CSE 1729 - Introduction to Principles of Programming — October 17, 2017

Laboratory Assignment 7

Objectives

- Work with lists

Activities

1. There are a number of equality functions in Scheme: `=, eq?, eqv?, equal?` for example, but none of them "do it all" that is, work for symbols, numbers, and lists. For example, none of these will say `#t` when comparing '(a b c (1 2) 3.0) and '(a b c (1 2) 3), although `(= 3 3.0)` is true, and `(equal? '(a b c (1 2) 3.0) '(a b c (1 2) 3.0)` is true.

   Write a Scheme function `(better-equal? a b)` that works for symbols, numbers, and lists using the following definition:

$$\texttt{(better-equal? a b)} = \begin{cases} \#t & \text{if } a \text{ and } b \text{ are symbols and } \texttt{(eq? a b)}, \\ \#t & \text{if } a \text{ and } b \text{ are numbers and } \texttt{(= a b)}, \\ \#t & \text{if } a \text{ and } b \text{ are lists whose corresponding members are } \texttt{better-equal?}, \\ \#f & \text{otherwise} \end{cases}$$

2. Define a Scheme function `(remove-all x l)` that takes a number or symbol and a list, and returns a list that is the same as `l` with all elements equal to `x` removed. This is different from the `remove` presented in lecture that removed one element, but like the lecture version it should not try to remove `x` from any nested elements. For examples

```
>(remove-all 'a '(a b a c))
(b c)
>(remove-all 'z '(a b a c))
(a b a c)
>(remove-all 'a '(a (a b c) b a ((a))) c)
((a b c) b ((a))) c)
```

3. The **dot product** of two lists of numbers $(x1\ x2\ x3)$ and $(y1\ y2\ y3)$ is

$$x1 * y1 + x2 * y2 + x3 * y3$$

   (a) Define a Scheme function `dot-prod` that takes two lists of numbers as its inputs and returns the dot product of those two lists. Do not use the built-in `map` function in this function. You can assume the two lists have the same length.

(b) Define a Scheme function `dot-prod-with-map` that computes the dot product (as defined above), but uses the built-in `map` function. You may find a use for a function you defined in Lab 6.

4. For Question 3, use an unordered list to represent sets (as in Section 2.3.3 in the textbook). Use the following functions from the book as a starting point:

```
(define (element-of-set? x set)
  (cond ((null? set) #f)
        ((equal? x (car set)) #t)
        (else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))

(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

Define a Scheme function `union-set` which takes a two sets and returns the union of those two sets (which is a set).

5. Define a Scheme function `has-duplicates?` which takes a list and returns `#f` if no two of its members are equal, `#t` otherwise. *Hint:* Use the `member` function.

6. Define a Scheme function `num-zeroes` which takes a (possibly nested) list, and return the number of zeros in the list. You should also include any zeroes in nested sublists in the count. The list may include items of any type, and should count both 0 and 0.0 as zero.

7. Define a Scheme function `(nested-reverse lst)` which reverses a possibly nested list as follows:

   - the elements of the list are in reverse order in the result, and
   - any nested list is also reversed using `(nested-reverse lst)`.

   For example:

```
> (nested-reverse '(a b c))
(c b a)
> (nested-reverse '((a b c) 42 (do re mi (1 2 3))))
(((3 2 1) mi re do) 42 (c b a))
```