Proyecto 1

Algoritmos y Estructuras de Datos I
Laboratorio

26 de agosto de 2011

El objetivo de este práctico es recordar (o aprender) 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). 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.

En todos los ejercicios escribe el tipo (y la clase) al que pertenece la función. Cuando haya que definir dos versiones de la función `ejemplo`, llama a la segunda versión `ejemplo'`.

Se recomienda hacer primero los ejercicios sin punto estrella. Una vez finalizados, revisados y testeados intentar recién, en ese momento, hacer los estrella.

1. Definir con lapiz y papel (se recomienda que apague la compu) funciones por recursión para cada una de las siguientes descripciones. Una vez obtenido este resultado escrito en una hoja, prográmelos en Haskell incluyendo su tipo.

   a) La función $paratodo$, que dado un predicado P sobre valores de tipo A (es decir una función que toma estos valores y devuelve booleanos) y una lista xs de elementos de tipo A, determina si todos los elementos de las lista cumplen el predicado. La función tiene la siguiente especificación:

   $$\begin{array}{|l} paratodo : (A \to Bool) \to [A] \to Bool \\ \hline paratodo.P.xs \doteq \langle\, \forall\, i \;:\; 0 \leq i < \#xs :\; P.(xs.i) \,\rangle \end{array}$$

   b) La función $existe$, que dado un predicado P sobre valores de tipo A y una lista xs de elementos de tipo A, determina si algún elemento de las lista cumple el predicado:

   $$\begin{array}{|l} existe : (A \to Bool) \to [A] \to Bool \\ \hline existe.P.xs \doteq \langle\, \exists\, i \;:\; 0 \leq i < \#xs :\; P.(xs.i) \,\rangle \end{array}$$

   c) La función $sumatoria$, que dada una función que toma elementos de tipo A y devuelve números, y una lista xs de elementos de tipo A, devuelve la suma de los elementos de la lista transformados por la función parámetro:

   $$\begin{array}{|l} sumatoria : (A \to Num) \to [A] \to Num \\ \hline sumatoria.f.xs \doteq \langle \sum i \;:\; 0 \leq i < \#xs :\; f.(xs.i) \,\rangle \end{array}$$

   d) La función $productoria$, que dada una función que toma elementos de tipo A y devuelve números, y una lista xs de elementos de tipo A, devuelve el producto de los elementos de la lista transformados por la función parámetro:

   $$\begin{array}{|l} productoria : (A \to Num) \to [A] \to Num \\ \hline productoria.f.xs \doteq \langle \prod i \;:\; 0 \leq i < \#xs :\; f.(xs.i) \,\rangle \end{array}$$

   e) La función $cantidad$, que dado un predicado P sobre valores de tipo A y una lista xs de elementos de tipo A, determina la cantidad de elementos que cumplen el predicado:

   $$\begin{array}{|l} cantidad : (A \to Bool) \to [A] \to Num \\ \hline cantidad.P.xs \doteq \langle \mathrm{N}\, i \;:\; 0 \leq i < \#xs :\; P.(xs.i) \,\rangle \end{array}$$

   Hacer dos definiciones: una recursiva utilizando caso base e inductivo, y otra utilizando la función $sumatoria$ anterior (sin recursión, en una sola definición).

Punto ⋆ 1 *Definirla utilizando la función sumatoria sin escribir la lista (segundo parámetro) en el lado izquierdo de la definición*[1]*. Esto es*

```
cantidad p = ...
```

Punto ⋆ 2 *Al escribir las tres últimas funciones en Haskell, se puede declara su tipo de forma tal que admitan distinto valores aritméticos en* `Int`, `Integer`, `Double`, *etc?*

Ayuda: *Usar clases.*

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

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

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

   c) $sumaCuadrados :: Num- > Num$ dado un número no negativo n devuelve la suma de los primeros n cuadrados:

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

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

   Pregunta: ¿Que sucede si $n > m$?

   d) $cantImpar :: [Num] \to Num$ devuelve la cantidad de numeros impares en la lista.

3. Notar que todos los ejercicios del punto 1 son similares: se aplica cierta función a cada elemento de la lista y se aplica algún operador entre todos los elementos transformados obteniendose así el resultado final.

   a) Guiándose por esta observación, definir de manera recursiva la función $cuantGen$ (denota la cuantificación generalizada) que toma un operador, una función término de la cuantificación y una lista, y aplica el operador a los elementos transformados por la función término:

   $$\begin{array}{|l} cuantGen : (A \to B \to B) \to (C \to A) \to [C] \to B \\ \hline cuantGen.\oplus.T.xs \doteq \langle \oplus i \ : 0 \leq i < \#xs : \ T.(xs.i) \ \rangle \end{array}$$

   Ayuda: *Al definir la función recursiva, se deberá agregar como parámetro el neutro del operador $\oplus$ que está implícito en la enterior definición. Esto es, la definición recursiva tendrá la forma:*

   $$\begin{array}{|l} cuantGen : (A \to B \to B) \to B \to (C \to A) \to [C] \to B \\ \hline cuantGen.\oplus.z.f.xs \doteq \cdots \end{array}$$

   con z el neutro del operador $\oplus$.

   b) Reescribir en Haskell todas las funciones del punto 2 utilizando el cuantificador generalizado (sin usar inducción y en una línea por función).

   Punto ⋆ 3 *Escribirlas con la menor cantidad de parámetros posibles utilizando aplicación parcial.*

Punto ⋆ 4 *La función* `map :: (a -> b) -> [a] -> [b]` *toma una función y una lista, transforma cada elemento de la lista con la función.*

La función `foldr :: (a -> b -> b) -> b -> [a] -> b` *toma un operador binario y lo aplica entre los elementos de una lista (teniendo en cuenta el caso base en el segundo parámetro).*

En base a esto y a la observación en el enunciado del ejercicio, escribir $cuantGen$ en función de `map` *y* `foldr` *en una sola línea (sin recursión).*

4. Considere las siguientes funciones:

[1] La técnica se llama aplicación parcial.

- `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) Escribe `sumarALista` y `encabezar` usando caso base y caso inductivo.

b) Escribe ambas funciones utilizando el funcional `map` . Esta vez, no puedes usar más de una línea, para cada una; y tampoco puede haber definiciones locales.

5. Programar una función que dada una lista de numeros devuelve aquellos que son pares.

a) Programarla con caso base e inductivo.

b) Buscar y utilizar una función en el preludio para poder programarla sin recursión.

Ayuda: *La función del preludio* filtra *los elementos de una lista que cumplen un predicado. Tiene tipo* `(a -> Bool) -> [a] -> [a]`.
Buscarla en `http://haskell.org/hoogle`.

6. 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 [] = ""
```

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

b) Programarla sin recursion utilizando la función del preludio en el ejercicio anterior.

7. 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) En el preludio existe una función que es más general que `primIgualesA`; utilizá esa función para programar `primIgualesA` en una sóla linea.

Ayuda: *La función en el preludio tiene tipo* `(a -> Bool) -> [a] -> [a]`.

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.

8. 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.

9. 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.