UBA – Facultad de Ciencias Exactas y Naturales – Departamento de Computación– Algoritmos y Estructura de Datos I
Recuperatorio de Programación Funcional– 07/07/2010

LU:
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. No está permitido utilizar alto orden. Al igual que para el TP, para la resolución del parcial, se pueden usar únicamente las funciones y operadores `last`, `init`, `head`, `tail`, `!!`, `reverse`, `++`, `elem`, `length` y los operadores de comparación entre elementos de un mismo tipo.

En el comienzo del parcial trabajaremos con los tipos de datos abstractos `Menu` y `Combo`, definidos de la siguiente manera:

`tipo` Nombre = String ;
`tipo` Producto = String ;
`tipo` Precio = $\mathbb{R}$;

```
tipo Menu {
  observador productos (m: Menu) : [Producto] ;
  observador importe (m: Menu, p: Producto) : Precio ;
      requiere p ∈ productos(m) ;
  observador ofertas (m: Menu) : [Combo] ;
  invariante distintos(productos(m)) ;
  invariante (∀p ← productos(m))importe(m, p) ≥ 0 ;
  invariante (∀c ← ofertas(m), p ← productos(c))
    p ∈ productos(m) ;
  invariante distintos([nombre(c)|c ← ofertas(m)]) ;
  invariante (∀c ← ofertas(m))masConveniente(m, c) ;
}
```

```
tipo Combo {
  observador nombre (c: Combo) : Nombre ;
  observador productos (c: Combo) : [Producto] ;
  observador importe (c: Combo) : Precio ;
  invariante distintos(productos(c)) ;
  invariante importe(c) ≥ 0 ;
}
```

`aux masConveniente` (m:Menu,c:Combo) : Bool = $importe(c) \leq \sum[importe(m, p)|p \leftarrow productos(c)]$;
`aux distintos` (l:[T]) : Bool = $(\forall i, j \leftarrow [0..|l|), i \neq j)l_i \neq l_j$;

Las funciones que implementan estos tipos en Haskell son `nombreC :: Combo -> Nombre`, `productosC :: Combo -> [Producto]`, `importeC :: Combo -> Precio` , `productosM :: Menu -> [Producto]`, `importeM :: Menu -> Producto -> Precio` y `ofertasM :: Menu -> [Combo]`

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

a) **[15 p.]** `problema combOne` (m: Menu) = `result` : [(Producto,Nombre)]
b) **[20 p.]** `problema comboMas` (m: Menu) = `result` : Combo

```
problema combOne (m: Menu) = result : [(Producto,Nombre)] {
  asegura (∀p ← productos(m), estaEnUnSoloCombo(m, p))(p, comboQueLoTiene(m, p)) ∈ result ;
  asegura (∀x ← result)estaEnUnSoloCombo(m, prm(x)) ∧ sgd(x) == comboQueLoTiene(m, p) ;
  asegura distintos([prm(result)|p ← result]) ;
  aux estaEnUnSoloCombo (m:Menu,p:Producto) : Bool = |combosQueLoContienen(m, p)| == 1 ;
  aux comboQueLoTiene (m:Menu,p:Producto) : Nombre = cab(combosQueLoContienen(m, p)) ;
  aux combosQueLoContienen (m:Menu,p:Producto) : [Nombre] = [ nombre(c) | c ← ofertas(m), p ∈ productos(c) ] ;
}
```

```
problema comboMas (m: Menu) = result : Combo {
  requiere |ofertas(m)| > 0 ;
  asegura result ∈ [ c | c ← ofertas(m), esElMasConveniente(m, c) ] ;
  aux esElMasConveniente (m:Menu,c:Combo) : Bool = (∀d ← ofertas(m), c ≠ d)ahorro(m, c) ≥ ahorro(m, d) ∨ (ahorro(m, c) ==
    ahorro(m, d) ∧ |productos(c)| ≥ |productos(d)|) ;
  aux ahorro (m:Menu,c:Combo) : ℝ = ∑[ importe(m, p) | p ← productos(c) ] − importe(c) ;
}
```

Tipos algebraicos.

Nota Importante: No está permitido utilizar el tipo lista para resolver los ejercicios de tipos algebraicos (Ejercicios 2 y 3).

Se cuenta con el tipo compuesto Calesita que modela el tan reconocido medio de diversión consistente en una plataforma rotatoria con asientos para los pasajeros.

