Aplicación parcial y orden superior

funcional aplicacion-parcial orden-superior pattern-matching tuplas listas

14 de Abril, 2025

Tarea para la clase que viene:

Resolver el TP “Hora de lectura”. **Importante**: si bien en esta bitácora está la resolución paso a paso y el código de la misma, sugerimos que intenten resolver por su cuenta el ejercicio antes de ver la solución (¡y preguntarnos lo que no se entienda!).

¿Qué vimos hoy?

Los nuevos temas fueron:

- Aplicación parcial: aplicar a una función con menos argumentos de los “normales”, para obtener otra que espere los faltantes. Por ejemplo:

```
> max 6 9
9
-- max está aplicada totalmente (tiene los dos parámetros) y devuelve 9.

> max 6
<function>
-- en esta ocasión max está aplicada parcialmente (le falta un parámetro) y devuelve un

-- otro ejemplo con composición redefiniendo doble del siguiente
dobleDelSiguiente :: Num a => a -> a
dobleDelSiguiente unNumero = (*2).(+1) $ unNumero
```

- Tuplas: conjunto de elementos de diferentes tipos cuya longitud es fija. Por ejemplo:

```
("Pepita", 38176598)
-- una dupla (tupla de dos elementos) con un string y un número.
```

Para las duplas ya tenemos definidas `fst` y `snd` que devuelven el primer y el segundo elemento, respectivamente, de una tupla de dos elementos.

```
fst ("Pepita", 38176598)
"Pepita"
```

```
snd ("Pepita", 38176598)
38176598
```

También existen las n-ternas, o sea, tuplas de N elementos. Por ejemplo:

```
("hola", 1, 'e')
-- una terna (tupla de 3 elementos) compuesta de elementos de diferentes tipos.
```

Para hacer el equivalente al `fst` y `snd` de las duplas, existen lo que llamamos un **accessor**.

```
primerElemento :: (String, Int, String) -> String
primerElemento (primero, _, _) = primero
primerElemento ("hola", 1, 'e')
>> "hola"
-- una terna (tupla de 3 elementos) compuesta de elementos de diferentes tipos.
```

El uso del accessor es posible gracias a **pattern matching**, que es el concepto asociado al chequeo estructural de un dato respecto de una estructura esperada. Gracias a esto podemos tener un código más declarativo y simple. Sin embargo, su desventaja es que depende de los cambios de estructuras. Imaginémonos que nosotros agregamos un cuarto elemento a la tupla. Esto haría que no fuera posible utilizar nuestro accessor inicial, ya que estructuralmente la tupla cambió.

Algo más para decir de esta función es que está usando **variables anónimas** (los _). Las vamos a utilizar cuando necesitemos recibir un parámetro pero que no nos interesa conocer su valor (no nos es útil conocerlo) para la definición de la función.

Una cuestión muy importante a tener en cuenta es que en funcional existen tres mundos: **el de los valores, el de los tipos y el de los patrones**.

```
nombreDeLaFuncion :: Mundo de los Tipos
nombreDeLaFuncion Mundo de los Patrones = Mundo de los Valores
```

Las variables anónimas solo viven en el mundo de los patrones. Por lo tanto, **van del lado izquierdo del igual y nunca deben ir del lado derecho ni en el tipado.** En el mundo de los tipos no puede ir porque tenemos que especificar los tipos de nuestras funciones y valores, aún cuando son variables. En el de los valores tampoco tiene sentido porque es donde especificamos nuestros retornos, no podemos devolver “lo que sea”.

