UBA – Facultad de Ciencias Exactas y Naturales – Departamento de Computación– Algoritmos y Estructura de Datos I
Recuperatorio Parcial de Programación Funcional– 08/07/2013

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 `fst`, `snd`, `last`, `init`, `head`, `tail`, `!!`, `reverse`, `++`, `elem`, `length` y los operadores de comparación entre elementos de un mismo tipo.

En esta ocasión, nos centraremos en una banda musical de reconocida barra brava de fútbol. Esta banda está compuesta por músicos de sobrada experiencia ... no precisamente musical. La banda sólo sabe/puede interpretar una sola pieza musical. Cada músico tiene su propia partitura (anotada en una servilleta de choripan usada). Las partituras se encuentran compuestas por una lista de duplas. Cada dupla representa la nota que se debe ejecutar y en qué momento. Todas las notas tienen la misma duración, duran sólo ese momento.

De esta manera, obtenemos la siguiente representación:

```
tipo Nombre = String;
tipo Momento = ℤ;
tipo NotaMusical = Do, Re, Mi, Fa, Sol, La, Si;
tipo Partitura = [(Momento, NotaMusical)];
tipo Musico {
  observador apodo (m : Musico) : Nombre;
  observador servilleta (m : Musico) : Partitura;
  invariante mtoPositivo : (∀p ← servilleta(m))prm(p) ≥ 0;
  invariante mtoOrdenadoYSinRepe(servilleta(m));
}
```

```
tipo Banda {
  observador integrantes (b : Banda) : [Musico];
  invariante hayEquipo : |integrantes(b)| > 0;
  invariante musicosNoRepetidos : sinRepetidos(apodos(b));
  invariante alguienToca :
    |momentosDeLosIntegrantes(b)| > 0;
  invariante noMeCallo : (∀m1 ← rangoMomentos(b))
    (∃m2 ← momentosDeLosIntegrantes(b))m1 == m2;
}
```

```
aux minimoMomento (b:Banda) : Momento =
  [m1|m1 ← momentosDeLosIntegrantes(b), (∀m2 ← momentosDeLosIntegrantes(b))m1 ≤ m2)]0;
aux maximoMomento (b:Banda) : Momento =
  [m1|m1 ← momentosDeLosIntegrantes(b), (∀m2 ← momentosDeLosIntegrantes(b))m1 ≥ m2)]0;
aux apodos (b:Banda) : [Nombre] = [apodo(i)|i ← integrantes(b)];
aux rangoMomentos (b:Banda) : [Momento] = [minimoMomento(b)..maximoMomento(b)];
aux sinRepetidos (l : [T]) : Bool = (∀i ← [0..|l|)) li ∉ l(i..|l|);
aux momentosDeLosIntegrantes (b:Banda) : [Momento] = [prm(p)|i ← integrantes(b), p ← servilleta(i)];
aux mtoOrdenadoYSinRepe (xs : Partitura) : Bool = (∀i ← [0..|xs| − 1)) prm(xsi) < prm(xsi+1);
```

Las funciones que implementan estos tipos en Haskell son `apodoM :: Musico -> Nombre`, `servilletaM :: Musico -> [(Momento,NotaMusical)]`, `integrantesB :: Banda -> [Musico]`

Ejercicio 1. [45 puntos] Implementar en Haskell los siguientes problemas especificados más adelante

a) **[20 p.]** `problema apodosDeModa (bs : [Banda]) = result : [Nombre]`
b) **[25 p.]** `problema queNoDecaiga (b : Banda) = result : Momento`

```
problema apodosDeModa (bs : [Banda]) = result : [Nombre] {
  asegura sinRepetidos(result);
  asegura (∀n ∈ result)estaEnTodasLasBandas(n, bs);
  asegura (∀b ← bs, n ← apodos(b), estaEnTodasLasBandas(n, bs))n ∈ result;
}
aux estaEnTodasLasBandas (n:Nombre,bs:[Banda]) : Bool = (∀b ← bs)n ∈ apodos(b);
problema queNoDecaiga (b : Banda) = result : Momento {
  asegura result ∈ rangoMomentos(b);
  asegura (∀i ← rangoMomentos(b))
    |todasLasNotasDelMomentoDistintas(b, result)| ≥ |todasLasNotasDelMomentoDistintas(b, i)|;
}
aux todasLasNotasDelMomentoDistintas (b : Banda, m : Momento) : [NotaMusical] =
  [n|n ← [Do, Re, Mi, Fa, Sol, La, Si], n ∈ todasLasNotasDelMomento(b, m)];
aux todasLasNotasDelMomento (b : Banda, m : Momento) : [NotaMusical] =
  [sgd(p)|i ← integrantes(b), p ← servilleta(i), prm(p) == m];
```

Tipos algebraicos. ⇒ Nota Importante: No está permitido CONVERTIR EL TIPO ALGEBRAICO AL TIPO LISTAS para resolver los ejercicios de tipos algebraicos (Ejercicios 2 y 3).

