Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 1

Funciones, tipos y clases en Haskell

1. Objetivo

El objetivo de este proyecto es recordar cómo definir funciones en Haskell. En particular, se evaluará la definición de funciones recursivas usando caso base y caso inductivo (a través de análisis por casos, o pattern-matching). Se evaluará también el uso de las funciones del Preludio sobre listas, que se pueden usar para definir muchas funciones polimórficas.

En algunas de las funciones será necesario utilizar definiciones locales y también el uso de guardas para alternativas booleanas. En otros casos deberás utilizar lo que en Haskell se llaman "sections", que corresponde a la aplicación parcial de operadores binarios.

Algunas instrucciones:

- Hacer todo el proyecto en un mismo archivo.
- Introducir cada ejercicio por un comentario (por ejemplo: `-- ejercicio n`)
- Escribir el tipo de todas las funciones.
- Nombrar las distintas versiones de una misma función `f`, `f''`, `f'''`, ...
- Se recomienda hacer primero los ejercicios sin punto estrella.

Además de los ejercicios, se verificará el conocimiento de los comandos vistos en el teórico (como 'cd', 'ls', 'cp', 'mv', 'mkdir'...) y que el ciclo edición-interpretación se haga de manera eficiente sin usar el mouse, es decir solamente usando atajos de teclado.

2. Ejercicios

1. Una vez terminado el ejercicio de los cuantificadores del práctico (en formalismo básico) programar las funciones siguientes en Haskell.

   a) $paratodo$: dada una lista de valores $R : [A]$ y un predicado $T : A \to Bool$, determina si todos los elementos en R hacen verdadero el predicado T, es decir:

   $$\begin{array}{|l} paratodo : [A] \to (A \to Bool) \to Bool \\ \hline paratodo.R.T \doteq \langle\, \forall\, i \;:\; i \in R :\; T.i \,\rangle \end{array}$$

   b) $existe$: dada una lista de valores $R : [A]$ y un predicado $T : A \to Bool$, determina si algún elemento en R hace verdadero el predicado T, es decir:

   $$\begin{array}{|l} existe : [A] \to (A \to Bool) \to Bool \\ \hline existe.R.T \doteq \langle\, \exists\, i \;:\; i \in R :\; T.i \,\rangle \end{array}$$

c) $sumatoria$: dada una lista de valores $R : [A]$ y una función $T : A \to Int$ (toma elementos de A y devuelve enteros), calcula la suma de la aplicación de T a los elementos en R es decir:

$$\begin{array}{|l} sumatoria : [A] \to (A \to Int) \to Int \\ \hline sumatoria.R.T \doteq \langle \sum i \ : i \in R : \ T.i \rangle \end{array}$$

d) $productoria$: dada una lista de valores $R : [A]$ y una función $T : A \to Int$, calcula el producto de la aplicación de T a los elementos de R, es decir:

$$\begin{array}{|l} productoria : [A] \to (A \to Int) \to Int \\ \hline productoria.R.T \doteq \langle \prod i \ : i \in R : \ T.i \rangle \end{array}$$

Punto ⋆ 1 *Al escribir las dos últimas funciones en Haskell, cómo declarar su tipo de forma tal que admitan distintos valores aritméticos en* `Int`, `Integer`, `Double`, *etc?*

Ayuda: *Usar clases.*

2. Utilizando las funciones anteriores escribir las siguientes en Haskell:

   a) $todosPares :: [Int] \to Bool$ devuelve verdadero si los todos los elementos son pares.

   Ayuda: *La función que determina si un Integral es par está en el Preludio de Haskell*

   b) $hayMultiplo :: Int \to [Int] \to Bool$ devuelve verdadero si hay algún número en la lista que es múltiplo del primer parámetro.

   c) $sumaCuadrados :: Int \to Int$ dado un número no negativo n devuelve la suma de los primeros n cuadrados:

   $$\begin{array}{|l} sumaCuadrados : Int \to Int \\ \hline sumaCuadrados.n \doteq \langle \sum i \ : 0 \leq i < n : \ i^2 \rangle \end{array}$$

   Ayuda: *En Haskell se puede escribir la lista que contiene el rango de números entre n y m como* `[n..m]`.