- Listas: conjunto de elementos de un mismo tipo. Por ejemplo:

```
[1, 2, 4, 5, 6, 8, 10, 100, 20000]
-- listas de números.

["die", "bart", "die"]
-- lista de strings.

[True, True]
-- lista de booleanos.

[("@skinnerOk", "Es una aurora boreal"), ("@archuN", "puedo verla??"), ("@skinnerOk", "
-- listas de tuplas (representan un tweet).

[]
-- lista vacía.
```

Vimos que hay varias funciones que podemos usar con las listas:

```
> length ["hola", "¿cómo", "estás?"]
3
> length [6,7,8,9,10,11,12]
7
-- length: devuelve la cantidad de elementos de la lista.

> head [1,2,3,4]
1
-- head: devuelve el primer elemento de una lista

> tail [1,2,3,4]
[2,3,4]
-- tail: devuelve una nueva lista pero sin el primer elemento de la original



> elem 1 [1,2,3,4]
True

> elem 5 [1,2,3,4]
False
-- elem: devuelve si un elemento forma parte o no de una lista

> sum [1,2,3,4]
10
-- sum: devuelve la suma de todos los elementos de una lista. ¡Sólo funciona con lista

> any even [1,2,3,4]
True
-- any: devuelve si alguno de los elementos de la lista cumplen la condición pasada por

> all odd [1,2,3,5,7]
False
-- all: devuelve si todos los elementos de la lista cumplen la condición pasada por par
```

```
> filter (>4) [1,2,7,1,9]
[7,9]
-- filter: dada una condición (función que devuelve un booleano) y una lista, devuelve

> map length ["hola", "murcielago"]
[4, 10]

> map (+1) [100, 41, 26]
[101, 42, 27]
-- map: dada una función y una lista, devuelve otra lista que contenga a los elementos
```

Las funciones `map` y `filter` (y otras más que iremos viendo o ya vimos, como la composición (`.`) o la aplicación (`$`)) son llamadas de **orden superior** porque reciben por parámetro otra función . ¡El orden superior es buenísimo porque nos permite crear funciones que reciban comportamiento (otras funciones) como argumento! De esa forma podemos pensar de forma mucho más declarativa:

- Puedo aislar y reutilizar comportamiento común.
- Puedo partir mi problema, separando responsabilidades, entre el código que tiene orden superior, y el comportamiento parametrizado.
- Puedo tener un código con partes incompletas, esperando rellenarlos pasando comportamiento por parámetro, y no sólo datos.

¡Ahora sí! Resolvamos el ejercicio "Hora de Lectura". Pero... ¿Por dónde empezamos? 😄 Si vamos a trabajar con libros, empecemos por ahí. ¡A modelarlos!

⚠ Disclaimer: Es muy importante leer todo el enunciado antes de ponerse a codear. En este caso, vamos a ir a nuestro ritmo sólo por fines pedagógicos. ⚠

Tenemos que crear cada libro, para eso, vamos a crear... ¿variables? ¡No! ☹ En funcional **no existen las variables** porque las cosas no varían. Recordá: **en Haskell no hay estado**. Es por eso que vamos a crear **etiquetas** representando a cada libro. ¿Y cómo los vamos a representar? Bueno, sabemos que cada libro tiene un título, un autor y una cantidad de páginas, entonces podríamos crear a *"El visitante"* y a *"Shingeki no Kyojin capítulo 1"* de esta forma:

```
elVisitante :: (String, String, Int)
elVisitante = ("el visitante", "Stephen King", 592)

shingekiNoKyojin1 :: (String, String, Int)
shingekiNoKyojin1 = ("shingeki no kyojin 1 ", "Hajime Isayama", 40)
```

Y así seguimos con los demás títulos. Peeeero, antes de avanzar, ¿no hay algo que te llame la atención? Pongámonos a filosofar: ¿qué es `elVisitante`? ¿qué es `shingekiNoKyojin1`? ¡Son libros ! Entonces, ¿no estaría bueno poder llamarlos como corresponde?

```
elVisitante :: Libro
elVisitante = ("el visitante", "Stephen King", 592)

shingekiNoKyojin1 = Libro
shingekiNoKyojin1 = ("shingeki no kyojin 1 ", "Hajime Isayama", 40)
```

