Lab: Scheme-Like Lists
CSC 161, “Imperative Problem Solving and Data Structures”
Department of Computer Science · Grinnell College
April 12, 2016

Pairs

- Yuyin Sun and Tanner Tufto
- Sophie Gaschott and Zachary Susag
- Lex Martin and Saung Thuya
- Tyler Williams and Colin Greenman
- Anna Blindermann and Ying Zhang
- Ella Nicolson and Eli Salm
- Jae Hong Shin and Dennis Chan
- Jong Hoon Bae and Cory McCartan
- Josh Lavin and Gemma Nash
- Faizaan Ali and Lilly Webster
- Erhaan Ahmad and Sanjay Sudhir

Mattori Birnbaum will take the place of anyone who is absent.

Exercises

These exercises use the struct node and list types from Monday’s lab.

Exercise 000: Write and test a last function that returns the last integer in a given non-empty list of integers.

Exercise 001: Write and test a last_node function that returns a pointer to the last node in a given non-empty list of integers.

C offers us a choice of two strategies for concatenating two lists to form one larger one. One approach involves allocating a new node for each element of the first list, adding each new node to the beginning of the second list one at a time. In Scheme, using recursion, the code would look like this:

\[
\text{(define concatenate}
\quad \text{(lambda (left right)}
\quad \text{(if (null? left)}
\quad \quad \text{right}
\quad \quad \text{(cons (car left) (concatenate (cdr left) right))})
\text{))}
\]

(This is basically how the two-argument form of the built-in append procedure is written.)

Exercise 010: Implement the concatenate function in C, using a similar recursive structure. The function should return a list that is NULL if both arguments are empty lists and otherwise is a pointer to the initial node of a list containing all of the elements of both of the given lists. Test your code.

The other approach is to use last_node to find the node at the end of left (unless left is the empty list) and then to overwrite the NULL pointer in the next field of that node with a copy of right — which is a pointer to the beginning of the other list. In this implementation, it is not necessary to allocate any new nodes. All of the nodes in the result list will be nodes that also occurred in the given list.

Exercise 011: Implement and test this approach, calling your function d_concatenate rather than concatenate.
The \texttt{d} in \texttt{d_concatenate} stands for “destructive,” because overwriting the NULL at the end of \texttt{left} has a kind of side effect on any other list pointers that point at nodes within \texttt{left}: The elements of \texttt{right} have, in effect, been concatenated onto all of those lists, because their common NULL terminator was replaced. The \texttt{concatenate} function is “non-destructive” because it doesn’t disrupt the values of other list variables in this way.

Exercise 100: Write a non-destructive function \texttt{reverse_list} in C similar to Scheme’s \texttt{reverse} procedure: Given a list, it constructs and returns an entirely separate list containing the same int values, but in the opposite order.

Hint: A Scheme version would look like this:

\begin{verbatim}
(define reverse-list
  (lambda (ls)
    (let helper ((so-far '())
                    (rest ls))
      (if (null? rest)
          so-far
          (helper (cons (car rest) so-far) (cdr rest))))))
\end{verbatim}

Exercise 101: It is also possible for a destructive list-reversing function to take a list and replace all the next fields in the nodes making up that list so that those nodes are linked together in the opposite of their original order. Draw a box-and-pointer diagram of a list containing the int values 12, 29, 46, and 52 and another diagram showing that same nodes, containing the same datum fields, but with their next fields relinked to impose the opposite order on the elements. Is there a sensible way to keep track of the changes that need to be made while traversing the original list?

Exercise 110: Implement the destructive list-reversing function (under the name \texttt{d_reverse_list} in C and test the result. Make sure to return a pointer to the node at the front of the relinked list (or a NULL pointer, if the function is given an empty list).

Exercise 111: Write a C program that first sets up two lists of length 2, one (called \texttt{lows}) containing the values 12 and 29, the other (called \texttt{highs}) containing the values 46 and 52, then calls \texttt{d_concatenate} to combine these lists (assigning the result to another list variable, \texttt{combined}), and finally calls \texttt{d_reverse_list} on the combined list (assigning the result to yet another list variable, \texttt{reversed}). At the end, display (with appropriate labels) the lists \texttt{lows}, \texttt{highs}, \texttt{combined}, and \texttt{reversed}. Explain how the side effects of \texttt{d_concatenate} and \texttt{d_reverse_list} produce the results you see.