Repaso y simulacro de parcial

Taller de Álgebra I

Verano 2020

En las últimas clases estuvimos resolviendo problemas utilizando *recursión*.
Este método se basa en definir las funciones para dos casos escenciales:

Esquema recursivo

- El caso base (o *los casos base*).
- El caso general, basada en una solución para un parámetro "más cercano" al caso base.

¿A qué nos referimos con "más cercano"?

Si queremos calcular `f(n)` a partir de `f( g(n) )`, necesitamos que la función `g` sea tal que al aplicarla a `n`, y luego a su resultado, y a su resultado, . . . , nos lleve a un caso base **en una cantidad finita de aplicaciones**.

Con números naturales, una forma básica es definir `f(0)`, y luego `f(n)` utilizando `f(n-1)`.

Repaso de clases anteriores

Recursión en enteros

Repasemos algunos problemas que resolvimos definiendo funciones recursivas:

- Casos que se resuelven aplicando recursivamente la definición sobre el entero anterior

Factorial

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- Casos cuya definición recursiva no es respecto del entero inmediatamente anterior

División

```
dividir :: Integer -> Integer -> Integer
dividir a b | a < b      = 0
            | otherwise = 1 + dividir (a-b) b
```

Casos donde la recursión se debe realizar sobre un valor que no es el parámetro de entrada

Suma de divisores

Definir `sumaDivisores` que calcule la suma de todos los divisores positivos de n.

```
sumaDivisores :: Integer -> Integer
sumaDivisores n = sumaDivisoresHasta n n

sumaDivisoresHasta :: Integer -> Integer -> Integer
sumaDivisoresHasta n 0 = 0
sumaDivisoresHasta n m
          | n `mod` m == 0 = m + sumaDivisoresHasta n (m-1)
          | otherwise      = sumaDivisoresHasta n (m-1)
```

El esquema no cambia, pero debemos definir una función nueva con un parámetro adicional para poder aplicar la recursión.

Casos donde la recursión se debe realizar sobre múltiples parámetros

Sumas dobles

Definir una función que calcule $f(n, m) = \sum_{i=1}^{n} \sum_{j=1}^{m} i \times j$

```
sumaInterior :: Integer -> Integer -> Integer
sumaInterior n 0 = 0
sumaInterior n m = (n*m) + sumaInterior n (m-1)

sumaDoble :: Integer -> Integer -> Integer
sumaDoble 0 m = 0
sumaDoble n m = sumaInterior n m + sumaDoble (n-1) m
```

El esquema no cambia, pero la operación para obtener `sumaDoble n m` a partir de `sumaDoble (n-1) m` involucra, a su vez, una función recursiva en m.

Repaso de clases anteriores

Listas

El tipo lista (`[a]`) representa una colección de objetos de un mismo tipo.

A diferencia de las tuplas, las listas de distintas longitudes son del mismo tipo. Sin embargo, las tuplas pueden definirse con elementos de distintos tipos.

Los patrones fundamentales (*constructores*) de las listas son:

- `[]`, la lista vacía.
- `(x:xs)`, una lista con *head* `x` y *tail* `xs`.

Como con los enteros, vamos a usar recursión para resolver problemas sobre listas. Volvamos al esquema recursivo:

Esquema recursivo

- Definimos una solución para el caso base (o *los casos base*).
- Definimos una solución en el caso general, basada en una solución para un parámetro "más cercano" al caso base.

¿Cómo se aplica esta idea en una lista?

Vamos a necesitar saber resolver el problema para una lista con pocos (o ningún) elemento, y ser capaces de resolver el problema para una lista de n elementos suponiendo que tenemos la solución para una lista con **estrictamente menos** elementos.

Repaso de clases anteriores

Pattern Matching sobre Listas

Para preguntar si estamos en el caso base, podemos utilizar la función `length`, o, en el caso de la lista vacía, haciendo `xs==[]`.

Pero también podemos utilizar *pattern matching* para describir esta pregunta de manera más directa:

- Patrón para la lista vacía: `[]`
- Patrón para la lista con *exactamente* un elemento: `(x:[])`, `[x]`
- Patrón para la lista con *al menos* un elemento: `(x:xs)`
- Patrón para la lista con *exactamente* dos elementos: `(x:y:[])`, `[x,y]`
- Patrón para la lista con *al menos* dos elementos: `(x:y:xs)`

... Esto se puede extender hasta donde haga falta.

En cualquier posición, si el valor del parámetro en el patrón no me interesa, le puedo dar nombre _ para que Haskell lo descarte.