Por suerte, esto no va a quedar en un deseo. Lo vamos a poder hacer realidad con el **type alias** (en criollo: un apodo o un alias). Gracias a esta herramienta vamos a lograr que nuestro código sea más expresivo 😊. Entonces, para hacer esto posible deberemos agregar a nuestra solución un type alias:

```
type Libro = (String, String, Int)

elVisitante :: Libro
elVisitante = ("el visitante", "Stephen King", 592)

shingekiNoKyojin1 = Libro
shingekiNoKyojin1 = ("shingeki no kyojin 1 ", "Hajime Isayama", 40)
```

¡Y podemos seguir mejorando la expresividad de nuestro código! Si sabemos que nuestro libro es una terna compuesta por el título, autor y la cantidad de páginas... Mirá :

```
type Titulo = String
type Autor = String
type CantidadDePaginas = Int
type Libro = (Titulo, Autor, CantidadDePaginas)

elVisitante :: Libro
elVisitante = ("el visitante", "Stephen King", 592)

shingekiNoKyojin1 = Libro
shingekiNoKyojin1 = ("shingeki no kyojin 1 ", "Hajime Isayama", 40)
```

Y una vez que hayamos modelado todos los libros, ¡es hora de armar la biblioteca! Para eso vamos a usar vari... ¡Noooo! 😫 Crearemos una etiqueta. ¿Y cómo va a ser la biblioteca? Bueno, una lista con los libros que modelamos. ¿Y su tipo? ¿Una lista de `(String, String, Int)`? ¿O una lista de `Libro`s? Si bien dijimos que `(String, String, Int)` y `Libro` eran lo mismo, para darle un sentido semántico a nuestra solución, vamos a elegir como tipo de la biblioteca a lista de `Libro`s:

```
biblioteca :: [Libro]
biblioteca = [elVisitante, shingekiNoKyojin1, shingekiNoKyojin3, shingekiNoKyojin27, fu
```

Sabemos lo que estás pensando... también nos gustaría tomar un helado . ¿Eh? ¿Eso no era lo que pensabas? 😳 No bueno, sí, también creemos que sería una buena idea crear un type alias para la biblioteca 😄:

```
type Biblioteca = [Libro]

biblioteca :: Biblioteca
biblioteca = [elVisitante, shingekiNoKyojin1, shingekiNoKyojin3, shingekiNoKyojin27, fu
```

Hagamos un recreíto de tanto código . Te vamos a contar un secreto : ¿viste el `String`? Bueno, ¡también es un apodo! ¿Te imaginás cuál es su verdadero nombre? Es `type String = [Char]`. Así es, un `String` no es nada más ni nada menos que una lista de caracteres, una lista de `Char`. ¿Eso significa que a partir de ahora vamos a llamarle `[Char]`? ¡Nooo! El `String` va a seguir llamándose `String`, por algo alguien decidió crear ese type alias. ¿No nos crees? Mirá este ejemplo:

```
> "¡Hola!" == ['¡', 'H', 'o', 'l', 'a', '!']
True
```

El string `¡Hola!` es lo mismo que la lista de caracteres `['¡', 'H', 'o', 'l', 'a', '!']`. Porque como dijimos, `String` es un type alias de `[Char]`. ¡Sigamos con el TP!

Ya modelamos los libros y la biblioteca. Es hora de definir las funciones que nos piden . La primera es `promedioDePaginas`. ¿Por dónde la encaramos 😨? **Nuestro consejo es empezar por el tipo de la función**. De esa forma, vamos a tener en mente qué parámetros toma y qué devuelve para luego definirla. 😊 ¿Cuántos parámetros toma `promedioDePaginas`? Uno solo, la biblioteca. Entonces, vamos a empezar poniendo una sola flechita (**recordá que el tipo de una función tiene la misma cantidad de flechitas que de parámetros**):

```
promedioDePaginas :: ...  -> ...
```

