Solutions to CS 61A Challenge Problems: Advanced Scheme

Other worksheets and 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)
```

```
(define (filter lst pred)
  (cond ((null? lst) lst)
        ((pred (car lst)) (cons (car lst) (filter (cdr lst) pred)))
        (else (filter (cdr lst) pred))))
```

In this function, as in many others, there are many choices as to what function you use to create lists. Here, we use `cons`. `append` with appropriate lists would also work.

(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)
```

```
(define (map lst fn)
  (cond ((null? lst) lst)
        (else (cons (fn (car lst)) (map (cdr lst) fn)))))
```

Simple recursion!

(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!
```

```
(define (accumulate lst fn seed)
  (cond ((null? lst) seed)
        (else (accumulate (cdr lst) fn (fn seed (car lst))))))
```

Don't worry about what direction we do the accumulating.

(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
```

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

This is a higher-order function in Scheme. We simply return a lambda function. Do not be fooled by the presence of the lambda. Lambda functions are simply nameless functions in Scheme. You will see later on that the `define` function is very closely related to `lambda`.

(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
```

```
(define (safe-fn fn pred)
  (lambda (x)
    (cond ((pred x) (fn x))
          (else #f))))
```

Another example of a higher-order function. This time, it's a little more complicated, but the idea is the same. If you get stuck, try writing it first in Python and translating it into Scheme.

(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)
```

```
(define (repl-one item k)
  (cond ((= k 0) '())
        (else (cons item (repl-one item (- k 1))))))

(define (replicate lst k)
  (cond ((null? lst) lst)
        (else (append (repl-one (car lst) k) (replicate (cdr lst) k)))))
```

It would be extremely difficult to write this without a helper function. We start by writing `repl-one` that simply takes one element and returns a list `k` copies of it. Then our `replicate` function becomes much easier. We simply go through the list and replicate each item individually and append them all together. Note that we must use `append` here, because we want to concatenate sublists together.

(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)
```

```
(define (remove-k lst k)
  (cond ((= k 0) (cdr lst))
        (else (cons (car lst) (remove-k (cdr lst) (- k 1))))))
```

While there are other ways to solve this problem, we use a clever trick to make it much more elegant. Intuitively, we start of with `k` at the front of the list, and traverse down the list. Each time we go the next pair, we decrement `k`. When we get down to `k` being 0, then we skip the current element and tack on the rest of the list. Notice that if we start off with `k` being 0, we simply return the rest of the list without the first element, as expected.

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.

```
(define (decode lst)
  (cond ((null? lst) lst)
        ((pair? (car lst)) (append (replicate (list (car (car lst))) (cdr (car lst)))
                                   (decode (cdr lst))))
        (else (cons (car lst) (decode (cdr lst))))))
```

Despite the long lines of code, this function is very simple because of all the utility functions we have already written. When we see a pair in the list, the first element is some item, and the second element is the number of times it appears in that run. `replicate` is perfect for the job. We replicate that item the appropriate number of times and append it to the result of decoding the rest of the list. If we see a regular element, then it just appears by itself, and we simply use `cons` to attach that element to the result of decoding the rest of the list.

(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.

```
(define (encode-helper lst item run)
  (cond ((null? lst) (list (cons item run)))
        ((equal? item (car lst)) (encode-helper (cdr lst) item (+ run 1)))
        (else (cons (cons item run) (encode-helper (cdr lst) (car lst) 1)))))

(define (encode lst)
  (map (encode-helper (cdr lst) (car lst) 1)
       (lambda (pair) (cond ((= (cdr pair) 1) (car pair))
                            (else pair)))))
```

This is a difficult problem. Almost all of the work being done is by `encode-helper`. Notice that `encode-helper` has arguments that store the current item we are looking at, and how many we have seen already. The basic logic is that every time we see another element in our list that matches `item`, we just add 1 to `run`. If the first element of the list doesn't match, then we have just ended a run. Then we create a new pair with that information and recurse with a new `item` and reset `run` to 1. Our base case is if we have nothing left in our list, then we return the last pair of the element and a run in a list by itself. Our `encode` function calls this helper function by starting everything off correctly. It sets `item` to the first element of the list, `run` to 1, and feeds it the rest of the list not including the first element (we've already seen it and counted it). Notice that for the first example above, this would give us the list, `((a . 1) (b . 3) (c . 1) (d . 2) (e . 1) (a . 1))`. This is almost correct. The last

step is to use `map` and wherever we see a pair with a run-length 1, we simply replace it with the element itself. The lambda function fed into `map` is where this decision is made.

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.

```
(define (sum-tail-helper lst fn result)
  (cond ((null? lst) result)
        (else (sum-tail-helper (cdr lst) fn (+ result (fn (car lst)))))))

(define (sum-tail lst fn)
  (sum-tail-helper lst fn 0))
```

It is a common pattern to see tail-recursive functions being written with a helper function and a wrapper function. Our limitation in the original function is that the `+` operator is outside of the recursive call, and must hang around until the recursion completely backtracks. To avoid this, we use an argument in the helper to store the result so far, and this carries over to all subsequent recursive calls.

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

```
(define (power-helper x y result)
  (cond ((= y 0) result)
        (else (power-helper x (- y 1) (* result x)))))

(define (power x y)
  (power-helper x y 1))
```

This is fairly similar to `sum-tail` above. We use another argument in a helper function to store the result so far. You might also try writing a tail-recursive version of `power` that uses fast exponentiation. Recall, this cuts the time of exponentiation from $\Theta(y)$ to $\Theta(\log y)$. With tail-recursion, this is made even faster in Scheme.

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.