Homework 6: Big- & small-step semantics for LC
CIS 352: Programming Languages
27 February 2016, Version 2

Administrivia

- When you trade ideas with another student, document it in your source file.
- Turn in the Part 1 problems (on paper) via the CIS 352 box on the 4th floor of SciTech. Include a cover sheet.
- Turn in the Part 2 problems via Blackboard. Include (i) the source files, (ii) the transcripts of test runs, and (iii) the cover sheet.

Grading Criteria

- The homework is out of 100 points.
- Each programming problem is \( \approx 70\% \) correctness and \( \approx 30\% \) testing.
- Omitting your name(s) in the source code looses you 5 points.
- When you trade ideas with another student, document it in your source file.
- Turn in the Part 1 problems (on paper) via the CIS 352 box on the 4th floor of SciTech. Include a cover sheet.
- Turn in the Part 2 problems via Blackboard. Include (i) the source files, (ii) the transcripts of test runs, and (iii) the cover sheet.

Background

This assignment involves using, extending, and implementing the big-step and small-step operational semantics of Pitts' LC language. Also see the rules reference on page 4 below and chapter 3 of Hennessy's notes.

Part I: Problems on Paper

Notation: \( s_{i,j,k} \) = the state \( \{ \ell_0 \mapsto i, \ell_1 \mapsto j, \ell_2 \mapsto k \} \). E.g., \( s_{3,24,0} \) = the state \( \{ \ell_0 \mapsto 3, \ell_1 \mapsto 24, \ell_2 \mapsto 0 \} \).

\( \blacklozenge \) Problem 1 (12 points) \( \blacklozenge \)

Give a full justification (i.e., proof) of each of the following small-step transitions:

(a) \( \langle ((!\ell_2 \cdot 6) - (2 \cdot 2)), s_{1,0,5} \rangle \rightarrow \langle ((5 \cdot 6) - (2 \cdot 2)), s_{1,0,5} \rangle \)
(b) \( \langle \ell_0 := 11; \text{skip}, s_{5,4,3} \rangle \rightarrow \langle \text{skip; skip}, s_{11,4,3} \rangle \)
(c) \( \langle \text{if } !\ell_0 \neq 0 \text{ then } \{ \ell_0 := !\ell_0 - 1; \ell_1 := 23 \} \text{ else skip}, s_{3,4,5} \rangle \)
\( \rightarrow \langle \text{if } 3 \neq 0 \text{ then } \{ \ell_0 := !\ell_0 - 1; \ell_1 := 23 \} \text{ else skip}, s_{3,4,5} \rangle \)

\( \blacklozenge \) Problem 2 (8 points) \( \blacklozenge \)

For (a) and (b) below, give the complete transition sequence starting from the configuration.

(a) \( \langle ((!\ell_2 \cdot 6) - (2 \cdot 2)), s_{1,0,5} \rangle \)
(b) \( \langle \text{if } !\ell_0 \neq 0 \text{ then } \{ \ell_0 := !\ell_0 - 1; \ell_1 := 23 \} \text{ else skip}, s_{3,4,5} \rangle \)

Do not show the justification of each step, but, of course, each step must be justifiable from the small-step rules.
Suppose we add a new command to LC:

\[ \text{iter } C \text{ by } E \]

which is executed at a given state by first evaluating \( E \) to an integer value \( n \) and then executing \( C \) \( n \)-many times, provided \( n > 0 \); when \( n \leq 0 \), a \( \text{skip} \) is executed. Formally:

\[
\begin{align*}
\text{Iter}_1: & \quad \langle E, s \rangle \downarrow \langle n, s' \rangle \\
& \quad \langle \text{iter } C \text{ by } E, s \rangle \downarrow \langle \text{skip}, s' \rangle \quad (n \leq 0)
\end{align*}
\]

\[
\begin{align*}
\text{Iter}_2: & \quad \langle E, s \rangle \downarrow \langle n, s_1 \rangle \\
& \quad \langle C; \text{iter } C \text{ by } n', s_1 \rangle \downarrow \langle \text{skip}, s' \rangle \\
& \quad \langle \text{iter } C \text{ by } E, s \rangle \downarrow \langle \text{skip}, s' \rangle \quad (n > 0 \& n' = n - 1)
\end{align*}
\]

(a) (10 points) Give small-step transition rules for \( \text{iter} \)-commands.

(b) (10 points) Give the full derivation of the small-step transition relation starting from: \( \langle \text{iter } \ell := !\ell + 1 \text{ by } !\ell', [\ell \mapsto \ell', \ell' \mapsto 2] \rangle \)

Part II: Implementation Problems

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:³

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC.hs</td>
<td>contains the abstract syntax of LC, utilities functions (e.g., aApply), and QuickCheck generators.</td>
</tr>
<tr>
<td>State.hs</td>
<td>contains the data structure for LC-states.</td>
</tr>
<tr>
<td>LCParser.hs</td>
<td>contains a parser for LC.</td>
</tr>
<tr>
<td>LCCEK.hs</td>
<td>contains a CEK interpreter (from class).</td>
</tr>
<tr>
<td>LCbs.hs</td>
<td>contains a big-step interpreter.</td>
</tr>
<tr>
<td>LCss.hs</td>
<td>contains a start at a small-step interpreter.</td>
</tr>
</tbody>
</table>

³From: http://www.cis.syr.edu/courses/cis352/code/LC/. You will be modifying just LCbs.hs and LCss.hs.

\[ \text{LC phrases} \quad \text{These are of three sorts: (A) arithmetic expressions, (B) boolean expressions, and (C) commands.} \]

The big-step interpreter deals with these different sorts by having three different evaluators: evalA, evalB, and evalC for, respectively, Arithmetic expressions, Boolean expressions, and 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.

