Bitácora

Expresiones lambda, recursividad y fold

funcional currificación expresiones lambda recursividad fold

05 de Mayo, 2025

Tarea para la clase que viene:

- Hacer una copia en su drive del documento y resolver el TP Pattern Matching.
- Terminar la primera entrega del TP grupal de funcional.

¿Qué vimos hoy?

Expresiones lambda

Comenzamos viendo una nueva manera de crear funciones, ¡las **expresiones lambda**!

Si por ejemplo queremos obtener los dobles de una lista de números, hasta ahora podíamos hacerlo de esta manera...

```
dobles :: Num a => [a] -> [a]
dobles numeros = map doble numeros

dobles :: Num a => a -> a
doble numero = numero * 2
```

... o de esta...

```
dobles :: Num a => [a] -> [a]
dobles numeros = map (*2) numeros
```

Con expresiones lambda podríamos hacer lo siguiente:

```
dobles :: Num a => [a] -> [a]
dobles numeros = map (\numero -> numero * 2) numeros
```

¿Esto significa que a partir de ahora todas nuestras funciones las vamos a definir utilizando expresiones lambda? ¡No! ✋ Solo estamos viendo una nueva herramienta dentro del amplio espectro del paradigma funcional. De hecho, también se conoce a estas funciones como funciones anónimas , ya que, si bien podemos darles un nombre, solemos utilizarlas para casos muy específicos donde no nos interesa darle demasiada entidad a esa lógica. Entonces, ¿cuándo usarlas?

- Cuando no tenemos un buen nombre para ponerle a una función.
- Si sólo la vamos a usar una única vez, sino estamos repitiendo lógica.
- Si no necesitamos usar guardas ni pattern matching (con más de una ecuación).

  Y respecto a los 3 mundos del paradigma Funcional (el de los tipos, el de los valores y el de los patrones), las expresiones lambda sólo existen en el mundo de los valores.

Currificación y parámetros implícitos

```haskell
-- Sin currificar
sumaDe3Numeros :: Int -> Int -> Int -> Int
sumaDe3Numeros numero1 numero2 numero3 = numero1 + numero2 + numero3
```

¡Hay otra forma de escribir esto! 😮 Teniendo en cuenta que `sumaDe3Numeros` es una función, ¡lo podemos realizar con una lambda!:

```haskell
sumaDe3Numeros' :: (Int -> Int -> Int -> Int)
sumaDe3Numeros' = (\numero1 numero2 numero3 -> numero1 + numero2 + numero3)
```

A diferencia de `sumaDe3Numeros`, se utiliza una lambda para realizar su definición. Por eso (de forma didáctica), su tipo es la función `(Int -> Int -> Int -> Int)`.

Si realizamos esta consulta en consola:

```
> sumaDe3Numeros 5
<function>
```

Esto ocurre porque, como ya vimos, podemos crear funciones si llamamos, utilizando aplicación parcial, a otras funciones. Si vemos el tipo de `sumaDe3Numeros 5`:

```
>:t sumaDe3Numeros 5
(sumaDe3Numeros 5) :: Int -> Int -> Int
```

Ahora, si vemos el tipo de `sumaDe3Numeros`, ¿refleja bien lo que está ocurriendo? Veamos si hacemos, utilizando lambdas, algo que refleje mejor esto:

```haskell
sumaDe3Numeros'' :: (Int -> (Int -> Int -> Int))
sumaDe3Numeros'' = (\numero1 -> (\numero2 numero3 -> numero1 + numero2 + numero3))
```

¡Gracias a esto podemos aplicar parcialmente nuestras funciones! Cuando llamamos a `sumaDe3Numeros'' 5`, nos va a devolver la segunda función lambda que creamos.

¡Pero sabemos que `sumaDe3Numeros 5 5` también nos devuelve una función!

