Due Date: Monday, Sep 26, 2011

Fun with Lists

This homework provides some practice in Scheme programming. You can download MIT Scheme from http://www.gnu.org/software/mit-scheme. The Racket environment, which can be downloaded from http://racket-lang.org is another, friendlier, scheme implementation. Either system can be installed on most operating systems (eg,. Linux, Unix, Windows).

Turning in your work

Turnin: Use the class turnin page: http://www.eecs.wsu.edu/~hauser/cs355/turnin. All work should be typed into a .txt, .pdf, or .doc document. If you can make a PDF file it would be appreciated.

General rules

Unless directed otherwise, you should implement your functions using recursive definitions built up from the basic built-in functions such as CONS, CAR, CDR, LIST, etc.

Don't use set! and don't `define` anything except functions.

Practice

For practice, but not to turn in, be sure to do the scheme calisthentics from the course resources page.

nth

Define a function that returns the nth element of a list (0-based indexing). Assume that the index, n, is at least 0 and smaller than the length of the list.

```
>  (nth '(1 2 3 4) 1)
2
```

repl

Define a function repl

```
(define (repl l i v) ... )
```

that returns a (new) list which is the same as l except that the ith element is v. Again, assume that the index i is at least 0 and smaller than the length of the list.

```
>  (repl '(1 2 3 4) 1 7)
(1 7 3 4)
```

range

Define a LISP function range like the range function in python:

```
(define (range min max) ... )
```

that return a list of integers (min min+1 ... max-1). If min ≥ max return the empty list.

```
>  (range 4 6)
(4 5)
```

filter

This higher-order function takes as its first argument one-argument function, called a predicate, which returns #t or #f, and as its second argument a list. It returns a list of all elements in the second argument that satisfy that predicate. The elements must appear in the result in the same order that they appear in the original list.

```
> (filter (lambda (x) (= x 1)) '(1 2))
(1)
> (filter (lambda (x) (< x 3)) '(1 2))
(1 2)
```

merge2

Define a function merge2 that merges two lists of integers, each already in ascending order, into a new list that is also in ascending order. The length of the new list is the sum of the lengths of the original lists.

```
(define (merge2 l1 l2) ...)
```

For example

```
> (merge2 '(2 4 6) '(1 4 5))
(1 2 4 4 5 6)
```

mergeN

Using merge2 and the reduce function defined in class you can now define mergeN which takes a list of lists, each already in ascending order, and returns a new list containing all of the elements in ascending order. For example,

```
> (mergeN '())
()
> (mergeN '((2 4 6) (1 4 5)))
(1 2 4 4 5 6)
> (mergeN '((2 4 5) (1 4 6) (3 7 9)))
(1 2 3 4 4 5 6 7 9)
```

Note that this requires expanding your understanding of reduce to see that it can return a list, not just a simple value; but the solution is, in fact, a very simple use of reduce once you choose the right value for the base case corresponding to the empty list as input.