ปลาย Problem 4 (40 points)
(a) (20 Points) Complete the small-step interpreter in LCss.hs. ¹
(b) (5 Points) Testing 1: Try: (stepRunC' fact state4) on your small-step interpreter; it runs a LC command for computing 4! and its final state should be:

\[ s[0]=0 \quad s[1]=24 \quad s[2]=0 \quad s[3]=0 \quad s[4]=0 \]

(c) (5 Points) Testing 2: Try: (quickCheck ss_prop), which runs 100 random LC commands (sans 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 Points) Testing 3. Devise and run your own set of tests to make sure your implementation of while-loops is correct.

ปลาย Problem 5 (20 points)
This is the continuation of Problem 3.
(a) (10 points) Extend the LC big-step evaluator of LCbs.hs to handle iter-commands according to the rules given in Figure 1.
(b) (10 points) Come up with some convincing tests that your implementation for part (a) is correct.

Challenge Problems

_challenge Problem 1: (10 points).।
Do Exercise 3.4.2 on page 34 in Pitts notes. This is not than hard, but you need to think things through carefully!

_challenge Problem 2: (10 points).।
Modify the big-step interpreter to implement the rules of the previous problems. Come up with some convincing tests that your implementation is correct.

References


¹ Just like in the big-step case, premises in rules show up as recursive-calls in the where-clauses in the implementation.
Rules reference

**LC: Big-steps rules**

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

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

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

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

\[ \downarrow \text{-If}_1: \langle B, s \rangle \downarrow \langle \text{true}, s' \rangle \quad \langle C_1, s' \rangle \downarrow \langle \text{skip}, s'' \rangle \quad \text{if } B \text{ then } C_1 \text{ else } C_2, s \downarrow \langle \text{skip}, s'' \rangle \]

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

\[ \downarrow \text{-While}_1: \langle B, s \rangle \downarrow \langle \text{true}, s' \rangle \quad \langle C, s' \rangle \downarrow \langle \text{skip}, s'' \rangle \quad \langle \text{while } B \text{ do } C, s \rangle \downarrow \langle \text{skip}, s'' \rangle \]

**LC: Small-steps rules**

\[ \rightarrow \text{-op}_1: \langle E_1, s \rangle \rightarrow \langle E'_1, s' \rangle \quad \langle E_1 \odot E_2, s \rangle \rightarrow \langle E'_1 \odot E_2, s' \rangle \]

\[ \rightarrow \text{-op}_2: \langle E_2, s \rangle \rightarrow \langle E'_2, s' \rangle \quad \langle n_1 \odot E_2, s \rangle \rightarrow \langle n_1 \odot E'_2, s' \rangle \]

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

\[ \rightarrow \text{-set}_1: \langle \ell \leftarrow E, s \rangle \rightarrow \langle \ell \leftarrow E', s' \rangle \]

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

\[ \rightarrow \text{-if}_1: \langle B, s \rangle \rightarrow \langle B', s' \rangle \quad \text{if } B \text{ then } C_1 \text{ else } C_2, s \rightarrow \langle \text{if } B' \text{ then } C_1 \text{ else } C_2, s' \rangle \]

\[ \rightarrow \text{-if}_2: \langle \text{if true then } C_1 \text{ else } C_2, s \rangle \rightarrow \langle C_1, s \rangle \quad \langle \text{if false then } C_1 \text{ else } C_2, s \rangle \rightarrow \langle C_2, s \rangle \]

\[ \rightarrow \text{while}: \langle \text{while } B \text{ do } C, s \rangle \rightarrow \langle \text{if } B \text{ then } \{ C; \text{ while } B \text{ do } \} \text{ else } \text{skip}, s \rangle \]
Testing Tools

Tools for LCbs.hs

- At the bottom of LCbs.hs you will find some sample integer expressions (ie0, ..., ie3), boolean expressions (be0, ..., be3), and commands (cmd0, ..., cmd8).
- Evaluate these use runA (for arithmetic expressions), runB (for boolean expressions), and runC (for commands). Each of these will return the final configuration of the evaluation. For example:

```
*Main> ie2
"val(x1)+2"
*Main> runA ie2 state0
(2,fromList [((0,1),(1,0),(2,3))])
*Main> cmd3
"{ x0 := (-2); x3 := ((-3)+val(x1)) }"
*Main> runC cmd3 state0
(skip,fromList [((0,-2),(1,0),(2,3),(3,-3))])
```

Tools for LCss.hs

- At the bottom of LCss.hs you will find our friends ie0, ..., cmd8.
- To run a command and just get the final configuration, use run’. For example:

```
*Main> cmd3
"{ x0 := (-2); x3 := ((-3)+val(x1)) }"
*Main> run’ cmd3 state0
Step: -1
C skip
```

- To get a trace of the computation, use stepRunA’, stepRunB’, and stepRunC’. For example:

```
*Main> stepRunC’ 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>
```

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