A veces, es muy claro qué devuelve una función, por lo que podemos empezar completando ese espacio del tipo de la misma. En este caso, como vamos a hacer un promedio, vamos a obtener como resultado un número entero:

```
promedioDePaginas :: ...  -> Int
```

¿Y de qué tipo es el parámetro que toma? Dijimos que era una biblioteca y el tipo de la biblioteca es `Biblioteca`:

```
promedioDePaginas :: Biblioteca -> Int
```

¡Wooohooo ! Tenemos el tipo de nuestra función. Ahora, definámosla:

```
promedioDePaginas :: Biblioteca -> Int
promedioDePaginas unaBiblioteca = div (cantidadDePaginasTotales unaBiblioteca) (length

cantidadDePaginasTotales :: Biblioteca -> Int
cantidadDePaginasTotales unaBiblioteca = sum . map cantidadDePaginas $ unaBiblioteca

cantidadDePaginas :: Libro -> Int
cantidadDePaginas (_, _, unasPaginas) = unasPaginas
```

Algo muy importante que hicimos en esta solución fue **delegar**. Es decir, dividimos nuestro gran problema en partecitas más pequeñas para poder resolverlo más fácilmente. De esta forma obtenemos un código más **declarativo**. Peeero, tampoco debemos irnos al extremo de sobredelegar: haber creado la función `cantidadDeLibrosDeLaBiblioteca` (que recibe una biblioteca y nos devuelve su longitud) es lo mismo que hacer directamente `length biblioteca`. La razón por la que no está bueno sobredelegar es que no estamos creando funciones que hagan nuevas cosas, sino que sólo estamos renombrando funciones que ya existen y conocemos.

Ahora toca el turno de definir `esLecturaObligatoria`. De vuelta vamos a descomponerla en funciones para que sea más fácil construirla. Un consejo que solemos dar es definir la función de tal forma que cuando la leamos, quede igual que el enunciado. Y la única forma de hacer esto posible es delegando:

```
type Saga = [Libro]

sagaDeEragon :: Saga
sagaDeEragon = [eragon, eldest, brisignr, legado]

autor :: Libro -> Autor
autor (_, unAutor, _) = unAutor

esLecturaObligatoria :: Libro -> Bool
esLecturaObligatoria unLibro = esDeStephenKing unLibro || perteneceASagaEragon unLibro

esDeStephenKing :: Libro -> Bool
esDeStephenKing unLibro = ((== "Stephen King") . autor) unLibro

perteneceASagaEragon :: Libro -> Bool
perteneceASagaEragon unLibro = elem unLibro sagaDeEragon

esFundacion :: Libro -> Bool
esFundacion unLibro = unLibro == fundacion
```

Fijate que la función `esLecturaObligatoria` quedó igual que el enunciado; el mismo dice: *"Es una lectura obligatoria cuando es de Stephen King o de la saga de Eragon o es el ejemplar de Fundación de 230 páginas de Isaac Asimov"*. A esto nos referimos con delegar y que se pueda leer como una oración del TP. 😁

Hagamos una observación : cuando creamos la `sagaDeEragon`, le pusimos como tipo `Saga`, donde saga es `[Libro]`. ¡Lo mismo que la biblioteca! ¿Y por qué no reutilizamos el tipo `Biblioteca` si también es `[Libro]` ? Porque si bien *sintácticamente* son lo mismo, *semánticamente* no lo son. Es decir, si bien las dos son del tipo `[Libro]`, una biblioteca no es lo mismo que una saga (y si no nos crees, buscalas en el diccionario 😉). Haciendo esta diferencia ganamos expresividad.