Ej.

```
head :: [a] -> a
head (x:_) = x
```

Las listas pueden ser tanto parte de la entrada como de la salida de una función:

- Funciones que toman una lista como entrada

Longitud

```
longitud :: [Integer] -> Integer
longitud []     = 0
longitud (_:xs) = 1 + longitud xs
```

- Funciones que devuelven una lista como salida

Listar Impares

```
imparesHasta :: Integer -> [Integer]
imparesHasta 0 = []
imparesHasta n | n `mod` 2 == 1 = n : imparesHasta (n-1)
               | otherwise      = imparesHasta (n-1)
```

También podemos tener un caso donde tanto la entrada como salida son listas

- Por ejemplo: definir una función que dada una lista l de enteros, devuelva una lista l' con los valores de l que son múltiplo de 7.

Filtrar listas

```
multiplosDe7 :: [Integer] -> [Integer]
multiplosDe7 []     = []
multiplosDe7 (x:xs) | x `mod` 7 == 0 = x:(multiplosDe7 xs)
                    | otherwise = multiplosDe7 xs
```

Todos estos esquemas son importantes!

Repasen los ejercicios que fueron vistos en clase, resuelvan los que no pudieron hacer antes y ¡consulten!

Departamento de Computación – FCEyN – UBA

Taller de Álgebra I - Simulacro de parcial

- El parcial se aprueba con tres ejercicios bien resueltos.
- Programe todas las funciones en Haskell. El código debe ser autocontenido. Si utiliza funciones que no existen en Haskell, debe programarlas. Incluya la signatura de todas las funciones que escriba.
- No está permitido alterar los tipos de datos presentados en el enunciado, ni utilizar técnicas no vistas en clase para resolver los ejercicios.

Ejercicio 1

Implementar la función `menorLex :: (Float, Float, Float) -> (Float, Float, Float) -> Bool` que dados dos vectores $x, y \in \mathbb{R}^3$ decida si x es menor a y en el sentido lexicográfico.

Por ejemplo:

`menorLex (3,-1,2) (5,10,0)` ⇝ `True` (pues $3 < 5$)
`menorLex (4,-1,7) (4,21,5)` ⇝ `True` (pues coinciden en la primera coordenada, y $-1 < 21$)
`menorLex (2,1,31) (2,1,-5)` ⇝ `False` (pues coinciden en las dos primeras coordenadas, pero $31 \not< -5$)

Ejercicio 2

Implementar una función `sumaFibonacci :: Integer -> Integer` que para cada $n \geq 1$ calcule $\sum_{j=0}^{n} f_n$, donde f_n es n–ésimo término de la sucesión de Fibonacci.

Por ejemplo:

`sumaFibonacci 2` ⇝ `1+1+2` ⇝ `4`
`sumaFibonacci 4` ⇝ `1+1+2+3+5` ⇝ `12`

Ejercicio 3

Implementar la funcion `esDefectivo :: Integer -> Bool` que dado un $n \in \mathbb{N}_{>0}$ determine si es *defectivo*, lo cual vale si y solo si la suma de los divisores propios positivos[1] de n es menor que n.

Por ejemplo:

`esDefectivo 16` ⇝ `True` (la suma de los divisores propios de 16 es $1 + 2 + 4 + 8 = 15$, que es menor que 16)
`esDefectivo 12` ⇝ `False` (la suma de los divisores propios de 12 es $1 + 2 + 3 + 4 + 6 = 16$, que no es menor que 12)

Ejercicio 4

Programe la función `maximaDistancia :: [Integer] -> Integer`, que determina cuál es la máxima distancia entre dos elementos consecutivos en una lista de números enteros.

Por ejemplo:

`maximaDistancia [1,6,2,7,8]` ⇝ `5` (la máxima distancia es entre 1 y 6)
`maximaDistancia [1,6,2,7,1]` ⇝ `6` (la máxima distancia es entre 7 y 1)
`maximaDistancia [1,5,-10,3]` ⇝ `15` (la máxima distancia es entre 5 y -10)

Aclaración: Puede asumir que la lista tiene al menos dos elementos.

Ejercicio 5

Programe la función `comprimir :: [Integer] -> [(Integer, Integer)]` que dada una lista de números enteros devuelva una lista que contenga una tupla (número, cantidad de apariciones) por cada ráfaga de números iguales adyacentes.

Por ejemplo:

`comprimir [1,1,7,7,4,4,1,4,4,4,3,3,3]` ⇝ `[(1,2),(7,2),(4,2),(1,1),(4,3),(3,3)]`

Sugerencia: Empiece reemplazando cada número n por una tupla (n, 1).

[1] Recordar: decimos que d es un *divisor propio* de n si $d \mid n$ y $d \neq n$.