Background

This assignment involves implementing the big-step and small-step operational semantics of Pitts’ LC language [Pit02]. (Also see chapter 3 of Hennessey’s notes [Hen14].) The CEK interpreter for LC will serve as our reference implementation of LC (to check against the two interpreters you will implement). We’ll make use of the following files:

- **LC.hs** defines (i) the abstract syntax of LC, (ii) utilities functions (e.g., aApply and bApply), and (iii) QuickCheck generators.
- **State.hs** defines the data structure for LC-states.
- **LCParser.hs** defines a parser for LC.
- **LCCEK.hs** an implementation of a CEK interpreter (from class).
- **LCbs.hs** an implementation of a big-step interpreter.
- **LCss.hs** a start at a small-step interpreter.

A key point. LC phrases are of three sorts: (a) arithmetic expressions, (b) boolean expressions, and (c) commands. The big-step interpreter deals with these different sorts by having three different evaluators: evalA (for arithmetic expressions), evalB (for boolean expressions), evalC (for commands). The small-step interpreter (and the CEK machine) deal with these three sorts of LC phrases via the data-type:

```
data Phrase = A AExp | B BExp | C Command
```

You can view A, B, and C as interpreter-specific tags. Whenever LC abstract syntax occurs in the interpreter, it must be tagged. (See the LCCEK.hs file for lots of examples.)

⚠️ Expect questions like Problems 1 and 2 on the next quiz.

Problems

- **Problem 1: (12 points). ▶️**
  Let \( s = [\ell_0 \mapsto 2, \ell_1 \mapsto 0, \ell_2 \mapsto 3] \).
  Give a full justification (i.e., proof) of each of the following small-step transitions:

  a. \( \langle (\{(3 + 2) \ast !\ell_2\} - (9 + 1), s) \rangle \rightarrow \langle (5 \ast !\ell_2) - (9 + 1), s \rangle \)
  b. \( \langle \ell_0 := 7; \ell_1 := 44, s \rangle \rightarrow \langle \text{skip}; \ell_1 := 44, s[\ell_0 \mapsto 7] \rangle \)
  c. \( \langle \text{if } !\ell_0 > 0 \text{ then } \ell_0 := !\ell_0 - 1 \text{ else } \ell_1 := !\ell_1 + 3, s \rangle \rightarrow \langle \text{if } 2 > 0 \text{ then } \ell_0 := !\ell_0 - 1 \text{ else } \ell_1 := !\ell_1 + 3, s \rangle \)

- **Problem 2: (8 points). ▶️**
  Again let \( s = [\ell_0 \mapsto 2, \ell_1 \mapsto 0, \ell_2 \mapsto 3] \).
  For (a) and (b) below, give the complete transition sequence starting from the configuration.

  a. \( \langle (\{(3 + 2) \ast !\ell_2\} - (9 + 1), s) \rangle \)
  b. \( \langle \text{if } !\ell_0 > 0 \text{ then } \ell_0 := !\ell_0 - 1 \text{ else } \ell_1 := !\ell_1 + 3, s \rangle \)

  Do not show the justification of each step, but, of course, each step must be justifiable from the small-step rules.

  Advice: Do these first by hand (for practice, understandings, and quiz-prep), then after finishing the next problem, use stepRun to check (and correct) your work.

- **Problem 3: (40 points). ▶️**

  a. **(20 Points)** Complete the small-step interpreter in LCss.hs. Just like in the big-step case, premises in rules show up as recursive-calls in the where-clauses in the implementation.

  b. **(5 Points)** Testing 1. Try: (stepRun fact state4) on your small-step interpreter; it runs a LC command for computing 4! and its final state should be:
\( s[0] = 4 \quad s[1] = 24 \quad s[2] = 0 \quad s[3] = 0 \quad s[4] = 0 \)

(c) \(5 \text{ Points}\) TESTING 2. Try: (quickCheck ss\_prop), which runs 100 random LC commands (sans \texttt{while}'s) on your small-step interpreter and on the CEK interpreter and compares the results. Your code should pass all 100 tests.