`tipo` Nombre = String ;

```
tipo Calesita {
  observador rondas (c:Calesita) : Int ;
  observador pasajeros (c:Calesita, i:ℤ) : [Nombre] ;
      requiere 0 ≤ i < rondas(c) ;
  observador alguienSacoSortija (c:Calesita, i:ℤ) : Bool ;
      requiere 0 ≤ i < rondas(c) ;
  observador elQueSacoLaSortija (c:Calesita, i:ℤ) : Nombre ;
      requiere 0 ≤ i < rondas(c) ;
      requiere alguienSacoSortija(c, i) ;
  invariante cantPositiva : rondas(c) ≥ 0 ;
  invariante noGiraVacia : (∀r ← [0..rondas(c))) |pasajeros(c, r)| > 0 ;
  invariante soloPasajerosSacanSortijas : (∀r ← [0..rondas(c)), alguienSacoSortija(c, r))elQueSacoLaSortija(c, r) ∈ pasajeros(c, r) ;
}
```

donde el observador *rondas* devuelve la cantidad de rondas que se realizaron en la calesita. Luego, se puede saber quiénes fueron los pasajeros en esa ronda, si alguien sacó la sortija y en caso afirmativo, de quién se trató.

Se busca implementar en Haskell algunos problemas sobre el tipo compuesto Calesita mediante el tipo algebraico de mismo nombre. Dicho tipo está definido con los siguientes constructores:

```
type Nombre = String
data Calesita = Nueva | Sube Nombre Calesita | Gira Calesita | GiraS Nombre Calesita
```

donde el primer constructor permite construir una calesita nueva, el segundo permite que se suba un niño (pasando su nombre como parámetro), el tercero hace que la calesita se ponga en movimiento, el cuarto constructor se utiliza en reemplazo del tercero en el caso que algún niño haya obtenido la sortija (el nombre del mismo se pasa como parámetro). Una posible calesita podría ser la siguiente: `Gira (Sube Flor (Sube Gabi (GiraS Gabi (Sube Maty (Sube Flor (Sube Gabi Nueva))))))`.

donde en la primer vuelta subieron Gabi, Flor y Maty. Gabi sacó la sortija, así que se subió a la siguiente ronda. Flor no se quizo quedar atrás y se subió también. Luego la calesita volvió a girar, pero ni Flor ni Gabi sacaron la sortija en esta oportunidad.

Ejercicio 2. [20 puntos] Implementar `bienFormada:: Calesita -> Bool`, que dada una Calesita devuelve verdadero si y solo si la calesita está bien formada. Una calesita está bien formada si:

- Nunca gira vacía, es decir que siempre subió alguien antes de que la calesita empezara a girar.
- En cada ronda que se saca la sortija, la persona que la saca estaba subida a la calesita en esa ronda.

Ejemplos:

- `GiraS "Flor" (Sube "Flor" (Sube "Gabi" Nueva))` en este caso está bien formada porque la única vez que la calesita gira no está vacía, y además Flor subió antes de sacar la sortija.
- `GiraS "Flor" (Sube "Mati" (Sube "Gabi" Nueva))` en este caso no lo está porque Flor saca la sortija pero en esa ronda no se había subido a la calestita.
- `Gira (Gira (Sube "Gabi" Nueva))` en este caso no lo está porque la segunda ronda la calesita gira sin que se haya subido nadie.

Ejercicio 3. [30 puntos] Implementar `sortilegio:: Calesita -> Nombre`, que dada una Calesita devuelve el nombre de la persona que sacó más veces la sortija

```
problema sortilegio (c : Calesita) = result : Nombre {
  requiere (∃r ← [0..rondas(c)))alguienSacoSortija(c, r) ;
  asegura result ∈ todosLosPasajeros(c) ;
  asegura (∀p ← todosLosPasajeros(c))cantSortijas(c, result) ≥ cantSortijas(c, p) ;
  aux todosLosPasajeros (c:Calesita) : [Nombre] = sacarRepetidos([ p | r ← [0..rondas(c)), p ← pasajeros(c, r) ]) ;
  aux cantSortijas (c:Calesita, n:Nombre) : ℤ = ∑ [ 1 | r ← [0..rondas(c)), alguienSacoSortija(c, r) ∧ elQueSacoLaSortija(c, r) == n ] ;
  aux sacarRepetidos (ls:[T]) : [T] = [ ls_i | i ← [0..|ls|), ls_i ∉ ls(i..|ls|) ] ;
}
```

Ejercicio 4. [15 puntos] (Ejercicio 2.8 tomado de la práctica 7)
Implementar `aplanarConNBlancos :: [[Char]] -> Integer -> [Char]`, que a partir de una lista de palabras, arma una lista de caracteres concatenándolas e insertando n blancos entre cada par (n debe ser no negativo).