Lab: Lists using pairs: the LIST-LAB language
CSC 312, “Programming language implementation”
Department of Computer Science · Grinnell College
September 28, 2015

Today’s pair-programming partnerships are:
• Mr. Boateng Asante and Mr. Dewey
• Mr. Nordin and Mr. Zahid
• Ms. Liddell and Mr. Schlager
• Mr. Champagne and Mr. Mataire
• Mr. Mulhall and Ms. DeWitt
• Ms. Byrne and Ms. Jervis
• Ms. Kyaruzi

In each pair, the partner listed first should log in and manage the keyboard and screen.

Exercises

A. Create a subdirectory called \texttt{LIST-LAB} in the directory you’re using for labs in this course and copy my implementation of the LIST-LAB programming language into it. The \texttt{/home/stone/courses/languages/code/LIST-LAB} directory contains seven language-specific modules:

\begin{itemize}
\item \texttt{expvals-and-environments.scm}
\item \texttt{stores.scm}
\item \texttt{tokens.scm}
\item \texttt{syntax-trees.scm}
\item \texttt{scanner.scm}
\item \texttt{parser.scm}
\item \texttt{interpreter.scm}
\end{itemize}

B. LIST-LAB is an extension of the MUTABLE-PAIRS language described in section 4.4 of the textbook, with the addition of a single new expressed value, the \texttt{null} value, and two new expression types that deal with this value: (a) a \texttt{null}-expression, which is simply the keyword \texttt{null}, and always denotes the null value; and (b) a \texttt{null?}-expression, with the syntax

\[
\texttt{<expression>} ::= \texttt{null? ( <expression> )}
\]

which evaluates its subexpression and has the true Boolean value if the value of the subexpression is the null value, and the false Boolean value if the subexpression has any non-null value.

The presence and detectability of the null value makes it possible to implement list structures in LIST-LAB as they are implemented in Scheme: The null value represents the empty list, and a non-empty list is represented by a pair with its first element as the left component of the pair and a list containing all the subsequent elements as the right component of a pair.

(1) Assuming this representation, the LIST-LAB language already contains exact analogues of Scheme’s \texttt{cons}, \texttt{car}, \texttt{cdr}, \texttt{set-car!}, and \texttt{set-cdr!} procedures. What are they?

(2) Write a LIST-LAB expression of which the value is a three-element list comprising the numbers 42, 23, and 137 in that order.
(3) Write a LIST-LAB expression of which the value is a procedure that finds the second element of a list of two or more elements. Then write and run a LIST-LAB program in which the body is a call-expression applying that procedure to the three-element list you constructed in the preceding step.

(4) Write a LIST-LAB program that uses a letrec-expression to define a recursive procedure length that determines the length of a list. In the body of the letrec-expression, apply that procedure to the three-element list you constructed in step (2).

C. (1) Revise the init-env procedure (in the expvals-and-environments.scm file) so that, in addition to the bindings for i, v, and x, the initial environment binds the name lor (“left-of-right”) to a storage location containing the procedure that you defined in part (3) of the preceding exercise. Check to make sure that your revision had the correct effect. Re-run the interpreter and test a few calls to lor.

(2) Revise the init-env procedure so that the initial environment also arranges for the identifier length to be bound to a storage location containing the recursive procedure that you defined in part (4) of the preceding exercise. (Hint: since you’ll be using extend-env-rec to construct the environment this time, you won’t need newref this time.) Check to make sure that your revision had the correct effect. Also confirm that the length of the empty list is 0 under your definition.

D. What other list procedures are sufficiently useful and common to be included in the initial environment for LIST-LAB programs? Define them and add them to the initial environment. Use currying for any that take two arguments (such as list-ref and append), and use let- and letrec-expressions to define local helper functions as needed (in the definition of reverse, for instance).