(d) \(10 \text{ Points}\) TESTING 3. Devise and run your own set of tests to make sure your implementation of \texttt{while}-loops is correct.

\[ \text{Problem 4: (40 points).} \]
Suppose we add a new command to LC:

\[ \texttt{repeat } C \texttt{ until } B \]

which in terms of the \texttt{do-while} command from C is equivalent to:
\[ \{ \texttt{do } C \texttt{ while (not } B \texttt{)} \} \]. A big-step semantics for \texttt{repeat-until}-commands is given at the bottom of this page. Recall that \texttt{ff} and \texttt{tf} are the LC constants for false and true, respectively.

\[ \text{(a) \(10 \text{ points}\) Extend the LC big-step evaluator of LCbs.hs to handle \texttt{repeat-until}-commands according to the big-step rules. (LC.hs and LCParse.hs already can handle \texttt{repeat-until}-commands.)} \]

\[ \text{(b) \(10 \text{ points}\) Come up with some convincing tests that your implementation for part (a) is correct.} \]

\[ \text{(c) \(10 \text{ points}\) Give small-step transition rules for \texttt{repeat-until}-commands.} \]

\[ \text{(d) \(10 \text{ points}\) Give the full derivation of the small-step transition relation starting from:} \]
\[ \{ \texttt{repeat } ℓ := ℓ - 1 \texttt{ until } (ℓ == 0),[ℓ → 1] \} \]

\begin{align*}
\text{Repeat}_1: \quad & \langle C, s_0 \rangle \Downarrow \langle \text{skip}, s_1 \rangle \quad \langle B, s_1 \rangle \Downarrow \langle \text{tt}, s_2 \rangle \\
& \langle \text{repeat } C \text{ until } B, s_0 \rangle \Downarrow \langle \text{skip}, s_2 \rangle \\
\text{Repeat}_2: \quad & \langle C, s_0 \rangle \Downarrow \langle \text{skip}, s_1 \rangle \quad \langle B, s_1 \rangle \Downarrow \langle \text{ff}, s_2 \rangle \\
& \langle \text{repeat } C \text{ until } B, s_0 \rangle \Downarrow \langle \text{skip}, s_2 \rangle}
\end{align*}
LC: Big-steps rules

- \[ \downarrow\text{Con}: \quad \langle c, s \rangle \downarrow \langle c, s \rangle \quad (c \in \mathbb{Z} \cup \mathbb{B}) \]

- \[ \downarrow\text{⊕}: \quad \langle E_1, s \rangle \downarrow (n_1, s') \quad \langle E_2, s' \rangle \downarrow (n_2, s'') \quad \langle E_1 \oplus E_2, s \rangle \downarrow \langle c, s'' \rangle \quad (c = n_1 \oplus n_2) \]

- \[ \downarrow\text{Skip}: \quad \langle \text{skip}, s \rangle \downarrow (\text{skip}, s) \]

- \[ \downarrow\text{Loc}: \quad \langle \ell, s \rangle \downarrow (s(\ell), s) \quad (\ell \in \text{dom}(s)) \]

- \[ \downarrow\text{Set}: \quad \langle \ell \leftarrow E, s \rangle \downarrow (\text{skip}, s'[\ell \mapsto n]) \]

- \[ \downarrow\text{Seq}: \quad \langle C_1, s \rangle \downarrow (\text{skip}, s') \quad \langle C_2, s' \rangle \downarrow (\text{skip}, s'') \quad \langle C_1; C_2, s \rangle \downarrow (\text{skip}, s'[\ell \mapsto n]) \]

- \[ \downarrow\text{If1}: \quad \langle B, s \rangle \downarrow (\text{true}, s') \quad \langle C_1, s' \rangle \downarrow (\text{skip}, s'') \quad \langle \text{if } B \text{ then } C_1 \text{ else } C_2, s \rangle \downarrow (\text{skip}, s'[\ell \mapsto n]) \]

- \[ \downarrow\text{If2}: \quad \langle B, s \rangle \downarrow (\text{false}, s') \quad \langle C_2, s' \rangle \downarrow (\text{skip}, s'') \quad \langle \text{if } B \text{ then } C_1 \text{ else } C_2, s \rangle \downarrow (\text{skip}, s'[\ell \mapsto n]) \]

- \[ \downarrow\text{While1}: \quad \langle B, s \rangle \downarrow (\text{true}, s') \quad \langle C, s' \rangle \downarrow (\text{skip}, s'') \quad \langle \text{while } B \text{ do } C, s \rangle \downarrow (\text{skip}, s'[\ell \mapsto n]) \quad \langle \text{while } B \text{ do } C, s \rangle \downarrow (\text{skip}, s'[\ell \mapsto n]) \]

- \[ \downarrow\text{While2}: \quad \langle B, s \rangle \downarrow (\text{false}, s') \quad \langle \text{while } B \text{ do } C, s \rangle \downarrow (\text{skip}, s') \]

