# CS 61A Challenge Problems: Advanced Scheme

Solutions at https://alextseng.net/teaching/cs61a/

Alex Tseng

## 1 Functions and Lambda

(a) Write `filter`. `filter` takes in a list and another predicate function, and returns a list of only the items that satisfy this predicate function.
```
(filter '(1 2 3 4 5 6 7) (lambda (x) (= (modulo x 3) 0))) ---> (3 6)
```

(b) Write `map`, which takes in a list and a function, and returns a new list with the same elements but with the function applied to them.
```
(map '(1 2 3 4 5 6 7) (lambda (x) (* x x))) ---> (1 4 9 16 25 36 49)
```

(c) Write `accumulate`. `accumulate` is the Scheme version of `reduce` in Python. It takes in a list, a function, and a seed. It condenses (or accumulates) the elements of the list using the function, where the starting point is the seed.
```
(accumulate '(1 2 3 4 5 6 7) (lambda (x y) (+ x y)) 0) ---> 28
(accumulate '(1 2 3 4 5 6 7) (lambda (x y) (* x y)) 1) ---> 5040 ; 7!
```

(d) Write the function `compose`, which takes in two functions `f` and `g` and evaluates to a new function that is the composition `f(g(.))`. Assume `f` and `g` are single-argument functions.
```
((compose (lambda (x) (* x x)) (lambda (x) (+ x 2))) 4) ---> 36
```

(e) Write the function `safe-fn`. `safe-fn` takes in a regular single-argument function and a predicate function, and evaluates to a new function that is a safer version by checking the argument using the predicate before evaluating.

```
((safe-fn sqrt (lambda (x) (and (number?  x) (> x 0)))) 16) ---> 4
((safe-fn sqrt (lambda (x) (and (number?  x) (> x 0)))) "not a number") ---> #f
((safe-fn sqrt (lambda (x) (and (number?  x) (> x 0)))) -1) ---> #f
```

(f) *Challenge* Write a function `replicate` that takes in a list and returns a new list with each element replicated `k` times.

```
(replicate '(1 2 3) 3) ---> (1 1 1 2 2 2 3 3 3)
```

(g) Write a function `remove-k` that removes the `k`th element from a given list.

```
(remove-k '(0 1 2 3 4 5) 4) ---> (0 1 2 3 5)
```

A run-length encoding is a way of decreasing the space required to store certain types of data. The general idea of a run-length encoding is that in a lot of types of data, there are long sequences of consecutive items that are the same (runs). For example, in strings, many characters in a row could be the same. In images, there could be a consecutive sequence of many pixels of the same color (JPEGs use this method). A run-length encoding compresses this data down by storing only 1 copy of an element in a run, and the number of times it appears, instead of many copies of the same element. In these next two problems, we will explore a way of performing run-length encoding and decoding on Scheme lists.

(h) Given a run-length encoding, write a function `decode` that turns an encoded list of elements and their counts into the original list. The encoded list consists of the same elements, but where there is a run of more than 1 of the same element in a row, they are condensed into a pair.

```
(define code '((a . 4) (b . 2) c a (b . 3)))
(decode code) ---> (a a a a b b c a b b b)
```

Hint: There is a very easy way to write this function, using some of the functions you have already written above.

(i) *Challenge* Write the corresponding `encode` function that turns a list of elements into a run-length encoded list.

```
(encode '(a b b b c d d e a)) ---> (a (b . 3) c (d . 2) e a)
(equal? (encode (decode code)) ---> #t)
```

Hint: It might be easier to start by writing a helper functions, where a run of 1 element is still encoded as `(x . 1)` instead of just `x`, and fix it later using a function you have already written above.

## 2 Tail Calls

(a) Here is a definition for a modified summing procedure that sums up the elements of a list:

```
(define (sum lst fn)
  (cond ((null? lst) 0)
        (else (+ (fn (car lst)) (sum (cdr lst) fn)))))
```

Rewrite the function to be tail-recursive.

(b) Write the function `power` that raises `x` to the power of `y` so that it is tail-recursive.
`(power 2 5) ---> 32`

Try running this on your Scheme interpreter. Plug in some large numbers, and compare this tail-recursive function and a non-tail-recursive counterpart. You will find that the tail-recursive version will be faster, whereas the non-tail-recursive version may not even finish if it runs out of memory.