Se cuenta con el tipo compuesto Torta que modela las tortas de una reconocida abuela.

`tipo` Ingrediente = Bizcochuelo, Chocolate, Crema;

```
tipo Torta {
  observador pisos (t: Torta) : [Ingrediente];
  invariante bienFormada(t);
}
```

`aux bienFormada` (t:Torta) : Bool $= tieneAlgo(t) \wedge superficieChoco(t) \wedge baseBizco(t)$;
`aux tieneAlgo` (t:Torta) : Bool $= |pisos(t)| \geq 3$;
`aux superficieChoco` (t:Torta) : Bool $= pisos(t)_{|pisos(t)|-1} == Chocolate$;
`aux baseBizco` (t:Torta) : Bool $= pisos(t)_0 == Bizcochuelo$;

donde el observador pisos(t) devuelve las lista de ingredientes de esa torta, en el orden en que se encuentran dispuestos los mismos, siendo la primer posición la base y la última la superficie.

Se busca implementar en Haskell algunos problemas sobre el tipo compuesto Torta mediante el tipo algebraico `Torta`. Dicho tipo se define a través de los siguientes constructores:

```
data Torta = Nueva | Bizcochuelo Torta | Chocolate Torta | Crema Torta
```

donde en el tipo Torta, el primer constructor permite construir una torta nueva sin nada de nada y los restantes constructores permiten agregarle en el último piso de la torta el ingrediente correspondiente.
Por ejemplo, una Torta t cuyo $pisos(t) == [Bizcochuelo, Crema, Crema, Chocolate, Crema, Chocolate]$ se construye de la siguiente manera: `Chocolate (Crema (Chocolate (Crema (Crema (Bizcochuelo Nueva)))))`.

Ejercicio 2. [20 puntos] Implementar `esSubTorta:: Torta -> Torta -> Bool`, que a partir de dos tortas, devuelve verdadero si solo si la segunda torta es una subtorta de la primera. Es decir,

Por ejemplo:

```
esSubTorta (Chocolate (Crema (Bizcochuelo (Crema (Bizcochuelo Nueva))))) (Chocolate (Crema (Bizcochuelo Nueva)))
```
debería dar como resultado verdadero

```
esSubTorta (Chocolate (Crema (Bizcochuelo (Crema (Bizcochuelo Nueva))))) (Chocolate (Chocolate (Bizcochuelo Nueva)))
```
debería dar como resultado falso

`problema esSubTorta` (t1: Torta, t2: Torta) = `result` : Bool {
  `requiere` $|pisos(t1)| \geq |pisos(t2)|$;
  `asegura` $result == (\exists i \leftarrow [0..|pisos(t1)| - |pisos(t2)|])pisos(t1)_{[i..i+|pisos(t2)|-1]} == pisos(t2)$;
}

Ejercicio 3. [25 puntos] Implementar `pisoteada:: Torta -> Torta`, que dada una Torta t devuelve una nueva torta semejante a la primera, pero en los pisos que posee un ingrediente repetido varias veces lo devuelve una sola vez.

Por ejemplo:

```
pisoteada (Chocolate (Chocolate (Bizcochuelo (Crema (Crema (Bizcochuelo Nueva))))))
```
debería dar como resultado (Chocolate (Bizcochuelo (Crema (Bizcochuelo Nueva))))

```
pisoteada (Chocolate (Chocolate (Chocolate (Chocolate (Bizcochuelo (Bizcochuelo (Crema (Bizcochuelo Nueva))))))))
```
debería dar como resultado (Chocolate (Bizcochuelo (Crema (Bizcochuelo Nueva))))

`problema pisoteada` (t: Torta) = `result` : Torta {
  `asegura` $result == (pisos(t)_0 : [pisos(t)_i | i \leftarrow [1..|pisos(t)|), pisos(t)_{i-1} \neq pisos(t)_i])$;
}

Ejercicio 4. [10 puntos] (Ejercicio 5.3 tomado de la práctica 7). Dada una lista de elementos de tipo $\mathbb{Z}$ denominada L, definir una función subListasOrdenadas que devuelva:

a) una lista de listas de elementos de tipo $\mathbb{Z}$ tal que cada lista tiene todos sus elementos iguales y longitud igual a la cantidad de apariciones de ese elemento en L, además la lista resultante está "ordenada" ascendentemente. Por ejemplo, $L = [8, 5, 5, 4, 9, 4, 4, 4, 8]$ la función debería devolver [[4,4,4,4],[5,5],[8,8],[9]].