Si queremos hacer una función que replique todos los casos, tendríamos que hacer algo así:

```haskell
sumaDe3Numeros''' :: (Int -> (Int -> (Int -> Int)))
sumaDe3Numeros''' = (\numero1 -> (\numero2 -> (\numero3 -> numero1 + numero2 + numero3)
```

¿Esto quiere decir que ahora todas las funciones las tenemos que hacer así? ¿Y tiparlas así? 😨 De nuevo: **¡no!** A lo que llegamos, es que *Haskell hace esto sin que nos demos cuenta*: lo que hace haskell es “partir” nuestra función en diferentes funciones de 1 parámetro, es decir, *currifica* nuestras funciones.

Ahora, entendiendo esto, llegamos a por qué existe aplicación parcial: si le pasamos un parámetro a `sumaDe3Numeros`, nos va a devolver la *siguiente* función, que toma dos parámetros y nos devuelve un entero.

En Haskell, a veces, podemos dejar implícitos los parámetros que se pasan a las funciones. ¿Esto qué significa? Que no es necesario escribir a la izquierda del `=` que estamos recibiendo ese parámetro. Veamos un ejemplo:

```haskell
siguiente :: Int -> Int
siguiente numero = (+) 1 numero
-- es equivalente a
siguiente = (+) 1
```

Esto es porque al haber aplicado un `1` a la función `+`, nos va a devolver una función `Int -> Int`. Justamente, ¡¡gracias a que todas las funciones en Haskell

están currificadas!!

```
(+)   :: Int -> Int -> Int
(+) 1 ::          Int -> Int
```

En este caso, `siguiente` estaría "recibiendo implícitamente" un `numero :: Int`. Y lo que estamos haciendo es simplemente darle un nuevo nombre a esa función, porque es un valor.

Los parámetros implícitos también son útiles y frecuentemente vistos en los casos en los que componemos funciones.

Volviendo al *TP "Hora de lectura"*, teníamos esta función:

```
nombreDeLaBiblioteca :: Biblioteca -> String
nombreDeLaBiblioteca unaBiblioteca = sinVocales . concatenatoriaDeTitulos $ unaBibliote
```

Ahora que sabemos que podemos dejar implícitos nuestros parámetros, podríamos reescribirla de esta manera:

```
nombreDeLaBiblioteca :: Biblioteca -> String
nombreDeLaBiblioteca = sinVocales . concatenatoriaDeTitulos
```

En este caso `nombreDeLaBiblioteca` recibe implícitamente a una `biblioteca :: Biblioteca`. Esto es porque a la derecha del igual tenemos una función `Biblioteca -> Biblioteca`, a la que le queremos poner un nombre, porque para nosotros esa función significa `nombreDeLaBiblioteca`.

Esto no significa que de aquí en adelante sea importante dejar implícitos nuestros parámetros. Podemos hacerlo o no y nuestras funciones resolverán exactamente los mismos problemas de la misma forma ya que la lógica no cambia.

Patrones de listas

Además del ya conocido patrón de lista vacía (`[ ]`), existe otro patrón que nos puede resultar útil: el de *cabeza y cola* `(cabeza:cola)`. De esta forma estamos describiendo una lista de por lo menos un elemento. Si quisiéramos describir a una lista de por lo menos dos elementos, podríamos aplicar el patrón: `(unElemento:otroElemento:cola)`.

Recursividad

- Caso base: corta la recursividad.
- Caso recursivo: donde la función se llama a sí misma.

Ejemplos comunes de esto son la sucesión de fibonacci y el factorial !:

```
factorial :: Int -> Int
factorial 0 = 1                     -- caso base
factorial n = n * factorial (n - 1) -- caso recursivo

fibonacci :: Int -> Int
fibonacci 0 = 0                                  -- caso base
fibonacci 1 = 1                                  -- caso base
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)  -- caso recursivo
```

Pero esto no se reduce solo a funciones matemáticas que rara vez usemos en nuestros programas, la recursividad también sirve para funciones más comunes corrientes como `length` que nos permite saber el largo de una lista:

```
length' :: [a] -> Int
length' [ ]    = 0              -- caso base
length' (_:xs) = 1 + length' xs -- caso recursivo
```

Acá podemos aprovechar y ver qué significa cada parte de la declaración de la función.

