UBA – Facultad de Ciencias Exactas y Naturales – Departamento de Computación– Algoritmos y Estructura de Datos I
Recuperatorio Extra de Parcial de Programación Funcional– 12/08/2009

1		2			3	**TOTAL**
a	b	a	b	c		

LU:
Nro Orden:
Apellidos:
Nombres:

Aclaraciones: El parcial NO es a libro abierto. Cualquier decisión de interpretación que se tome debe ser aclarada y justificada. Para aprobar se requieren al menos 60 puntos. Entregar cada ejercicio en hoja separada.

En el comienzo del parcial trabajaremos con los tipos de datos abstractos `Colectivo` y `Parada`, que son definidos de la siguiente manera:

```
tipo Colectivo {
  observador capacidad (c : Colectivo) : Int ;
  observador linea (c : Colectivo) : String ;
  observador recorrido (c : Colectivo) : [Parada] ;
  invariante capacidad(c) >= 0 ;
  invariante (∀p ← recorrido(c))linea(c) ∈ lineas(p) ;
}

tipo Parada {
  observador direccion (p : Parada) : String ;
  observador lineas (p : Parada) : [String] ;
  observador pasajerosEsperando (p : Parada) : Int ;
}
```

Las funciones que implementan estos tipos en Haskell son `capacidad:: Colectivo -> Int`, `linea :: Colectivo -> String`, `recorrido:: Colectivo -> [Parada]`,`direccion :: Parada -> String`, `lineas:: Parada -> [String]` , `pasajerosEsperando:: Parada -> Int`

Ejercicio 1. [35 puntos] Implementar en Haskell los siguientes problemas especificados más adelante

a) **[15 p.]** `problema seLaBancan` (cs : [Colectivo]) = `result` : [Colectivo]
b) **[20 p.]** `problema direccionesMasVisitadas` (cs:[Colectivo]) = `result` : [String]

```
problema seLaBancan (cs : [Colectivo]) = result : [Colectivo] {
  asegura (∀c ← result)c ∈ cs ;
  asegura (∀c ← cs, capacidad(c) >= genteEsperado(c))c ∈ result ;
}

problema direccionesMasVisitadas (cs : [Colectivo]) = result : [String] {
  requiere |cs| > 0 ;
  asegura mismos(result, lasMasVisitadas(direcciones(cs))) ;
}

aux genteEsperado (c: Colectivo) : Int = acum(total + pasajerosEsperando(p) | total : Int = 0, p ← recorrido(c)) ;
aux lasMasVisitadas (ss: [String]) : [String] = [ s | s ← sinRepe(ss), apariciones(s, ss) == máx([ apariciones(s, ss) | s ← ss ]) ] ;
aux direcciones (cs: [Colectivo]) : [String] = acum(total + +[direccion(p)] | total : [String] = [], c ← cs, p ← recorrido(c)) ;
```

Tipos algebraicos.

Se cuenta con el tipo compuesto SimilYenga que modela un tablero de SimilYenga. Se busca implementar en Haskell algunos problemas sobre este tipo compuesto mediante su tipo algebraico que se define con los siguientes constructores:

```
data SimilYenga = Nuevo | Izq Int SimilYenga | Der Int SimilYenga
```

que permiten crear un SimilYenga nuevo (vacío), agregar una pieza a la izquierda (con cierto peso) o a la derecha (también con cierto peso).

Ejercicio 2. [45 puntos]

a) **[10 p.]**(Ejercicio 8 de la práctica 4) Implementar `estaBalanceado :: SimilYenga -> Bool`, que devuelve verdadero sólo si se cumplen estas 2 condiciones a la vez:
- la cantidad de piezas a izquierda es igual a la cantidad de piezas a derecha,
- la suma de los pesos a izquierda y derecha coincide.

Por ejemplo, aplicar `estaBalanceado` al SimilYenga `Izq 1 (Der 5 Nuevo)` debería devolver Falso ya que la suma de los pesos a cada lado difiere

b) **[15 p.]**(Ejercicio 8 de la práctica 4) Implementar `compactar :: SimilYenga -> SimilYenga`, que toma un SimilYenga y lo compacta de la siguiente manera:
- Cada vez que haya dos piezas a izquiera o derecha consecutivas, las debe reemplazar por una sola, y el peso de ésta será la suma de los pesos de las piezas reemplazadas,
Ejemplo, dado el siguiente SimilYenga: `Der 3(Izq 1 (Izq 5 Nuevo))`
debería devolver `Der 3(Izq 6 Nuevo)`.

c) **[20 p.]** Implementar `zoom :: SimilYenga -> SimilYenga`, que toma un SimilYenga y lo aumenta de la siguiente manera:
- A cada una de sus piezas la debe duplicar tantas veces como su peso original lo indique, manteniendo en cada una de las nuevas piezas el peso original.
Ejemplo, dado el siguiente SimilYenga: `Der 3(Izq 2 Nuevo)`
debería devolver `Der 3(Der 3(Der 3(Izq 2(Izq 2 Nuevo))))`.

Ejercicio 3. [20 puntos] Supongamos la siguiente función en Haskell:
f(0) = 0
f(x) = f(x+1) - x - 1
¿Qué valor tiene la expresión f(f(-3))? Justificar la respuesta usando reducciones.