3. Verificar que se cumplan las reglas de rango vacío y unitario en el ejercicio 1. Para ello desplegar la definición de la función escrita en Haskell y/o probar corriendo los casos de prueba.

4. Una vez que termine de escribir la definición recursiva de la función $cuantGen$ que aparece en un ejercicio del práctico, traducirla al lenguaje Haskell incluyendo su tipo.

5. Reescribir en Haskell todas las funciones del punto 2 utilizando $cuantGen$ (sin usar inducción y en una línea por función).

6. Considere las siguientes funciones:

   - `sumarALista :: Num a => a -> [a] -> [a]` que toma un número y una lista de números y le suma a cada elemento de la lista el primer parámetro. Por ejemplo:

     ```
     sumarALista 3 [4,6,7] = [7,9,10]
     ```

   - `encabezar :: a -> [[a]] -> [[a]]` que toma una expresión de tipo a y lo pone en la cabeza de cada lista del segundo parámetro. Por ejemplo:

```
encabezar 3 [[2,1],[],[4,7]] = [[3,2,1],[3],[3,4,7]]
```

a) Escriba `sumarALista` y `encabezar` usando caso base y caso inductivo.

b) Escriba ambas funciones utilizando la función `map` .

7. Programar una función que dada una lista de números devuelve aquellos que son pares.

   a) Programarla con caso base e inductivo.

   b) Programarla utilizando la función `filter`.

8. Considere la función `encuentra` que dado un valor de tipo `Int` y una lista de pares `[(Int,String)]` devuelve el segundo componente del primer par cuyo primer componente es igual al primer parámetro. En el caso que ningún elemento de la lista cumpla con esto, devolver el string vacío. Por ejemplo:

   ```
   encuentra 10 [(40,"tos"),(10,"uno"),(16,"taza"),(10,"dos")] = "uno"
   encuentra 102 [(40,"tos"),(103,"vela"),(16,"taza")] = ""
   encuentra 102 [] = ""
   ```

   Definir la función en forma recursiva pensando el caso base y caso inductivo.

9. La función `primIgualesA` toma un valor y una lista y devuelve el tramo inicial más largo de la lista cuyos elementos son iguales al valor. Por ejemplo:

   ```
   primIgualesA 3 [3,3,4,1] = [3,3]
   primIgualesA 3 [4,3,3,4,1] = []
   primIgualesA 3 [] = []
   primIgualesA 'a' "aaadaa" = "aaa"
   ```

   a) Programar `primIgualesA` con caso base e inductivo.

   b) Programarla usando la función `takeWhile` del Preludio.

   c) La función `primIguales` toma una lista y devuelve el mayor tramo inicial de la lista cuyos elementos son todos iguales entre sí. Por ejemplo:

   ```
   primIguales [3,3,4,1] = [3,3]
   primIguales [4,3,3,4,1] = [4]
   primIguales [] = []
   primIguales "aaadaa" = "aaa"
   ```

   Programar con caso base e inductivo `primIguales`.

   d) Usar cualquier versión de `primIgualesA` para programar `primIguales` sin recursión.

10. Programar utilizando recursión las funciones:

    a) `contarLa :: String -> Integer` que cuenta la cantidad de apariciones de la subcadena `"la"` en un string.

    Ayuda: *Pensar dos casos bases y caso inductivo.*

    b) `contarLas :: String -> Integer` que cuenta la cantidad de apariciones de la subcadena `"las"` en un string.

11. Definir por recursión la función `minimo` que devuelva el mínimo de una lista.

a) Definirla para listas no vacías.

b) Definirla para listas vacías limitando su tipo a la clase `Bounded` para poder definir el caso base.