Practica 3 - Tipos de Datos Parte 2 de 2

Profesora: Karla Ramírez Pulido Ayudante: Héctor Enrique Gómez Morales

Fecha de inicio: 2 de septiembre de 2015
Fecha de entrega: 17 de septiembre de 2015

1. Instrucciones

En esta práctica se tienen trece 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` 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. Heart Rate Zones Para las siguientes funciones, se utilizaran los tipos `HRZ` ,`Coordinate` y `Frame` definidos en `practica3-base.rkt`:

```
(define-type HRZ
  [resting (low number?)
           (high number?)]
  [warm-up (low number?)
           (high number?)]
  [fat-burning (low number?)
               (high number?)]
  [aerobic (low number?)
           (high number?)]
  [anaerobic (low number?)
             (high number?)]
  [maximum (low number?)
           (high number?)])
```

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

(define-type Frame
  [trackpoint (loc GPS?)
              (hr exact-integer?)
              (zone HRZ?)
              (unix-time exact-integer?)])
```

El tipo de dato `HRZ` representa las zonas de frecuencia cardiaca, cada zona tiene como parámetros su mínimo y máximo ritmo cardiaco correspondiente.

El tipo de dato `Coordinate` representa una posición, se tiene el constructor de tipo `GPS` que representa una posicion GPS que consta de una latitud y una longitud.

El tipo de dato `Frame` representa un instante o marco, se tiene el constructor del tipo `trackpoint` que toma una coordenada GPS, un ritmo cardiaco, una zona de frecuencia cardiaca y una `timestamp` en formato `UNIX time`.

- **zones** - Dado el ritmo cardiaco de descanso y el máximo ritmo cardiaco de una persona se debe regresar la lista de zonas de frecuencia cardiaca. La formula utilizada comúnmente es:
  $range = max - rest$
  $zone\ 0\ (resting) = rest\ (Minimo)$
  $zone\ 0\ (resting) = (range * 0{,}5) - 1\ (Maximo)$

  $zone\ (i+1) = rest + range * (0{,}5 + (0{,}1 * i))\ (Minimo)$
  $zone\ (i+1) = rest + range * (0{,}5 + (0{,}1 * (i+1))) - 1\ (Maximo)$,
  donde i = 0.,4, *i.e.*

  ```
  > (zones 50 180)
  (list
   (resting 50 114.0)
   (warm-up 115.0 127.0)
   (fat-burning 128.0 140.0)
   (aerobic 141.0 153.0)
   (anaerobic 154.0 166.0)
   (maximum 167.0 180))
  ```

  Para los siguientes ejemplos se define:

  ```
  > (define my-zones (zones 50 180))
  ```

- **get-zone** - Dado un símbolo que es el nombre de una zona y una lista de zonas regresar el tipo de dato correspondiente, *i.e.*

  ```
  > (get-zone 'anaerobic my-zones)
  (anaerobic 154.0 166.0)
  > (get-zone 'maximum my-zones)
  (maximum 167.0 180)
  ```

- **bpm->zone** - Dado una lista de frecuencias cardiacas y una lista de zonas de frecuencia cardiaca regresar una lista de zonas por cada frecuencia cardiaca en la lista original, *i.e.*

```
> (bpm->zone empty my-zones)
'()
> (bpm->zone '(50 60) my-zones)
(list (resting 50 114.0) (resting 50 114.0))
> (bpm->zone '(140 141) my-zones)
(list (fat-burning 128.0 140.0) (aerobic 141.0 153.0))
```

- **create-trackpoints** - Dado una lista en la que cada elemento de la lista contiene: un tiempo en formato `UNIX`, una lista con la latitud y longitud y finalmente el ritmo cardiaco. Como segundo parámetro se tiene una lista de zonas cardiacas con lo que se tiene que regresar una lista de `trackpoint`s que contengan la información dada. Se tiene definida la variable `raw-data` con datos reales para sus pruebas, *i.e.*

```
> (take raw-data 4))
'((1425619654 (19.4907258 -99.24101) 104)
  (1425619655 (19.4907258 -99.24101) 104)
  (1425619658 (19.4907258 -99.24101) 108)
  (1425619662 (19.4907107 -99.2410833) 106))
> (create-trackpoints (take raw-data 4) my-zones)
(list
 (trackpoint (GPS 19.4907258 -99.24101) 104 (resting 50 114.0) 1425619654)
 (trackpoint (GPS 19.4907258 -99.24101) 104 (resting 50 114.0) 1425619655)
 (trackpoint (GPS 19.4907258 -99.24101) 108 (resting 50 114.0) 1425619658)
 (trackpoint (GPS 19.4907107 -99.2410833) 106 (resting 50 114.0) 1425619662))
```

- **total-distance** - Dada una lista de `trackpoint`s, regresar la distancia total recorrida, *i.e.*