- Cuando decimos `length' [ ] = ...` estamos diciendo que cuando la lista encaje con ese patrón (`[ ]` es el patrón de lista vacía) la función devuelve lo que está a la derecha.
- Cuando ponemos `length' (_:xs) = ...` estamos diciendo que cuando la lista tenga cabeza y cola (el patrón es `(cabeza:cola)`) la función devuelve lo que está del lado derecho. Acá es importante ver cómo usamos la variable anónima (`_`) para decir que queremos que tenga cabeza pero que no nos importa que valor tiene la cabeza.
- Tiene que quedar claro que lo mismo escrito del lado izquierdo del igual y del lado derecho no tienen el mismo significado, cuando vemos lo siguiente: `(x:xs)` no podemos decir si eso corresponde al patrón de lista (cabeza:cola) o si corresponde a usar la función `:` con `x` y `xs`. Lo mismo sucede con el patrón de lista vacía y la lista vacía (en ambos casos es `[ ]`, del lado izquierdo patrón y del lado derecho lista vacía).

Ahora, intentemos hacer la definición de `sum'` y `product'` de manera recursiva:

```
sum' :: Num a => [a] -> a
sum'           [ ] = 0
sum' (cabeza:cola) = cabeza + sum' cola

product' :: [Int] -> Int
product'             [ ] = 1
product' (cabeza : cola) = cabeza * product' cola
```

Si comparamos con las definiciones anteriores de `sum'` y `product'`, vemos que hay una repetición de lógica:

- En ambas definiciones esperamos que, cuando la lista esté vacía, retornemos un caso base.
- Luego, en ambas definiciones realizamos una operación que involucra el primer elemento de la lista, un operador (`+` o `*`) y una llamada recursiva de la función que estamos definiendo con la cola de la lista como parámetro.

Fold

Para solucionar este problema de repetición de lógica, surge `fold`:

```
-- Tipo de foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

-- Caso base: si la lista está vacía, retorno el acumulador

foldr _ acumulador [ ]     =  acumulador

-- Caso recursivo: si la lista no está vacía, ejecuto el operador con la cabeza de la l

foldr operador acumulador (cabeza : cola) = cabeza `operador` foldr operador acumulador
```

Ahora, las funciones de `sum'` y `product'` las podemos realizar sin repetir lógica:

```
sum' lista = foldr (+) 0 lista

product' lista = foldr (*) 1 lista

sumarUno valorAnterior _ = 1 + valorAnterior
length' lista = foldl sumarUno 0 lista
```

También, existe la función `foldl` que tiene la misma funcionalidad que `foldr` *pero* aplica la función recursiva cambiando la posición de los parámetros de la función que le pasamos por parámetro. ¡Así que cuidado con usarla con funciones que no sean asociativas! Por ejemplo:

```haskell
-- Como la suma es asociativa y conmutativa, obtenemos el mismo resultado si hacemos:

foldr (+) 0 [1,2,3,4]
> 10

-- o:

 foldl (+) 0 [1,2,3,4]
> 10

-- Pero si utilizamos fold con la resta (que no es asociativa ni conmutativa):
foldr (-) 0 [1,2,3,4]
> -2

-- No obtenemos los mismos resultados:

foldl (-) 0 [1,2,3,4]
> -10
```

¿Y qué pasa en los casos donde no se puede incluir un acumulador? (Por ejemplo, averiguar el máximo número de una lista de números). Para esto, tenemos las funciones `foldl1` y `foldr1`. En los siguientes ejemplos podemos ver que toma como acumulador al primer elemento de la lista:

```haskell
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 o (x:xs) = foldl funcion x xs

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 funcion (x:xs) = foldr funcion x xs
```

Links Útiles

- Código de la clase
- Expresiones Lambda
- Currificación
- Recursividad en haskell
- La familia fold
- Formas interactivas de ver fold y demás temas de hoy: https://stevekrouse.com/hs.js/ y https://pbv.github.io/haskelite/site/index.html

---

Podes ver nuestro github acá.
Y saber mas sobre nosotros acá

← Práctica de repaso y Git

Corrección de Entrega 1 TP Funcional →