UBA – Facultad de Ciencias Exactas y Naturales – Departamento de Computación– Algoritmos y Estructura de Datos I
Parcial de Programación Funcional– 27/10/2008

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.

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

```
tipo Peon {
  observador nombre (p : Peon) : String;
  observador sueldo (p : Peon) : Float;
  invariante sueldo(p) ≥ 0;
}

tipo Vaca {
  observador nombre (v : Vaca) : String;
  observador litros (v : Vaca) : Float;
  observador costo (v : Vaca) : Float;
  observador aCargo (v : Vaca) : String;
  invariante litros(v) ≥ 0 ∧ costo(v) ≥ 0;
}
```

```
tipo Tambo {
  observador razonSocial (t : Tambo) : String;
  observador peones (t : Tambo) : [Peon];
  observador vacas (t : Tambo) : [Vaca];
  invariante distintos(peones(t));
  invariante distintos(vacas(t));
  invariante (∀v ← vacas(t))aCargo(v) ∈ listaNombres(peones(t));
  aux distintos (l:[T]) : Bool = (∀i, j ← [0..|l|), i ≠ j)nombre(l_i) ≠ nombre(l_j);
  aux listaNombres (l:[T]) : Bool = [nombre(x)|x ← l];
}
```

Las funciones que implementan estos tipos en Haskell son `nombreP :: Peon -> String`, `sueldoP :: Peon -> Float`, `nombreV :: Vaca -> String`, `litrosV :: Vaca -> Float`, `costoV :: Vaca -> Float`, `aCargoV :: Vaca -> String`. `razonSocialT :: Tambo -> String`, `peonesT :: Tambo -> [Peon]` y `vacasT :: Tambo -> [Vaca]`

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

a) **[15 p.]** `problema inspeccion` (ts : [Tambo]) = `result` : [(String,Float)]
b) **[20 p.]** `problema empleadoDelMes` (t : Tambo) = `result` : [Peon]

```
problema inspeccion (ts : [Tambo]) = result : [(String,Float)] {
  asegura mismos(losNombres(result), nombresSinRepe(todosLosPeones(ts)));
  asegura (∀r ← result)sgd(r) == sumaSueldo(prm(r), ts);
  aux sumaSueldo (s:String,ts:[Tambo]) : Float = ∑[ sueldo(p) | t ← ts, p ← peones(t), nombre(p) == s ];
  aux losNombres (rs:[(String,Float)]) : [String] = [ prm(r) | r ← rs ];
  aux nombresSinRepe (ps:[Peon]) : [String] = sinRepe([ nombre(p) | p ← ps ]);
  aux todosLosPeones (ts:[Tambo]) : [Peon] = [ p | t ← ts, p ← peones(t) ];
  aux sinRepe (ls:[T]) : [T] = [ ls_i | i ← [0..|ls|), ls_i ∉ ls_(i..|ls|) ];
}
```

```
problema empleadoDelMes (t : Tambo) = result : [Peon] {
  asegura mismos(result, [ p | p ← peones(t), (∀x ← peones(t))produce(p,t) >= produce(x,t) ]);
  aux produce (p:Peon, t:Tambo) : Float = ∑[ litros(v) | v ← vacas(t), aCargo(v) == nombre(p) ];
}
```

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 Chocotorta que modela una deliciosa chocotorta. Básicamente, sobre una bandeja vacía, se pueden ir agregando capas de chocolinas y relleno (de dulce de leche o queso crema).

Se busca implementar en Haskell algunos problemas sobre el tipo compuesto Chocotorta mediante el tipo algebraico `Chocotorta`. Dicho tipo está definido con los siguientes constructores

```
data Chocotorta = Bandeja | Chocolinas Chocotorta | DulceDeLeche Chocotorta | QuesoCrema Chocotorta
```

que permiten crear una Chocotorta "Vacía" (Bandeja), agregar una capa de chocolinas, una capa de relleno de dulce de leche o una capa de queso crema respectivamente. Por ejemplo, una chocotorta que - comenzando desde la bandeja - tiene una capa de chocolinas, una capa de dulce de leche, una de chocolinas, una de queso crema y una de chocolinas se contruye de la siguiente manera:
`Chocolinas (QuesoCrema (Chocolinas (DulceDeLeche (Chocolinas Bandeja))))`.

Ejercicio 2. [**20 puntos**] Implementar `estaBienFormada :: Chocotorta -> Bool`, que devuelve verdadero solamente si: a) la primer capa de la chocotorta (la que se encuentra pegada a la bandeja) es de chocolinas y b) a cada capa de chocolinas le sigue una capa de relleno y viceversa (a cada capa de relleno le sigue una capa de chocolinas). No hay restricción para la última capa (puede ser tanto bandeja, como chocolinas o relleno).

Ejemplo:
`Bandeja` está bien formada.
`Chocolinas (QuesoCrema (Chocolinas (DulceDeLeche (Chocolinas Bandeja))))` está bien formada.
`Chocolinas (QuesoCrema (DulceDeLeche (Chocolinas Bandeja)))` NO está bien formada dado que hay dos capas de relleno consecutivas (dulce de leche y queso crema).
`Chocolinas (DulceDeLeche Bandeja)` NO está bien formada dado que la primer capa de la chocotorta (la que se encuentra pegada a la bandeja) es de relleno (Dulce de Leche) y NO de chocolinas.

Ejercicio 3. [**25 puntos**] Implementar `meRoboLaReceta :: Chocotorta -> Chocotorta -> Bool`, que devuelve solamente verdadero si la primer chocotorta está contenida en la segunda (dejando de lado la Bandeja). Es decir, si existe en la segunda chocotorta una secuencia de chocolinas y relleno igual a la primera.

Ejemplo: Si tenemos:

ChocotortaA =`Chocolinas (QuesoCrema (Chocolinas Bandeja))`
ChocotortaB = `Chocolinas (QuesoCrema (Chocolinas (DulceDeLeche (Chocolinas Bandeja))))`.
ChocotortaC = `Chocolinas (DulceDeLeche (Chocolinas (DulceDeLeche (Chocolinas Bandeja))))`
ChocotortaD = `Chocolinas (QuesoCrema (Chocolinas (DulceDeLeche (Chocolinas (DulceDeLeche (Chocolinas Bandeja))))))`
ChocotortaE = `QuesoCrema (Chocolinas (DulceDeLeche (Chocolinas Bandeja)))`

`meRoboLaReceta ChocotortaA ChocotortaB` es verdadero.
`meRoboLaReceta ChocotortaA ChocotortaC` es falso.
`meRoboLaReceta ChocotortaA ChocotortaA` es verdadero.
`meRoboLaReceta ChocotortaE ChocotortaD` es verdadero.

Ejercicio 4. [**20 puntos**] (Ejercicio tomado de la práctica 7) *sumaAcumulada* :: [*Int*] → [*Int*], que devuelve una lista, que posee en la posición i-ésima, la suma acumulada de 0 a i en la lista original. Por ejemplo *sumaAcumulada* $[1, 2, 3, 4, 5]$ es $[1, 3, 6, 10, 15]$.