```
> (define sample (create-trackpoints (take raw-data 100) my-zones))
> (total-distance (create-trackpoints sample my-zones))
0.9509291243812747
> (define trackpoints (create-trackpoints raw-data my-zones))
> (total-distance trackpoints)
5.051934549322941
```

- **average-hr** - Dada una lista de `trackpoint`s, regresar el promedio del ritmo cardiaco, el resultado debe ser un entero *i.e.*

```
> (average-hr sample)
134
> (average-hr trackpoints)
150
```

- **max-hr** - Dada una lista de `trackpoint`s, regresar el máximo ritmo cardiaco, el resultado debe ser un entero *i.e.*

```
> (max-hr sample)
147
> (max-hr trackpoints)
165
```

- **collapse-trackpoints** - Dada una lista de `trackpoints` y un epsilon `e`, obtener una nueva lista en que se agrupen los deltas consecutivos dado que se cumpla lo siguiente: la distancia de un trackpoint al otro trackpoint debe ser menor o igual a `e` y los trackpoints deben tener el mismo ritmo cardiaco, *i.e.*

```
> (define sample-four (create-trackpoints (take raw-data 4) my-zones))
> sample-four
(list
 (trackpoint (GPS 19.4907258 -99.24101) 104 (resting 50 114.0) 1425619654)
 (trackpoint (GPS 19.4907258 -99.24101) 104 (resting 50 114.0) 1425619655)
 (trackpoint (GPS 19.4907258 -99.24101) 108 (resting 50 114.0) 1425619658)
 (trackpoint (GPS 19.4907107 -99.2410833) 106 (resting 50 114.0) 1425619662))
> (collapse-trackpoints sample-four 0.01)
(list
 (trackpoint (GPS 19.4907258 -99.24101) 104 (resting 50 114.0) 1425619655)
 (trackpoint (GPS 19.4907258 -99.24101) 108 (resting 50 114.0) 1425619658)
 (trackpoint (GPS 19.4907107 -99.2410833) 106 (resting 50 114.0) 1425619662))
```

Sección II. Árboles Binarios Para las siguientes funciones, utiliza el tipo de dato `BTree` definido en `practica3-base.rkt`:

```
(define-type BTree
  [EmptyBT]
  [BNode (c procedure?)
         (l BTree?)
         (e any?)
         (r BTree?)])
```

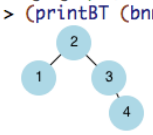
Donde el primer parámetro del constructor de tipo `BNode` es una función de comparación que recibe dos argumentos y regresa un booleano, indicando si el primer argumento es menor que el segundo. Para un árbol de números, la función de comparación sería `<`, para strings sería `string<?`.

En esta práctica utilizaremos abreviaciones para los constructores de tipos:

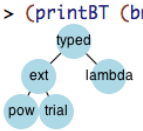
1. `ebt` es lo mismo que `(EmptyBT)`
2. `(bnn ebt 1 ebt)` es lo mismo que `(BNode < (EmptyBT) 1 (EmptyBT))`
3. `(bns ebt ‘‘hello’’ ebt)` es lo mismo que `(BNode string<? (EmptyBT) ‘‘hello’’ (EmptyBT))`

Por último, cuentas con la función `printBT` que recibe un árbol de tipo `BTree` y regresa su representación gráfica, para que puedas depurar tu código:

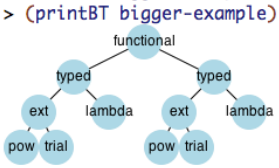
```
Welcome to DrRacket, version 6.1.1 [3m].
Language: plai; memory limit: 128 MB.
> (printBT (bnn (bnn ebt 1 ebt) 2 (bnn ebt 3 (bnn ebt 4 ebt))))
```




```
> (printBT (bns (bns (bns ebt "pow" ebt) "ext" (bns ebt "trial" ebt)) "typed" (bns ebt "lambda" ebt)))
```




```
> (define example1 (bns (bns (bns ebt "pow" ebt) "ext" (bns ebt "trial" ebt)) "typed" (bns ebt "lambda" ebt)))
> (define bigger-example (bns example1 "functional" example1))
> (printBT bigger-example)
```




```
>
```

- `ninBT` - Dado un árbol de tipo `BTree`, determinar el número de nodos internos que tiene.

