Guardas y Data

funcional guardas data

21 de Abril, 2025

Tarea para la clase que viene:

- Pueden hacer (casi) completas las guías de ejercicios de Funcional. En los últimos ejercicios de la última guía se pide que se resuelva con herramientas que todavía no se vieron, paciencia con esos, ya vas a poder realizarlos en unas semanas.
- Hacerse una cuenta de GitHub. Elegí un buen nombre porque te va a acompañar en lo que reste de la carrera y en tu vida profesional. Podés hacerlo con tu cuenta de mail personal y luego linkear el mail de la facultad para obtener los beneficios estudiantiles.

Guardas

Ahora queremos saber de qué género es un libro. Eso va a depender de:

- Si tiene menos de 40 páginas, es un cómic.
- Si el autor es Stephen King, es de terror.
- Si el autor es japonés, es un manga.
- En cualquier otro caso, no sabemos el género.

Aprendimos cómo ver si un libro tiene cierta cantidad de páginas o si es de un autor en especial pero... ¿cómo averiguamos la nacionalidad de un autor? 😅 ¿Deberíamos agregar la nacionalidad del autor en cada libro? ¡Momento ✋! Sólo interesa saber quiénes son de Japón y, como en nuestra solución, el único autor japonés es *"Hajime Isayama"*, no tendría sentido agregar más información a cada libro. La forma más fácil de resolverlo es con una función:

```haskell
esDeAutorJapones :: Libro -> Bool
esDeAutorJapones unLibro = elem (autor unLibro) autoresJaponenes

autoresJaponenes :: [String]
autoresJaponenes = ["Hajime Isayama"]

-- Si se llegaran a agregar otros autores japoneses, esta solución es más extensible.
```

¡Ya tenemos lo necesario para definir la función `genero`!

```haskell
genero :: Libro -> String
genero unLibro
  | esDe "Stephen King" unLibro = "Terror"
  | (esJapones.autor) unLibro = "Manga"
  | esLecturaLigera unLibro = "Comic"
  | otherwise = "Sin categoría"
```

Recordá no olvidarte el `otherwise` cuando utilices guardas ya que es donde entra todo lo que no abarcan las guardas de encima de él. Y, ¿por qué pasa eso? Resulta que `otherwise` es un sinónimo de `True`, por lo que siempre se va a poder entrar por esa condición cuando no se no cumplan ninguna de las demás. Utilizamos `otherwise` porque es más expresivo.

Veamos otra versión de `esLecturaObligatoria` con **pattern matching** (y nuestra versión preferida porque usa una herramienta del paradigma funcional y además, es más declarativa):

```haskell
esLecturaObligatoria' :: Libro -> Bool
esLecturaObligatoria' (_, "Stephen King", _) = True
```

```
esLecturaObligatoria' ("Fundacion", "Isaac Asimov", 230) = True
esLecturaObligatoria' unLibro = perteneceASagaEragon unLibro
```

⚠ Hay que tener mucho cuidado con el orden cuando utilizamos pattern matching. Los casos deben ir de lo más particular a lo más general. ⚠ En este caso `(_, "Stephen King", _)` y `(_, "Isaac Asimov", 230)` matchean con tuplas que tengan ese formato, mientras que `unLibro` matchea con cualquier tupla (por eso va después). Así vamos de los casos más específicos a los generales.

Y así como tenemos una solución preferida, tenemos una que no nos gusta para nada :

```
esLecturaObligatoria :: Libro -> Bool
esLecturaObligatoria unLibro
            | unLibro == eragon = True
            | unLibro == eldest = True
            | unLibro == brisignr = True
            | unLibro == legado = True
            | autor unLibro == "Stephen King" = True
            | unLibro == fundacion = True
    |otherwise = False
```

Usar **guardas** de esta forma es un 2 (2) automático en el parcial, un desaprobado. Es un **mal uso de booleanos** y una **muy muy mala práctica** de programación. Dicho esto, quien avisa no traiciona...

ACLARACIÓN DE RESTRICCIÓN DE TIPOS

Cuando definimos la función `modulo` lo hicimos indicando el tipo `modulo :: Num a => a -> a`. Esta restricción "le quedaba grande" a la función ya que en la primera guarda estamos fijándonos si `unNumero` es mayor a 0. Los tipos de datos que pueden ordenarse (pueden compararse por mayor/menor) son los de la familia `Ord` y la familia `Num` engloba también a los números complejos que no pueden ordenarse dada su naturaleza. Por eso es que si queremos restringir el tipo de dato de la función `modulo` a `Num`, debemos restringirlo aún más aclarando que debe pertenecer a la familia `Ord` de la siguiente forma: `modulo :: (Num a, Ord a) => a -> a`