¡Sigamos! Es el turno de `esFantasiosa`. Comencemos con su tipo, así que como recibe un parámetro, ponemos una flechita:

```
esFantasiosa :: ... -> ...
```

Sabemos que devuelve un booleano, por lo tanto:

```
esFantasiosa :: ... -> Bool
```

Y sabemos que toma una biblioteca:

```
esFantasiosa :: Biblioteca -> Bool
```

Tadáaa , tenemos el tipo de nuestra función. Ahora definámosla:

```
esFantasiosa :: Biblioteca -> Bool
esFantasiosa unaBiblioteca = any esLibroFantasioso unaBiblioteca

esLibroFantasioso :: Libro -> Bool
esLibroFantasioso unLibro = esDeChristopherPaolini unLibro || esDeNeilGaiman unLibro

esDeChristopherPaolini :: Libro -> Bool
esDeChristopherPaolini unLibro = ((== "Christopher Paolini") . autor) unLibro

esDeNeilGaiman :: Libro -> Bool
esDeNeilGaiman unLibro = ((== "Neil Gaiman") . autor) unLibro
```

Mmmm, un momento ✋. Algo está oliendo mal … ¡a repetición de lógica! Mirá estas tres funciones:

```
esDeStephenKing :: Libro -> Bool
esDeStephenKing unLibro = ((== "Stephen King") . autor) unLibro

esDeChristopherPaolini :: Libro -> Bool
esDeChristopherPaolini unLibro = ((== "Christopher Paolini") . autor) unLibro

esDeNeilGaiman :: Libro -> Bool
esDeNeilGaiman unLibro = ((== "Neil Gaiman") . autor) unLibro
```

Son prácticamente iguales 😱. En todas se **repite la lógica** de obtener el autor de un libro para fijarnos si es un autor en especial 😵. Para solucionar esto, vamos a crear una función que tenga sólo la lógica repetida, parametrizando lo único que cambia (que en este caso son los nombres de los autores):

```
esDe :: Autor -> Libro -> Bool
esDe unAutor unLibro = ((== unAutor) . autor) unLibro
```

Nuestra solución ahora quedaría así:

```
esFantasiosa :: Biblioteca -> Bool
esFantasiosa unaBiblioteca = any esLibroFantasioso unaBiblioteca

esLibroFantasioso :: Libro -> Bool
esLibroFantasioso unLibro = esDe "Christopher Paolini" unLibro || esDe "Neil Gaiman" un
```

Sigamos con `nombreDeLaBiblioteca`:

```
titulo :: Libro -> String
titulo (unTitulo, _, _) = unTitulo

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

sinVocales :: String -> String
sinVocales unString = filter (not . esVocal) unString

esVocal :: Char -> Bool
esVocal unCaracter = elem unCaracter "aeiouAEIOUÁÉÍÓÚ"

concatenatoriaDeTitulos :: Biblioteca -> String
concatenatoriaDeTitulos unaBiblioteca = concatMap titulo unaBiblioteca
```

Recordá que como un `String` es una `[Char]` es lo mismo “aeiouAEIOUÁÉÍÓÚ” que [‘a’, ‘e’, ‘i’, ‘o’, ‘u’, ‘A’, ‘E’, ‘I’, ‘O’ , ‘U’, ‘Á’, ‘É’, ‘Í’, ‘Ó’ , ‘Ú’] y es una forma mucho más fácil de escribirlo 😄. Si te quedó la duda de por qué repetimos las vocales en minúscula, mayúscula y con tildes, es para que matchee de las dos formas. 😄

¡Llegamos a la última función! La que nos dice si una biblioteca es ligera:

```
esBibliotecaLigera :: Biblioteca -> Bool
esBibliotecaLigera unaBiblioteca = all esLecturaLigera unaBiblioteca

esLecturaLigera :: Libro -> Bool
esLecturaLigera unLibro = ((<= 40) . cantidadDePaginas) unLibro
```

Y de esa forma completamos el TP usando las herramientas que aprendiste hasta ahora. La clase que viene seguimos.

Links Útiles

- Enunciado que hicimos en clase
- Código del TP
- Aplicación parcial
- Orden superior
- Pattern Matching

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

← Funcional, composición y tipos

Guardas y Data →