Pairs

Here are the programmer pairs for today’s lab:

• Dennis Chan and Linda Oyolu
• Marcel Champagne and Matt Jasaitis
• Michael Owusu and Samee Zahid
• Kevin Connors and Julia Fay
• Danielle Williams and Lindsey Byrne
• Anita DeWitt and Thu Nguyen
• Ella Nicolson and Madeleine Goldman
• Adam Hudson and Amanda Hinchman-Dominguez
• Bazil Mupisiri and Adam Wesely
• Jacob Ekstrand and Devin Dooley
• Muhammad Hamza and Jong Hoon Bae
• Tyler Williams and David Chang
• Elizabeth Zak and Corey Simmonds

Kathryn Yetter will fill in for anyone who is absent.

Exercises

Exercise 0: Open up DrRacket and change the language specification at the top of the Definitions window to #!r6rs. In the Definitions window, type an import form that imports the (rnrs base) and (rnrs lists) libraries. Click on the Run button.

Exercise 1 (Mapping over finite sequences): (a) Without using map, define and test a Scheme procedure string-lengths that takes a list of strings as its argument and returns a list containing the lengths of those strings. For example, the call (string-lengths (list "foo" "bar" "baz" "quux")) has the value (3 3 3 4).

(b) Define and test another version the same procedure, this time relying on the map procedure (in the (rnrs base) library) to manage the recursion.
Exercise 2 (Filtering and partitioning): (a) Without using filter or partition, define and test a Scheme procedure positive-only that takes a list of integers as its argument and returns a list containing just the elements of that list that are positive integers, in the same relative order. For example, the call (positive-only (list 5 -16 2 3 -5 -3 0 1 4)) has the value (5 2 3 1 4).

(b) Define and test the same procedure, this time relying on the filter procedure (in the (rnrs lists) library) to manage the recursion and selection of elements.

(c) In your solution to part (b) change the identifier filter to partition and try your test cases again. What does the partition procedure do?

Exercise 3 (for-all and exists.) (a) Without using for-all or exists, define and test a Scheme procedure list<? that takes two lists of real numbers and returns a Boolean value indicating whether each element of the first list is strictly less than the element in the corresponding position in the second list. It is a precondition of the list<? procedure that the arguments it receives are lists of equal length.

So, for instance, the call (list<? '(5 7 4 3) '(6 12 19 4)) has the value #t, since 5 is less than 6, 7 is less than 12, 4 is less than 19, and 3 is less than 4. But (list<? '(3 8 2) '(9 1 7)) has the value #f, since 8 is not less than 1.

(b) Define and test the same procedure, this time relying on the for-all procedure to manage the recursion.

(c) In your solution to part (b), change the identifier for-all to exists and try your test cases again. How are the for-all and exists procedures related?

Folding

The fold-right procedure abstracts a more general recursive design pattern over lists. When you give it a “combiner” procedure, a result to be returned in the base case, and a list of values, fold-right constructs a recursive procedure and applies it to the given list, returning the given base-case value when the list is empty and applying the combiner to the first element of the list and the result of applying the same procedure recursively to the rest of the list. For example, one could define the sum procedure

(define (sum ls)
  (fold-right + 0 ls))
and \texttt{fold-right} would manage the recursion, supplying the value 0 in the base case and otherwise adding the car of \texttt{ls} to the result of a recursive call to find the sum of the \texttt{cdr} of \texttt{ls}. In this case, the combiner is a predefined procedure, \texttt{+}, but in most uses of \texttt{fold-right} the combiner is either an anonymous procedure denoted by a \texttt{lambda}-expression or a programmer-defined procedure.

Exercise 4: \texttt{fold-right}. Using \texttt{fold-right}, define a procedure called \texttt{magnitude} that takes a list of real numbers as its argument and returns the non-negative square root of the sum of the squares of those numbers.

Exercise 5: Work out by hand the value of the call \((\texttt{fold-right cons }'() ' (0 1 2 3 4))\). Check your result using DrRacket; if DrRacket gives you an unexpected answer, account for the disparity.

The \texttt{fold-left} procedure abstracts a related design pattern for lists, one that uses tail recursion to accumulate a result in an extra parameter.

First, here’s the design pattern, with \texttt{BASE-RESULT} and \texttt{UPDATER} as the placeholders for the components of the pattern that vary from one instance to another:

\begin{verbatim}
(define (accumulator ls)
  (let loop ((so-far BASE-RESULT)
            (rest ls))
    (if (null? rest)
        so-far
        (loop (UPDATER so-far (car rest)) (cdr rest))))
)
\end{verbatim}

For example, to get a procedure that concatenates all of the strings in a list, placing a colon after each one, we could write

\begin{verbatim}
(define (concatenator-with-colons ls)
  (let loop ((so-far "")
            (rest ls))
    (if (null? rest)
        so-far
        (loop (string-append so-far (car rest) ":")
              (cdr rest))))
)
\end{verbatim}

This exemplifies the pattern, with the null string as \texttt{BASE-RESULT} and an \texttt{UPDATER} that appends each new string (and a colon) to the \texttt{so-far} parameter.
The **fold-left** procedure abstracts out the updater procedure and the null-case result. We could, therefore, define **concatenator-with-colons** thus:

```scheme
(define (concatenator-with-colons ls)
  (fold-left (lambda (so-far str)
               (string-append so-far str ":"))
             ""
             ls))
```

Exercise 6: Using **fold-left**, define a workalike for the **reverse** procedure from the (rnrs base) library: a procedure called **backwards** that takes any list as argument and returns a list containing the same elements, arranged in the reverse order.

Generating finite sequences

In studying finite sequences, we’ll sometimes want to work from an expression for an arbitrary element of a sequence in order to generate the whole thing. For instance, we might want to generate a sequence containing the first hundred powers of two, working with the expression (**expt** 2 \(k\)) — for any natural number \(k\), this expression computes the value that should be in (zero-based) position \(k\) of the sequence.

Exercise 7: Define and test a procedure **first-hundred-powers-of-two** that takes no arguments, but simply constructs and returns a list of the first hundred powers of 2. The list should begin with 1 and end with \(2^{99}\). The procedure definition should compute the element in position \(k\) of the list by evaluating the expression (**expt** 2 \(k\)).

Exercise 8: Revise your solution to exercise 7 so that it takes the length of the list as a parameter. The new procedure, **powers-of-two**, should take any natural number as argument and return a list of the length specified by that argument, in which each element is a power of 2 (beginning with 1).

Exercise 9: Revise your solution to exercise 8 so that it takes as parameters both a procedure for the generating expression and the length of the desired finite sequence. The new procedure, **generate-sequence**, should apply the procedure to each natural number from 0 up to, but not including, the value of the first argument and return a list of the results. (For instance, the call (**generate-sequence** (**lambda** (**k** (**expt** 2 \(k\))) 100) should be a list of the first hundred powers of 2.)
Exercise 10: Use your \texttt{generate-sequence} procedure to generate a list of the cubes of the first twenty natural numbers.

Exercise 11: Define and test a procedure named \texttt{iota} that takes one argument, a natural number, and returns a list of the natural numbers less than that argument, in ascending order.