Practica 2 - Tipos de Datos Parte 1 de 2

Profesora: Karla Ramírez Pulido Ayudante: Héctor Enrique Gómez Morales

Fecha de inicio: 26 de agosto de 2015
Fecha de entrega: 9 de septiembre de 2015

1. Instrucciones

En esta práctica se tienen dieciocho 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`. Pueden utilizar sólo la paquetería básica de `racket/base`, las funciones que se implementaron en la practica 1 y funciones auxiliares que ustedes definan.

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. Tipos de datos recursivos y no recursivos Define los siguientes tipos de datos que se te piden.

- **Array** - Definir un tipo de dato `Array` que tenga un constructor de tipo `MArray`. El entero sirve para definir el tamaño del arreglo., *i.e.*

  ```
  > (MArray 4 '(1 2 3))
  (Marray 4 '(1 2 3))
  ```

- **List** - Definir un tipo de dato recursivo llamado `MList` que tenga a la lista vacía `MEmpty` y el constructor de tipo `MCons`, *i.e.*

  ```
  > (MEmpty)
  (MEmpty)
  > (MCons 1 (MCons 2 (MCons 3 (MEmpty))))
  (MCons 1 (MCons 2 (MCons 3 (MEmpty))))
  > (MCons 7 (MCons 4 (MCons 10 (MEmpty))))
  (MCons 7 (MCons 4 (MCons 10 (MEmpty))))
  ```

- **NTree** - Definir un tipo de dato recursivo llamado `NTree` que tenga como una hoja nula `TLEmpty` y un constructor de tipo `NodeN` (estos árboles son n-ários), *i.e.*

```
> (TLEmpty)
(TLEmpty)
> (NodeN 1 (list (TLEmpty) (TLEmpty) (TLEmpty)))
(NodeN 1 (list (TLEmpty) (TLEmpty) (TLEmpty)))
> (NodeN 1 (list (NodeN 2 (list (TLEmpty)))
                 (NodeN 3 (list (TLEmpty)))
                 (NodeN 4 (list (TLEmpty) (TLEmpty) (TLEmpty)))))
(NodeN
 1
 (list
  (NodeN 2 (list (TLEmpty)))
  (NodeN 3 (list (TLEmpty)))
  (NodeN 4 (list (TLEmpty) (TLEmpty) (TLEmpty)))))
```

- **Position** - Define un tipo de dato llamado `Position` que tenga un constructor de tipo `2D-Point` que toma dos números reales que indican una posición en el plano cartesiano, *i.e.*

```
> (2D-Point 0 0)
(2D-Point 0 0)
> (2D-Point 1 (sqrt 2))
(2D-Point 1 1.4142135623730951)
```

- **Figure** - Define un tipo de dato llamado `Figure` que tenga tres constructores:
  - `Circle` - Un constructor que toma un centro dada por un `Position` y un radio.
  - `Square` - Un constructor que toma una posición de la esquina superior izquierda del cuadrado y una longitud.
  - `Rectangle` - Un constructor que toma una posición de la esquina superior izquierda del rectángulo, su ancho y su largo.

```
> (Circle (2D-Point 2 2) 2)
(Circle (2D-Point 2 2) 2)
> (Square (2D-Point 0 3) 3)
(Square (2D-Point 0 3) 3)
> (Rectangle (2D-Point 0 2) 2 3)
(Rectangle (2D-Point 0 2) 2 3)
```

Sección II. Funciones sobre Datos Define las siguientes funciones que se te piden. Puedes crear y utilizar funciones auxiliares. No puedes utilizar funciones de Racket que resuelvan **directamente** los ejercicios.

- **setvalueA** - Dado un arreglo de tipo `Array`, una posición y un valor numérico v, regresar otro arreglo con el valor v intercambiado en la posición indicada del arreglo original. En caso de que la posición sea igual o mayor al tamaño especificado en el arreglo, regresar un error `Out of bounds`, *i.e.*

```
> (define ar (MArray 5 '(0 0 0 0 0)))
> ar
(MArray 5 '(0 0 0 0 0))
> (setvalueA ar 2 4)
(MArray 5 '(0 0 4 0 0))
```

```
> (setvalueA ar 4 4)
(MArray 5 '(0 0 0 0 4))
> (setvalueA ar 5 4)
. . setvalueA: Out of bounds
```

- **MArray2MList** - Dado un arreglo de tipo `MArray`, regresar una lista de tipo `MList` que contenga todos los elementos del arreglo original, *i.e.*

```
> (MArray2MList (MArray 0 '()))
(MEmpty)
> (MArray2MList (MArray 5 '("a" "b")))
(MCons "a" (MCons "b" (MEmpty)))
> (MArray2MList (MArray 3 '(1 2 3)))
(MCons 1 (MCons 2 (MCons 3 (MEmpty))))
```

- **printML** - Dada una lista de tipo `MList`, imprimirla en un formato legible. Puedes utilizar las funciones para manipular cadenas y la función $\sim a$, *i.e.*

```
> (printML (MEmpty))
"[]"
> (printML (MCons 7 (MEmpty)))
"[7]"
> (printML (MCons 7 (MCons 4 (MEmpty))))
"[7, 4]"
> (printML (MCons (MCons 1 (MCons 2 (MEmpty))) (MCons 3 (MEmpty))))
"[[1, 2], 3]"
```