Data

¡Excelente! Ya tenemos funcionando la función `genero` . ¿Qué pasa si le mandamos como argumento una tupla que representa a una persona? No debería funcionar porque explicitamos en su tipo que recibía un `Libro`... Veamos qué pasa con la tupla que representa a nuestro querido ex-profe Gus:

```
genero ("Gustavo", "Trucco", 32)
> "Comic"
```

¿¡Entonces Gus es un cómic!? 😱 Ya quisiera (es muy muy fanático de los cómics), pero no lo es. Lo que pasó es que si bien dijimos que `genero` funciona sólo con `Libro`s, un `Libro` es una tupla de tipo `(String, String, Int)`, ¡el mismo tipo que la tupla que representa a una persona! 😄 Recordá que al usar el type alias, **no estamos creando un nuevo tipo de dato**, sino que le estamos dando un nombre a una estructura que tiene sentido para nuestra solución y así ganar expresividad.

Entonces, ¿cómo lo solucionamos? Creando nuestro propio tipo de dato con **Data**:

```
data Libro = UnLibro Titulo Autor CantidadDePaginas
```

En donde `UnLibro` es una función que llamamos **constructor** y su tipo es `UnLibro :: Titulo -> Autor -> CantidadDePaginas -> Libro`. Es decir, es una función que recibe los parámetros necesarios para crear un libro.

Modelemos a "El visitante":

```
UnLibro "El visitante" "Stephen King" 592
```

Si quisiéramos probarlo en la consola, nos tiraría un error porque el data que construimos no es “mostrable” 😩. Es decir, Haskell no sabe cómo mostrar nuestro tipo de dato, pero lo solucionamos escribiendo `deriving Show` al final de la declaración del data:

```
data Libro = UnLibro Titulo Autor CantidadDePaginas deriving Show
```

Y entonces, ¿qué ventajas tenemos al usar data? Porque pareciera ser lo mismo que usar tuplas con el type alias . La diferencia está en que, con el data, estamos creando nuestro propio tipo de dato y, gracias a eso, vamos a poder restringir a las funciones a que sólo funcionen con el tipo de dato que le decimos. Ahora, `genero` sólo va a recibir `Libro`s, de otra forma, romperá.

Otra ventaja es que podemos utilizar data con **record syntax** y, de esta forma, nos genera automáticamente los accessors:

```
data Libro = UnLibro { titulo :: String, autor :: Autor, paginas :: Int } deriving Show
```

En este caso tanto `libro` como `autor` y `paginas` son funciones (accessors) que van a acceder a cada elemento del data . En conclusión, ambas sintaxis para definir datas son equivalentes, solo que record syntax nos regala las funciones para acceder a las propiedades.

Es importante tener en cuenta, que al utilizar data estamos creando un tipo (`Libro`), una función constructora (`UnLibro`) y un patrón (`UnLibro unTitulo unAutor paginas`).

Por otro lado, si queremos comparar una instancia de data con otra, tenemos que decirle a Haskell que queremos que sean comparables. ¿Cómo hacemos eso? Agregando `Eq`:

```
data Libro = UnLibro { titulo :: String, autor :: Autor, paginas :: Int } deriving (Sho
```

Ahora vamos a modelar la función `agregarPaginas`. ¿Esta función va a modificar al libro original? ¡No! Los data, al igual que todo en el paradigma funcional, siguen siendo inmutables. Por ende, la función nos devolverá una copia del libro con la cantidad de páginas aumentada.

```
agregarPaginas :: Libro -> Int -> Libro
agregarPaginas (UnLibro unTitulo unAutor unasPaginas) paginasAAgregar = UnLibro unTitul
```

Podemos hacer lo mismo con record syntax:

```
agregarPaginas :: Libro -> Int -> Libro
agregarPaginas unLibro paginasAAgregar = unLibro { paginas = paginas unLibro +  paginas
```

Es importante destacar que para devolver la nueva cantidad de páginas debemos sumar la cantidad de páginas original. Para eso, utilizamos el accessor `paginas` y es importante pasarle por parámetro `unLibro` para que pueda darnos el valor. No olvidar que `paginas` sigue siendo una función que necesita su parámetro.

En caso de que queramos crear un libro editando mas de un valor, podemos hacer lo mismo que antes, separando cada valor con una `,`. Veamos un ejemplo: sacarSecuela, la cual no solo le agrega 50 páginas a un libro sino que le agrega un 2 al nombre.

```
sacarSecuela :: Libro -> Libro
sacarSecuela unLibro = unLibro { titulo = titulo libro ++ " 2", paginas = paginas unLib
```

Links Útiles

- Data
- Nuevo enunciado “Hora de Lectura”
- Código de la clase

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

← Aplicación parcial y orden superior

Práctica de repaso y Git →