{align*}
\frac{}{(\rho, \sigma, v) \Downarrow (\sigma, v)}
\end{align*}
\] (1)

\[
\begin{align*}
\frac{}{(\rho, \sigma, x) \Downarrow (\sigma, (\sigma(\rho(x))))}
\end{align*}
\] (2)

\[
\begin{align*}
\frac{(\rho, \sigma, e_1) \Downarrow (\sigma', n_1)}{(\rho, \sigma', e_2) \Downarrow (\sigma'', n_2)}
\end{align*}
\] (3)

\[
\begin{align*}
\frac{(\rho, \sigma, e_1) \Downarrow (\sigma', n_1)}{(\rho, \sigma', e_2) \Downarrow (\sigma'', n_2)}
\end{align*}
\] (4)

\[
\begin{align*}
\frac{(\rho, \sigma, e_1) \Downarrow (\sigma', n_1)}{(\rho, \sigma', e_2) \Downarrow (\sigma'', n_2)}
\end{align*}
\] (5)

\[
\begin{align*}
\frac{(\rho, \sigma, e) \Downarrow (\sigma', b)}{(\rho, \sigma, \text{not } e) \Downarrow (\sigma', \neg b)}
\end{align*}
\] (6)

\[
\begin{align*}
\frac{(\rho, \sigma, e_1) \Downarrow (\sigma', n_1)}{(\rho, \sigma', e_2) \Downarrow (\sigma'', n_2)}
\end{align*}
\] (7)

\[
\begin{align*}
\frac{(\rho, \sigma, e_1) \Downarrow (\sigma', n_1)}{(\rho, \sigma', e_2) \Downarrow (\sigma'', n_2)}
\end{align*}
\] (8)

\[
\begin{align*}
\frac{(\rho, \sigma, e_1) \Downarrow (\sigma', n_1)}{(\rho, \sigma', e_2) \Downarrow (\sigma'', n_2)}
\end{align*}
\] (9)

Figure 2: Evaluation Rules for Expressions
\begin{align*}
  (\rho, \sigma, \text{skip}) \Downarrow \sigma & \quad \text{(10)} \\
  \frac{ (\rho, \sigma, c) \Downarrow (\sigma', v) \quad \rho(x) = l }{ (\rho, \sigma, x := e) \Downarrow (\sigma', l = v) } & \quad \text{(11)} \\
  \frac{ (\rho, \sigma, c_1) \Downarrow \sigma' \quad (\rho, \sigma', c_2) \Downarrow \sigma'' }{ (\rho, \sigma, (c_1 ; c_2)) \Downarrow \sigma'' } & \quad \text{(12)} \\
  \frac{ (\rho, \sigma, e) \Downarrow (\sigma', \text{tt}) \quad (\rho, \sigma', c) \Downarrow \sigma'' }{ (\rho, \sigma, \text{if } e \text{ then } c) \Downarrow \sigma'' } & \quad \text{(13)} \\
  \frac{ (\rho, \sigma, e) \Downarrow (\sigma', \text{ff}) }{ (\rho, \sigma, \text{if } e \text{ then } c) \Downarrow \sigma' } & \quad \text{(14)} \\
  \frac{ (\rho, \sigma, e) \Downarrow (\sigma', \text{tt}) \quad (\rho, \sigma', c) \Downarrow \sigma'' \quad (\rho, \sigma'', \text{while } e \text{ do } c) \Downarrow \sigma''' }{ (\rho, \sigma, \text{while } e \text{ do } c) \Downarrow \sigma''' } & \quad \text{(15)} \\
  \frac{ (\rho, \sigma, e) \Downarrow (\sigma', \text{ff}) }{ (\rho, \sigma, \text{while } e \text{ do } c) \Downarrow \sigma' } & \quad \text{(16)} \\
  \frac{ (\rho, \sigma, e) \Downarrow (\sigma', v) \quad ((\rho, x = l), (\sigma', l = v), c) \Downarrow \sigma'' \quad l \notin \text{dom } \sigma }{ (\rho, \sigma, \text{newvar } x := e \text{ in } c) \Downarrow \sigma'' } & \quad \text{(17)} \\
\end{align*}

Figure 3: Evaluation Rules for Commands