- **concatML** - Dadas dos listas de tipo `MList`, regresar la concatenación, *i.e.*

```
> (concatML (MCons 7 (MCons 4 (MEmpty))) (MCons 1 (MEmpty)))
(MCons 7 (MCons 4 (MCons 1 (MEmpty))))
> (concatML (MCons 7 (MCons 4 (MEmpty))) (MCons 1 (MCons 10 (MEmpty))))
(MCons 7 (MCons 4 (MCons 1 (MCons 10 (MEmpty)))))
```

- **lengthML** - Dada una lista de tipo MLista, regresar la cantidad de elementos que tiene, *i.e.*

```
> (lengthML (MEmpty))
0
> (lengthML (MCons 7 (MCons 4 (MEmpty))))
2
```

- **mapML** - Dada una lista de tipo MLista y una funcion de aridad 1, regresar una lista de tipo MLista con la aplicación de la funcion a cada uno de los elementos de la lista original, *i.e.*

```
> (mapML add1 (MCons 7 (MCons 4 (MEmpty))))
(MCons 8 (MCons 5 (MEmpty)))
> (mapML (lambda (x) (* x x)) (MCons 10 (MCons 3 (MEmpty))))
(MCons 100 (MCons 9 (MEmpty)))
```

- **filterML** - Dada una lista de tipo MLista y un predicado de un argumento, regresar una lista de tipo MLista sin los elementos que al aplicar el predicado, regresa falso, *i.e.*

```
> (filterML (lambda (x) (not (zero? x))) (MCons 2 (MCons 0 (MCons 1 (MEmpty)))))
(MCons 2 (MCons 1 (MEmpty)))
> (filterML (lambda (l) (not (MEmpty? l)))
            (MCons (MCons 1 (MCons 4 (MEmpty))) (MCons (MEmpty) (MCons 1 (MEmpty)))))
(MCons (MCons 1 (MCons 4 (MEmpty))) (MCons 1 (MEmpty)))
```

  Definamos los siguientes tipos de datos y valores:

```
(define-type Coordinates
  [GPS (lat number?)
       (long number?)])

(define-type Location
  [building (name string?)
            (loc GPS?)])

;; Coordenadas GPS
(define gps-satelite (GPS 19.510482 -99.23411900000002))
(define gps-ciencias (GPS 19.3239411016 -99.179806709))
(define gps-zocalo (GPS 19.432721893261117 -99.13332939147949))
(define gps-perisur (GPS 19.304135 -99.19001000000003))

(define plaza-satelite (building "Plaza Satelite" gps-satelite))
(define ciencias (building "Facultad de Ciencias" gps-ciencias))
(define zocalo (building "Zocalo" gps-zocalo))
(define plaza-perisur (building "Plaza Perisur" gps-perisur))

(define plazas (MCons plaza-satelite (MCons plaza-perisur (MEmpty))))
```

- **haversine** - Dados dos valores de tipo `GPS` calcular su distancia usando la formula de `haversine`, *i.e.*

```
> (haversine gps-ciencias gps-zocalo)
13.033219276117368
> (haversine gps-ciencias gps-perisur)
2.44727738966455
> (haversine gps-satelite gps-perisur)
23.391736010506026
```

- **gps-coordinates** - Dada una lista de edificios, regresar la lista de coordenadas GPS de los edificios, las listas deben ser construidas con el tipo de dato `MList`, *i.e.*

```
> (gps-coordinates (MEmpty))
(MEmpty)
> (gps-coordinates plazas)
(MCons
 (GPS 19.510482 -99.23411900000002)
 (MCons (GPS 19.304135 -99.19001000000003) (MEmpty)))
```

- **closest-building** - Dado b un valor de tipo `building` y una lista de tipo `MList` de buildings, regresar el edificio mas cercano a b, *i.e.*

```
> (closest-building zocalo plazas)
(building "Plaza Satelite" (GPS 19.510482 -99.23411900000002))
> (closest-building ciencias plazas)
(building "Plaza Perisur" (GPS 19.304135 -99.19001000000003))
```

- **buildings-at-distance** - Dado b un valor de tipo `building`, una lista de tipo `MList` de buildings y una distancia d, regresar una nueva lista de todos los edificios que están a distancia menor o igual a d del edificio b, *i.e.*

```
> (buildings-at-distance ciencias plazas 10)
(MCons (building "Plaza Perisur" (GPS 19.304135 -99.19001000000003)) (MEmpty))
> (buildings-at-distance ciencias plazas 20)
(MCons (building "Plaza Perisur" (GPS 19.304135 -99.19001000000003)) (MEmpty))
> (buildings-at-distance ciencias plazas 25)
(MCons
 (building "Plaza Satelite" (GPS 19.510482 -99.23411900000002))
 (MCons (building "Plaza Perisur" (GPS 19.304135 -99.19001000000003)) (MEmpty)))
```

- **area** - Dada una figura del tipo `Figure`, regresar el área de la figura, *i.e.*

```
> (area (Circle (2D-Point 5 5) 4))
50.26548245743669
> (area (Square (2D-Point 0 0) 20))
400
> (area (Rectangle (2D-Point 3 4) 5 10))
50
```

- **in-figure?** - Dada una figura `fig` del tipo `Figure` y una posición `p` del tipo `2D-Point` regresa #t si `p` esta dentro de `fig` y #f en caso contrario, *i.e.*

```
> (in-figure? (Square (2D-Point 5 5) 4) (2D-Point 6 6))
#t
> (in-figure? (Rectangle (2D-Point 5 5) 4 6) (2D-Point 4 4))
#f
```