LC: Small-steps rules

- \[ \rightarrow\text{op1}: \quad \langle E_1, s \rangle \rightarrow \langle E'_1, s' \rangle \quad \langle E_2, s \rangle \rightarrow \langle E'_2, s' \rangle \quad \langle E_1 \oplus E_2, s \rangle \rightarrow \langle E'_1 \oplus E'_2, s' \rangle \]

- \[ \rightarrow\text{op2}: \quad \langle n_1 \oplus n_2, s \rangle \rightarrow \langle c, s \rangle \quad (c = n_1 \oplus n_2) \]

- \[ \rightarrow\text{loc}: \quad \langle \ell, s \rangle \rightarrow (s(\ell), s) \quad (\ell \in \text{dom}(s)) \]

- \[ \rightarrow\text{seq1}: \quad \langle C_1, s \rangle \rightarrow \langle C'_1, s' \rangle \quad \langle C_1; C_2, s \rangle \rightarrow \langle C'_1; C'_2, s' \rangle \]

- \[ \rightarrow\text{seq2}: \quad \langle \text{skip}; C, s \rangle \rightarrow (C, s) \quad \langle B, s \rangle \rightarrow (B', s') \quad \langle \text{if } B \text{ then } C_1 \text{ else } C_2, s \rangle \rightarrow \langle \text{if } B' \text{ then } C_1 \text{ else } C_2, s' \rangle \]

- \[ \rightarrow\text{If1}: \quad \langle \text{if } B \text{ then } C_1 \text{ else } C_2, s \rangle \rightarrow \langle \text{if } B' \text{ then } C_1 \text{ else } C_2, s' \rangle \]

- \[ \rightarrow\text{If2}: \quad \langle \text{if true then } C_1 \text{ else } C_2, s \rangle \rightarrow \langle C_1, s \rangle \quad \langle \text{if false then } C_1 \text{ else } C_2, s \rangle \rightarrow \langle C_2, s \rangle \]

- \[ \rightarrow\text{While}: \quad \langle \text{while } B \text{ do } C, s \rangle \rightarrow \langle \text{if } B \text{ then } C \text{ while } B \text{ do } C \text{ else skip}, s \rangle \]

Correction: The previous version has \( \downarrow \) in place of \( \rightarrow \) in the small-step rules.
Testing Tools

- In the current version of LCbs.hs.
  - Look at the bottom of the file.
  - There you will find some sample integer expressions (\textit{ie0},...,,\textit{ie3}), boolean expressions (\textit{be0},...,,\textit{be3}), and commands (\textit{cmd0},...,,\textit{cmd8}).
  - Evaluate these use \texttt{runI} (for integer expressions), \texttt{runB} (for boolean expressions), and \texttt{run} (for commands). Each of these will return the final configuration of the evaluation. For example:

  \begin{verbatim}
  *Main> ie2
  "val(x1)+2"
  *Main> runI ie2 state0
  (2,fromList [(0,1),(1,0),(2,3)])
  *Main> cmd3
  "{ x0 := (-2); x3 := ((-3)+val(x1)) }"
  *Main> run cmd3 state0
  (skip,fromList [(0,-2),(1,0),(2,3),(3,-3)])
  \end{verbatim}

- In the current version of LCss.hs.
  - Look at the bottom of the file.
  - There you will also find our friends \textit{ie0},...,,\textit{cmd8}.
  - To run a command and just get the final configuration, use \texttt{run’}.
  - To get a trace of the computation, use \texttt{stepRunI’},\texttt{stepRunB’}, and \texttt{stepRun}. For example:

  \begin{verbatim}
  *Main> cmd3
  "{ x0 := (-2); x3 := ((-3)+val(x1)) }"
  *Main> run’ cmd3 state0
  Step: -1
  C skip
  *Main> stepRun’ cmd3 state0
  Step: 0
  C { x0 := (-2); x3 := ((-3)+val(x1)) }
  s[0]=1 s[1]=0 s[2]=3 <tap return>
  Step: 1
  C { skip; x3 := ((-3)+val(x1)) }
  s[0]=-2 s[1]=0 s[2]=3 <tap return>
  Step: 2
  C x3 := ((-3)+val(x1))
  s[0]=-2 s[1]=0 s[2]=3 <tap return>
  Step: 3
  C x3 := ((-3)+0)
  s[0]=-2 s[1]=0 s[2]=3 <tap return>
  Step: 4
  C x3 := (-3)
  s[0]=-2 s[1]=0 s[2]=3 <tap return>
  Step: 5
  C skip
  \end{verbatim}

Confession: The “Step: -1” thing in the output of \texttt{run’} is me being lazy and reusing a function from the step-by-step output code.