UBA – Facultad de Ciencias Exactas y Naturales – Departamento de Computación– Algoritmos y Estructura de Datos I
Parcial de Programación Funcional– 21/05/2010

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 `Receta` y `Alacena`, definidos de la siguiente manera:

`tipo` Nombre = String ;
`tipo` Producto = String ;
`tipo` Cantidad = $\mathbb{Z}$;

```
tipo Alacena {
  observador productos (a : Alacena) : [Producto] ;
  observador cant (a : Alacena, p: Producto) : Cantidad ;
      requiere p ∈ productos(a) ;
  invariante distintos(productos(a)) ;
  invariante (∀p ← productos(a))cant(a, p) ≥ 0 ;
}
```

```
tipo Receta {
  observador nombre (r : Receta) : Nombre ;
  observador tiempo (r : Receta) : Float ;
  observador ingredientes (r : Receta) : [(Producto, Cantidad)] ;
  invariante tiempo(r) > 0 ;
  invariante distintos([prm(i)|i ← ingredientes(r)]) ;
  invariante (∀i ← ingredientes(i))sgd(i) > 0 ;
}
aux distintos (l:[T]) : Bool = (∀i, j ← [0..|l|), i ≠ j)l_i ≠ l_j ;
```

Las funciones que implementan estos tipos en Haskell son `productosA :: Alacena -> [Producto]`, `cantA :: Alacena -> Producto -> Cantidad`, `nombreR :: Receta -> Nombre`, `tiempoR :: Receta -> Cantidad`, `ingredientesR :: Receta -> [(Producto,Float)]`

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

a) **[15 p.]** `problema cuantoRinde` (a: Alacena, r: Receta) = `result` : Cantidad
b) **[20 p.]** `problema recetasGourmet` (a: Alacena, rs:[Receta]) = `result` : [Receta]

```
problema cuantoRinde (a: Alacena, r:Receta) = result : Cantidad {
  asegura faltaAlguno(a, r) → result == 0 ;
  asegura ¬faltaAlguno(a, r) → alcanzaParaHacerlaNVecesMax(a, r, result) ;
}
```

```
problema recetasGourmet (a: Alacena, rs:[Receta]) = result : [Receta] {
  requiere (∀r ← rs)¬faltaAlguno(a, r) ∧ hayCantidadSuficienteEnLaAlacena(a, r, 1) ;
  asegura mismos(result, recetasMasRapidas(recetasQueMejorRinden(a, rs))) ;
}
```

```
aux alcanzaParaHacerlaNVecesMax (a:Alacena, r:Receta, n:ℤ) : Bool = hayCantidadSuficienteEnLaAlacena(a, r, n)
  ∧ ¬hayCantidadSuficienteEnLaAlacena(a, r, n + 1) ;
aux cotaMaxima (a:Alacena) : ℤ = max([cant(a, p)|p ← productos(a)]) ;
aux faltaAlguno (a:Alacena,r:Receta) : Bool = (∃p ← ingredientesDeReceta(r))p ∉ productos(a) ;
aux hayCantidadSuficienteEnLaAlacena (a:Alacena, r:Receta, n:ℤ) : Bool = (∀i ← ingredientes(r))cant(a, prm(i)) ≥ sgd(i) ∗ n ;
aux ingredientesDeReceta (r:Receta) : Producto = [prm(p)|p ← ingredientes(r)] ;
aux maximoRinde (a:Alacena, r:Receta) : ℤ = [n|n ← [0..cotaMaxima(a)], alcanzaParaHacerlaNVecesMax(a, r, n)]_0 ;
aux recetasMasRapidas (rs:[Receta]) : [Receta] = [r|r ← rs, (∀s ← rs)tiempo(r) ≤ tiempo(s)] ;
aux recetasQueMejorRinden (a:Alacena, rs:[Receta]) : [Receta] = [r|r ← rs, (∀s ← rs)maximoRinde(a, r) ≥ maximoRinde(a, s)] ;
```

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 Juego que modela el tan reconocido Piedra, Papel, o Tijera, juego de manos en el cual los jugadores dicen juntos "... Piedra, papel o tijera" y justo al acabar muestran todos al mismo tiempo una de sus manos, de modo que puede verse el arma que cada uno ha elegido:

- Piedra: un puño cerrado.
- Papel: todos los dedos extendidos, con la palma de la mano mirando hacia abajo.
- Tijera: dedos índice y mayor extendidos y separados formando una "V".

El objetivo es vencer al oponente seleccionando el arma que gana a la que ha elegido él, siguiendo estas reglas:

a) La piedra aplasta o rompe la tijera (gana la piedra)
b) La tijera corta el papel (gana la tijera)
c) El papel envuelve la piedra (gana el papel)
d) Si los jugadores eligen la misma arma es un empate.

Finalmente, contamos con el tipo compuesto Juego, que modela una partida de Piedra, Papel o Tijera entre dos jugadores.
`tipo` Mano = Piedra, Papel, Tijera ;

```
tipo Juego {
  observador cantidadDeJugadas (j: Juego) : Int ;
```

```
  observador iesimaJugada (j: Juego, i:ℤ) : (Mano,Mano) ;
      requiere 0 ≤ i < cantidadDeJugadas(j) ;
  invariante cantPositiva : cantidadDeJugadas(j) ≥ 0 ;
}
```

donde el observador cantidadDeJugadas(j) devuelve la cantidad de jugadas realizadas en ese juego. Luego, se puede ver cada jugada a partir del observador iesimaJugada(j,i). Donde si $i == 0$, iesimaJugada(j,i) devuelve la primer jugada, si $i == 1$, iesimaJugada(j,i) devuelve la segunda jugada, y así siguiendo. La primer componente del par ordenado corresponde a la jugada hecha por el jugador 1. La segunda componente corresponde al jugador 2.