```
> (ninBT (EmptyBT))
0
> (ninBT (BNode < (BNode < (EmptyBT) 3 (EmptyBT)) 1 (BNode < (EmptyBT) 2 (EmptyBT))))
1
```

- `nlBT` - Dado un árbol de tipo `BTree`, determinar el número de hojas no vacías.

```
> (nlBT (EmptyBT))
0
> (nlBT (BNode < (BNode < (EmptyBT) 3 (EmptyBT)) 1 (BNode < (EmptyBT) 2 (EmptyBT))))
2
```

- `nnBT` - Dado un árbol de tipo `BTree`, determinar el número de nodos que tiene. Las hojas vacías no cuentan.

```
> (nnBT (EmptyBT))
0
> (nnBT (BNode < (BNode < (EmptyBT) 3 (EmptyBT)) 1 (BNode < (EmptyBT) 2 (EmptyBT))))
3
```

- `mapBT` - Dado una función de aridad 1 y un árbol de tipo `BTree`, aplicar la función sobre todos los valores de los nodos del árbol (las funciones de aridad 1 sólo regresas números).

```
> (mapBT add1 (EmptyBT))
(EmptyBT)
> (mapBT add1 (BNode < (EmptyBT) 1 (BNode < (EmptyBT) 2 (EmptyBT))))
(BNode < (EmptyBT) 2 (BNode < (EmptyBT) 3 (EmptyBT)))
> (mapAB (lambda (x) (* x x)) (BNode < (EmptyBT) 3 (BNode < (EmptyBT) 2 (EmptyBT))))
(BNode < (EmptyBT) 9 (BNode < (EmptyBT) 4 (EmptyBT)))
```

Ahora, sea `arbol-base`...

```
(define arbol-base (bns (bns (bns ebt "A" ebt) "B" (bns (bns ebt "C" ebt) "D" (bns ebt "E" ebt)))
                        "F"
                        (bns ebt "G" (bns (bns ebt "H" ebt) "I" ebt))))
```

1. **preorderBT** - Dado un árbol de tipo `BTree`, regresar una lista de sus elementos recorridos en preorden.

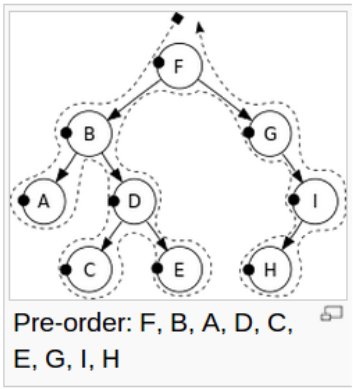
   ```
   > (preorderBT arbol-base)
   '("F" "B" "A" "D" "C" "E" "G" "I" "H")
   ```

2. **inorderBT** - Dado un árbol de tipo `BTree`, regresar una lista de sus elementos recorridos en inorden.

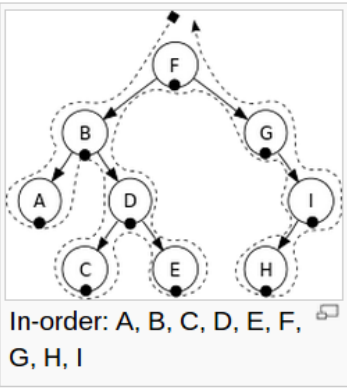
   ```
   > (inorderBT arbol-base)
   '("A" "B" "C" "D" "E" "F" "G" "H" "I")
   ```

3. **posorderBT** - Dado un árbol de tipo `BTree`, regresar una lista de sus elementos recorridos en post-orden.

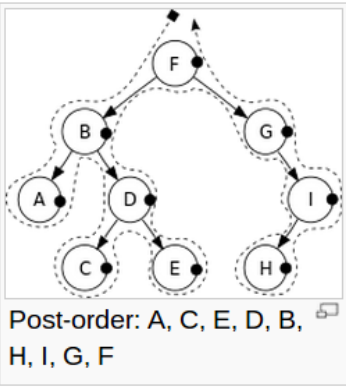
   ```
   > (posorderBT arbol-base)
   '("A" "C" "E" "D" "B" "H" "I" "G" "F")
   ```



Pre-order: F, B, A, D, C, E, G, I, H



In-order: A, B, C, D, E, F, G, H, I



Post-order: A, C, E, D, B, H, I, G, F