Practica 1 - Principios de Racket y Recursión

Profesora: Karla Ramírez Pulido Ayudante: Héctor Enrique Gómez Morales

Fecha de inicio: 19 de agosto de 2015
Fecha de entrega: 2 de septiembre de 2015

1. Instrucciones

En esta práctica se tienen once ejercicios.

Esta práctica debe ser implementada con la variante plai, es decir su archivo con terminación *.rkt* debe tener como primer linea lo siguiente: `#lang plai`

Todos los ejercicios requieren contar con pruebas mediante el uso de la función `test`:

```
(test <result-expr> <expected-expr>)
```

En donde *result-expr* es una expresión que se evalúa y su valor obtenido es comparado con valor obtenido de la expresión *expected-expr* que es el valor esperado de la prueba. Si las dos expresiones evalúan a lo mismo la prueba imprime el éxito de la prueba, en caso contrario indicar un error.

```
> (test (+ 1 2) 3)
(good (+ 1 2) 3 3 "at line 34")

> (test (+ 1 2) 4)
(bad (+ 1 2) 3 4 "at line 36")
```

Cada ejercicio debe contar al menos con **cinco** pruebas.

2. Ejercicios

Sección I. Define las funciones que se te piden. Puedes crear y utilizar funciones auxiliares. No puedes utilizar funciones de Racket que resuelvan **directamente** los ejercicios.

- **pow** - Define la función pow tal que toma dos números enteros positivos z y w y regresa el numero que se obtiene de elevar el numero z a la potencia w, *i.e.*

  ```
  > (pow 2000 0)
  1
  > (pow  2 3)
  8
  > (pow 8 6)
  262144
  ```

- **average** - Dado una lista no vacía de números, regresar el promedio de esta, *i.e.*

```
> (average '(5))
5
> (average '(3 2 6 2 1 7 2 1))
3
> (average '(10 7 13))
10
```

- **primes** - Dado un numero entero positivo regresar una lista los números primos contenidos entre 2 y el numero entero dado, *i.e.*

```
> (primes 30)
'(2 3 5 7 11 13 17 19 23 29)
> (primes 11)
'(2 3 5 7 11)
> (primes 1)
'()
```

- **zip** - Dadas dos listas, regresar una lista cuyos elementos son listas de tamaños dos, tal que par la i-ésima lista, el primer elemento es el i-ésimo de la primera lista original y el segundo elemento es el i-ésimo de la segunda lista original, si una lista es de menor tamaño que la otra, la lista resultante es del tamaño de la menor, y si una de las listas es vacía, regresar una lista vacía, *i.e.*

```
> (zip '(1 2) '(3 4))
'((1 3) (2 4))
> (zip '(1 2 3) '())
'()
> (zip '() '(4 5 6))
'()
> (zip '(8 9) '(3 2 1 4))
'((8 3) (9 2))
> (zip '(8 9 1 2) '(3 4))
'((8 3) (9 4))
```

- **reduce** - Dada una función de aridad 2 y una lista de n elementos, regresar la evaluación de la función encadenada de todos los elementos, *i.e.*

```
> (reduce + '(1 2 3 4 5 6 7 8 9 10))
55
> (reduce zip '((1 2 3) (4 5 6) (7 8 9)))
'((1 (4 7)) (2 (5 8)) (3 (6 9)))
```

Sección II. Define las funciones que se te piden. Puedes crear y utilizar funciones auxiliares. Para el manejo de listas sólo puedes utilizar funciones básicas de Racket como son `car, cdr, cons, list, empty, empty?, if, let, let*`

- **mconcat** - Dadas dos listas, regresa la concatenación de la primera con la segunda, *i.e.*

```
> (mconcat '(1 2 3) '(4 5 6))
'(1 2 3 4 5 6)
> (mconcat '() '(4 5 6))
'(4 5 6)
> (mconcat '(1 2 3) '())
'(1 2 3)
```

- **mmap** - Dada una función de aridad 1 y una lista, regresar una lista con la aplicación de la función a cada uno de los elementos de la lista original, *i.e.*

```
> (mmap add1 '(1 2 3 4))
'(2 3 4 5)
> (mmap car '((1 2 3) (4 5 6) (7 8 9)))
'(1 4 7)
> (mmap cdr '((1 2 3) (4 5 6) (7 8 9)))
'((2 3) (5 6) (8 9))
```

- **mfilter** - Dado un predicado de un argumento y una lista, regresar la lista original sin los elementos que al aplicar el predicado, regresa falso (un predicado es una función que regresa un valor booleano), *i.e.*

```
> (mfilter (lambda (x) (not (zero? x))) '(2 0 1 4 0))
'(2 1 4)
> (mfilter (lambda (l) (not (empty? l))) '((1 4 2) () (2 4) ()))
'((1 4 2) (2 4))
> (mfilter (lambda (n) (= (modulo n 2) 0)) '(1 2 3 4 5 6))
'(2 4 6)
```

- **any?** - Dado un predicado de un argumento y una lista se debe regresar $\#t$ cuando por lo menos un elemento de la lista regresa $\#t$ con el predicado dado. En caso contrario regresa $\#f$, *i.e.*

```
> (any? number? '())
#f
> (any? number? '(a b c d 1))
#t
> (any? symbol? '(1 2 3 4))
#f
```

- **every?** - Dado un predicado de un argumento y una lista se debe regresar solamente $\#t$ cuando cada uno de los elementos de la lista regresa $\#t$ para el predicado dado. En caso contrario regresa $\#f$, *i.e.*

```
> (every? number? '())
#t
> (every? number? '(1 2 3))
#t
> (every? number? '(1 2 3 a))
#f
> (every? symbol? '(1 2 3 a))
#f
```

- **mpowerset** - Dada una lista, regresar otra como el conjunto potencia de la original (un conjunto potencia de S es un conjunto de subconjuntos de S), *i.e.*

```
> (mpowerset '())
'(())
> (mpowerset '(1))
'(() (1))
> (mpowerset '(1 2))
'(() (1) (2) (1 2))
> (mpowerset '(1 2 3))
'(() (1) (2) (3) (1 2) (1 3) (2 3) (1 2 3))
```