Se busca implementar en Haskell algunos problemas sobre el tipo compuesto `Juego` mediante los tipos algebraicos `Mano` y `Juego`. Dichos tipos están definidos con los siguientes constructores:

```
data Mano = Piedra | Papel | Tijera
data Juego = Nuevo | M Mano Mano Juego
```

donde en el tipo Juego, el primer constructor permite construir un juego nuevo, y el segundo permite realizar una jugada (el primer parámetro Mano corresponde al jugador 1 y el segundo al jugador 2). Por ejemplo, un juego de 3 jugadas podría ser: `M Tijera Tijera (M Papel Piedra (M Piedra Tijera Nuevo))`.

donde la primer jugada es Piedra-Tijera, la segunda Papel-Piedra y la tercera Tijera-Tijera.

Ejercicio 2. [30 puntos] Implementar `subJuego:: Juego -> Int -> Int -> Juego`, que dado un Juego y dos índices, devuelve un juego con las jugadas que se encuentran entre esos índices.

```
problema subJuego (j: Juego, a,b:ℤ) = result : Juego  {
  requiere 0 ≤ a ≤ b < cantidadDeJugadas(j) ;
  asegura cantidadDeJugadas(result) == b − a + 1 ;
  asegura (∀i ← [a..b])iesimaJugada(j, i) == iesimaJugada(result, i − a) ;
}
```

Ejemplo: Si contamos con la siguiente juegada:

```
jA =  M Papel Tijera (M Tijera Tijera (M Papel Piedra (M Piedra Tijera Nuevo)))
```

entonces:

`subJuego jA 0 1` devuelve `M Papel Piedra (M Piedra Tijera Nuevo)`.
`subJuego jA 2 2` devuelve `M Tijera Tijera Nuevo`.

Ejercicio 3. [20 puntos]

Suponga que cuenta con el problema `puntaje` y su implementación.

```
problema puntaje (j : Juego) = result : (ℤ,ℤ)  {
  asegura result == sumaRangoDeJugadas(j, 0, cantidadDeJugadas(j) − 1) ;
}
aux sumaRangoDeJugadas (j:Juego, a,b:ℤ) : ℤ  = acum(sumaJugada(j, i, t) | t : (ℤ, ℤ) = (0, 0), i ← [a..b]) ;
aux sumaJugada (j:Juego, i:ℤ, t:(ℤ,ℤ)) : (ℤ,ℤ)  = (prm(t) + 1 ∗ β(ganaJugador1(j, i)), sgd(p) + 1 ∗ β(ganaJugador2) ;
aux ganaJugador1 (j:Juego,i:ℤ) : Bool  = iesimaJugada(j, i) == (Piedra, Tijera) ∨ iesimaJugada(j, i) == (Papel, Piedra)
  ∨ iesimaJugada(j, i) == (Tijera, Papel) ;
aux ganaJugador2 (j:Juego,i:ℤ) : Bool  = iesimaJugada(j, i) == (Tijera, Piedra) ∨ iesimaJugada(j, i) == (Piedra, Papel)
  ∨ iesimaJugada(j, i) == (Papel, Tijera) ;
```

Ejemplo:
`puntaje Nuevo` debería devolver (0,0)
`puntaje (M Tijera Tijera (M Papel Piedra (M Piedra Tijera Nuevo)))` debería devolver (2,0)
`puntaje (M Papel Tijera (M Papel Papel Nuevo))` debería devolver (0,1)

Implementar `mejorRancha1:: Juego -> Juego`, que dado un Juego devuelve el subjuego en que mejor le fue al jugador 1.

```
problema mejorRancha1 (j: Juego) = result : Juego  {
  asegura esUnSubJuego(j, result) ;
  asegura esLaMejorRachaDe1(j, result) ;
  aux esUnSubJuego (j,r:Juego) : Bool  = cantidadDeJugadas(r) ≤ cantidadDeJugadas(j)
    ∧ (∃a ← [0..cantidadDeJugadas(j) − cantidadDeJugadas(r)](∀i ← [a..a + cantidadDeJugadas(r)))iesimaJugada(j, i) ==
    iesimaJugada(r, i − a) ;
  aux esLaMejorRachaDe1 (j,r:Juego) : Bool  = (∀a, b ← [0..cantidadDeJugadas(j)), a ≤ b)
    racha(j, a, b) ≤ racha(r, 0, cantidadDeJugadas(r) − 1) ;
  aux racha (j:Juego,a,b:ℤ) : ℤ  = prm(sumaRangoDeJugadas(j, a, b)) − snd(sumaRangoDeJugadas(j, a, b)) ;
}
```

Ejercicio 4. [15 puntos] (Ejercicio 4 tomado de la práctica 7) Definir la siguiente función *descomponerEnPrimos* :: $[Int] \to [[Int]]$, que devuelve la lista de listas, que resulta de descomponer en números primos cada uno de los números de la lista original. Por ejemplo *descomponerEnPrimos* $[2, 10, 6, 9]$ es $[[2], [2, 5], [2, 3], [3, 3]]$.