Section 6 - Racket, Mutation, Thunks, Streams, Memoization

This handout was composed by Porter Jones. There are probably plenty of typos/incorrect solutions/etc for you to catch! Please email me with any issues, comments, or feedback at pbjones@cs.washington.edu. All thoughts are welcome :)

Practice with Scope and Mutation

1) Write a function `greater-by-one-list` that takes in a list of numbers and returns a new list that has replaced all of the numbers from the original list with the number one greater than them.

2) Write a function `increase-by-one` that takes in a mutable list of numbers (that is, one made using `mcons`) and replaces all of the numbers from the original list with the number one greater than them.

3) Write a function `silly-previous` which takes an int and returns the previous int with which the function was called (initially 0). For instance, the call `(silly-previous 42)` would return 0, but a subsequent call to `(silly-previous 13)` would return 42.

4) Write a function `silly-only-unique` which takes in an int and returns that int if it has not been passed to `silly-only-unique` before. If it has been passed to `silly-only-unique`, the function should terminate with `(error "silly-only-unique: value used previously")`. Racket has a list function `member` which may be helpful. Member "locates the first element of lst that is equal? to v. If such an element exists, the tail of lst starting with that element is returned. Otherwise, the result is #f."

Thunks and Streams

1) Define a function `powers-of-two` that returns a stream of the powers of two, that is 1, 2, 4, 8, etc.

2) Define a function `zero-through-three` that returns a stream which cycles through the values 0, 1, 2, 3 every time it's called, starting with 0 (Racket has a function modulo that may be useful).

3) Define a function `zero-through-n` that takes a number n and returns a stream which cycles through the values 0, 1, 2, …, n every time it's called, starting with 0. You may assume n is non-negative.

4) Define a function `get-ith` which takes a stream and a number i and returns the ith value of the stream (the first value of the stream is considered the 0th value). Assume the given number is non-negative.

5) Define a function `stream-maker` which takes a function and an initial value and creates a stream that starts at initial value and whose next value is determined by a call to the given function on the previous value.

6) Define a function `powers-of-two-2` that returns the same stream as problem 1 but uses `stream-maker`.

Memoization for Efficiency

1) What is the efficiency of the following implementation of `fibonacci`?

```
(define (fibonacci x)
  (if (or (= x 1) (= x 2))
      1
      (+ (fibonacci (- x 1))
         (fibonacci (- x 2)))))
```

2) Below is another implementation of `fibonacci` that uses memoization to “remember” previous results. What is the efficiency of the implementation below?

```
(define memo-fibonacci
  (letrec([memo null]
          [f (lambda (x)
               (let ([ans (assoc x memo)])
                 (if ans
                     (cdr ans) ; return memoized answer
                     (let ([new-ans (if (or (= x 1) (= x 2))
                                        1
                                        (+ (f (- x 1))
                                           (f (- x 2))))])
                       (begin
                         (set! memo (cons (cons x new-ans) memo))
                         new-ans)))))])
    f))
```

Section 6 - Solutions

This handout was composed by Porter Jones. There are probably plenty of typos/incorrect solutions/etc for you to catch! Please email me with any issues, comments, or feedback at pbjones@cs.washington.edu. All thoughts are welcome :)

Practice with Scope and Mutation

```
1) (define (greater-by-one-list xs)
     (map (lambda (x) (+ 1 x)) xs))

   (define (greater-by-one-list xs)
     (if (null? xs)
         xs
         (cons (+ 1 (car xs))
               (greater-by-one-list (cdr xs)))))


2) (define (increase-by-one xs)
     (cond [(null? xs) null]
           [#t (begin (increase-by-one (mcdr xs))
                      (set-mcar! xs (+ (mcar xs) 1)))]))


3) (define silly-previous
     (let ([prev 0])
       (lambda (x) (let ([res prev])
                     (begin (set! prev x) res)))))


4) (define silly-only-unique
     (let ([prev null])
       (lambda (x)
         (if (member x prev)
             (error "silly-only-unique: value used previously")
             (begin (set! prev (cons x prev)) x)))))
```

Thunks and Streams

1)
```
(define powers-of-two
  (letrec ([next-thunk (lambda (x)
                         (cons x (lambda () (next-thunk (* x 2)))))])
    (lambda () (next-thunk 1))))
```

2)
```
(define zero-through-three
  (letrec ([next-thunk (lambda (x)
                         (cons (modulo x 4)
                               (lambda () (next-thunk (+ x 1)))))])
    (lambda () (next-thunk 0))))
```

3)
```
(define (zero-through-n n)
  (letrec ([next-thunk (lambda (x)
                         (cons (modulo x (+ n 1))
                               (lambda () (next-thunk (+ x 1)))))])
    (lambda () (next-thunk 0))))
```

4)
```
(define (get-ith s i)
  (if (= i 0)
      (car (s))
      (get-ith (cdr (s)) (- i 1))))
```

5)
```
(define (stream-maker init fn)
  (letrec ([next-thunk (lambda (x)
                 (cons x (lambda () (next-thunk (fn x)))))])
    (lambda () (next-thunk init))))
```

6)
```
(define powers-of-two2 (stream-maker 1 (lambda (x) (* x 2))))
```

Memoization for Efficiency

1) Will run on the order of 2^n, since for each number 2 -> n there needs to be two fibonacci calls computed. It might help to draw out a tree of the function calls to understand this.

2) Will run on the order of n. Since adding the memoization stores the most recent calculated values at the front of the list, a call to fibonacci will only have to look at the first two values in the previously remembered results. Again, a tree of function calls may help understand this.