Recursión

Programación funcional para la física computacional

21 de marzo de 2023

Funciones bien definidas

Una de las ventajas de la programación funcional es que, al basarse en funciones, se alienta a que la usuaria defina nuevas a conveniencia. Pero estas funciones deben estar bien definidas para evitar problemas al ejecutarse. Para nadie es raro el clásico problema de la división de dos números enteros que regresa una resultado en principio mal pues regresa un número natural a menos que se le indique por algún comando especial.

Para evitar tener que manejar distintos comandos extras, en *haskell* podemos decir que tipo de datos acepta y cuales regresa desde la misma definición de la función.

Ejemplos de como se pudedn definir nuevas funciones a partir de viejas funciones (ya en la librería de *haskell*)

- Una función que indica si el número dado es par, es decir, toma como valor un entero de precisión variable (Integral) y regresa un *booleano*.

```
pari :: Integral a => a -> Bool
pari n = n `mod` 2 == 0
```

- La función que divide una lista en sus primeros n elementos y los restantes, recibe una lista y regresa una lista de listas

```
divi :: Int -> [a] -> ([a],[a])
divi n xs = (take n xs, drop n xs)
```

- La función que resta los elementos correspondientes de cada lista

```
reste :: [Float] -> [Float] -> [Float]
reste xs ys = zipWith (-) xs ys
```

Pero podemos usar algunas cosas dentro del lenguaje para definir funciones más allá, por ejemplo, usando expresiones condicionales (cosa común en la definición de funciones en *python* y que también está en *haskell*)

- Expresiones condicionales para definir la función que regresa el valor absoluto de un entero

```
absol :: Int -> Int
absol n = if n >= 0 then n else -n
```

- Expresiones condicionales anidadas para que regrese el signo del número dado (como cadena)

```
signi :: Int -> String
signi n = if n < 0 then "negativo" else
            if n == 0 then "cero" else "positivo"
```

Ya hemos trabajado un poco con comprensión de listas, aunque no hemos visto mucho detalle. Lo que sabemos es que podemos usar *guardas* tal como se usan para definir conjuntos en matemáticas. Facilitan y reducen el tamaño de las definiciones, volvemos a definir la función de líneas arriba pero ahora de manera más sencilla:

```
signg :: Int -> String
signg n | n < 0 = "negativo"
        | n > 0 = "positivo"
        |otherwise = "cero"
```

La sección pasada vimos reconocimiento de patrones, no era tan sencillo para *haskell*, pero realmente esta la funcionalidad incluida en algunas de sus operaciones. Por ejemplo en las operaciones *booleanas* del tipo *True && False* lo que hace el lenguaje es reconocer los patrones y asociarlos a las operaciones lógicas. Pero hay otros casos de reconocimiento de patrones

- Patrones en eneadas

```
cabeza :: (a,b) -> a
cabeza (x,_) = x
```

- Lista de patrones

```
prob :: [Char] -> Bool
prob ['a',_,_] = True
prob _         = False
```

- Lista de patrones con el constructor

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

Como ya saben también podemos hacer uso de las definiciones con cálculo λ. El clásico ejemplo de definir la suma:

```
suma :: Int -> Int -> Int
suma = \x -> (\y -> x+y)
```

Definir funciones sólo por su estructura, sin tener que hacer alguna operación algebraica, en este caso sólo por el orden en que se dan los valores de entrada

```
consta :: a -> (b -> a)
consta x = \_ -> x
```

Incluso se pueden llamar a otras funciones dentro de la definición de la función, como en el siguiente caso:

```
impa :: Int -> [Int]
impa n = map f [0..n-1]
         where f x = x*2 + 1
```

O haciendo el mismo caso desde un nivel aún más básico

```
impa2 :: Int -> [Int]
impa2 n = map (\x -> x*2 + 1) [0..n-1]
```

Pero además este tipo de definiciones de funciones facilita definir operadores nuevos, por lo regular llamados *secciones*. La estructura en que se definen, de distintas formas, se muestra a continuación:

$$\begin{aligned}(\#) &= \backslash x -> (\backslash y -> x \# y)\\ (x\ \#) &= \backslash y -> x \# y\\ (\#\ y) &= \backslash x -> x \# y\end{aligned}$$

El primer caso espera dos entradas para operar con el nuevo operador, las dos últimas espera sólo una entrada para operar por la derecha o izquierda, respectivamente (podría ser un operador no conmutativo, así que el orden importa).

En ocasiones es complicado definir un nuevo operador, muchos de los símbolos ya están tomados por el sistema. Algunas de las funciones compactas propias de *haskell* definidas a partir de *secciones*:

- $(+)$ la suma que ya usamos
- $(1+)$ la función sucesor (la que suma 1 al único número dado)
- $(1/)$ el inverso de un número
- $(*2)$ el doble de un número
- $(/2)$ la mitad de un número

Comprensión de listas

Otro tema que ya hemos pasado de rápido es la comprensión de listas, veamos más a detalle. La idea más directa es que estaos definiendo conjuntos de una manera que se asemeja mucho a la notación matemática. Las *guardas*, que se escriben "|" funcionan como un *tal que* al leer la descripción.

- Podemos tener el caso donde hacemos una lista de parejas, de tener una lista más grande a emparejarse con otra de menor longitud se hace el apareamiento hasta que se llega al final de la más corta: $[(x, y) \mid x < -[1, 2, 3],\ y < -[4, 5]]$
- Si cambiamos el ordenamiento en la deinición, obtendremos las mismas parejas, pero con distinto orden: $[(x, y) \mid y < -[4, 5],\ x < -[1, 2, 3]]$

Podemos definir una función (que realmente ya existe en *haskell*) para conocer la longitud de una lista, pero a partir de comprensión de listas:

```
tam :: [a] -> Int
tam xs = sum [1| _ <- xs]
```

Un ejemplo que ya se saben obtener los números primos entre 1 y n, pero no de forma perezosa. Lo primero es identificar los factores de cada número entre 1 y n

```
factores :: Int -> [Int]
factores n = [x | x <- [1..n], n `mod` x == 0]
```

Agregamos para obtener los números primos a partir de la función ya definida para obtener los factores, todo hecho a partir de comprensión de listas. Lo que hace la función *primo* es llamar a la anterior función *factores* que da como resultado una lista y comparar si esa lista tiene como únicos factores al 1 y al mismo número, respondiendo un *verdadero* o *falso*.

Posteriormente la función *primo* es llamada dentro de la comprensión de listas de la función *primos* que nos regresará la lista de números entre 2 y n siempre y cuando el número cumpla con la condición de la función *primo*.

```
primo :: Int -> Bool
primo n = factores n == [1,n]

primos :: Int -> [Int]
primos n = [x | x <- [2..n], primo x]
```

Para buscar valores dentro de tablas

```
busca :: Eq a => a -> [(a,b)] -> [b]
busca k t = [v | (k',v) <- t, k==k']
```