Worksheet number 7: Michael Collins (material from Michael Leventon)

Assume the following definitions:

\[
\begin{align*}
(\text{define } (\text{map } \text{proc } \text{seq}) & = \\
& \begin{cases} 
'() & (\text{if } (\text{null? seq}) \\
(\text{cons } (\text{proc } (\text{car seq})) & \text{(map proc (cdr seq)))) & \end{cases} \\
(\text{define } (\text{filter } \text{pred } \text{seq}) & = \\
& \begin{cases} 
'() & (\text{if } (\text{null? seq}) \\
(\text{let } ((\text{rest } (\text{filter pred (cdr seq)}))) & \text{(if } (\text{pred } (\text{car seq})) \\
& \text{(cons } (\text{car seq}) \text{ rest)))} & \end{cases} \\
(\text{define } (\text{fold-right } \text{op } \text{init } \text{seq}) & = \\
& \begin{cases} 
\text{init} & (\text{if } (\text{null? seq}) \\
(\text{op } (\text{car seq}) & \text{(fold-right op init (cdr seq)))) & \end{cases} \\
\end{align*}
\]

(1) Write the following procedures using map, filter, and fold-right (no recursion!).

<table>
<thead>
<tr>
<th>Description</th>
<th>Example</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>square-list: squares all the elements in a list</td>
<td>((\text{square-list (list 1 2 3)}) \Rightarrow (1 4 9))</td>
<td>((\text{define } (\text{square-list } x) \Rightarrow \text{map square } x))</td>
</tr>
<tr>
<td>length: length of a list</td>
<td>((\text{length (list 1 2 3)}) \Rightarrow 3)</td>
<td>((\text{define } (\text{length } x) \Rightarrow \text{fold-right (lambda (a b) (+ 1 b)) 0 x}))</td>
</tr>
<tr>
<td>mean: mean of a list (assume that you have a function length that calculates the length of a list)</td>
<td>((\text{mean (list 1 2 3 4 5)}) \Rightarrow 3)</td>
<td>((\text{define } (\text{mean } x) \Rightarrow (\text{fold-right } + 0 x) (\text{length x})))</td>
</tr>
<tr>
<td>range: range of a list (max(list) - min (list))</td>
<td>((\text{range (list 1 2 3 4 5)}) \Rightarrow 4)</td>
<td>((\text{define } (\text{range } x) \Rightarrow (- (\text{fold-right max (car x) x}) (\text{fold-right min (car x) x})))</td>
</tr>
<tr>
<td>element?: returns true if elt is an element of x</td>
<td>((\text{element? (list 1 2 3 4) 3}) \Rightarrow #t)</td>
<td>((\text{define } (\text{element? } x \text{ elt}) \Rightarrow (\text{not } (\text{null? (filter (lambda (y) (= elt y)) x))}))</td>
</tr>
<tr>
<td>map: map a function over a list of elements</td>
<td>((\text{map double (list 1 2 3)}) \Rightarrow (2 4 6))</td>
<td>((\text{define } (\text{map proc } x) \Rightarrow \text{fold-right (lambda (a b) (cons proc a b)) () x}))</td>
</tr>
<tr>
<td>apply-all: takes a list of procs and an item, and returns a list of applying each procedure to the item.</td>
<td>((\text{apply-all (list sqrt square fact) 4}) \Rightarrow (2 16 24))</td>
<td>((\text{define } (\text{apply-all } \text{ops } x) \Rightarrow \text{map (lambda (op) (op x)) ops}))</td>
</tr>
</tbody>
</table>

(2) Uniqueness

Say we have a list of numbers, and we want to get back a list where each number in the original list appears exactly once. For example,

\[
\begin{align*}
(\text{define } x \text{ (list 1 2 2 3 1 5 4 3 4)}) \\
(\text{unique } x) \Rightarrow (1 2 3 5 4)
\end{align*}
\]
Let’s write the function `unique` to do this. But first, write the function `remove` that removes all instances of an element from a list. (e.g. `(remove 3 (list 1 3 2 3 4 3)) ==> (1 2 4 ).

```
(define (remove elt seq)
  (filter (lambda (x) (not (= x elt))) seq)
)
```

Now write `unique`, using `remove`.

```
(define (unique seq)
  (if (null? seq)
    '()
    (cons (car seq)
      (unique (remove (car seq) (cdr seq))))))
)
```

(3) Write a function `max-err` that takes a list of lists of numbers and computes the `sqrt` of the maximum of the sum of the squares of the sublists. For example,

```
(define data (list (list -1 2) (list 0 -4 3) (list 1 -1 -2))) ==> ((-1 2) (0 -4 3) (1 -1 -2))
(max-err data) ==> 5 (because \( \sqrt{0^2 + (-4)^2 + 3^2} = 5 > \sqrt{(-1)^2 + 2^2} \) and \( \sqrt{1^2 + (-1)^2 + (-2)^2} \))
```

HINT: think about a sequence of transformations of the list which can be performed using map, fold-right etc.

```
(define (max-err data)
  (sqrt
    (fold-right max 0
    (map
      (lambda (trial)
        (fold-right + 0
          (map square trial)))
    data)))
)
```

(4) All Pairs
Write the function `all-pairs` that takes a list and returns a list of all pairs of the elements in the list. For example, `(all-pairs (list 1 2 3 4)) ==> ((1 2) (1 3) (1 4) (2 3) (2 4) (3 4))
Again, first think about how you’d do this, and then translate into Scheme. Maybe start with trying to just get the pairs ((1 2) (1 3) (1 4)), and then build up from there.

```
(define (all-pairs s)
  (if (null? s)
    '()
    (append (map (lambda (x) (list (car s) x)) (cdr s))
             (all-pairs (cdr s))))
)
```

(5) Deep Reverse
So far, we’ve been working on lists, while we’ve ignored the elements of the list. What does the following return? `(reverse (list 1 (list 2 3) (list 4 5 6)))
Write a function `deep-reverse` that when called on the above tree will reverse all the elements. `(deep-reverse (list 1 (list 2 3) (list 4 5 6))) ==> ((6 5 4) (3 2) 1) Use map if you can (hint: you can use `reverse`)

```
(define (deep-reverse x)
  (if (not (pair? x))
    x
    (map deep-reverse (reverse x))))
)