Tipos predefinidos

BÁSICOS

Enteros:

Int representa enteros. Se utiliza para representar número enteros, por lo que 7 puede ser un Int pero 7.2 no puede. Int está acotado, lo que significa que tiene un valor máximo y un valor mínimo. Normalmente en máquinas de 32bits el valor máximo de Int es 2147483647 y el mínimo -2147483648.

Integer representa... esto... enteros también. La diferencia es que no están acotados así que pueden representar números muy grandes. Sin embargo, Int es más eficiente.

```
factorial :: Integer -> Integer

factorial n = product [1..n]


ghci> factorial 50

30414093201713378043612608166064768844377641568960512000000000000
```

Flotantes:

Float es un número real en coma flotante de simple precisión.

```
circumference :: Float -> Float

circumference r = 2 * pi * r


ghci> circumference 4.0

25.132742
```

Carácter:

Es el tipo booleano. Solo puede tener dos valores: True o False.

Booleanos:

Representa un carácter. Se define rodeado por comillas simples. Una lista de caracteres es una cadena.

Compuestos

Funciones:

Cuando escribimos nuestras propias funciones podemos darles un tipo explícito en su declaración. Generalmente está bien considerado escribir los tipos explícitamente en la declaración de una función, excepto cuando éstas son muy cortas.

```
removeNonUppercase :: [Char] -> [Char]

removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

removeNonUppercase tiene el tipo [Char] -> [Char], que significa que es una función que toma una cadena y devuelve otra cadena. El tipo [Char] es sinónimo de String así que sería más elegante escribir el tipo como removeNonUppercase :: String -> String. Función que toma tres enteros y los suma:

```
addThree :: Int -> Int -> Int -> Int

addThree x y z = x + y + z
```

Los parámetros están separados por -> y no existe ninguna diferencia especial entre los parámetros y el tipo que devuelve la función. El tipo que devuelve la función es el último elemento de la declaración y los parámetros son los restantes. Más tarde veremos porque simplemente están separados por -> en lugar de tener algún tipo de distinción más explícita entre los tipos de parámetros y el tipo de retorno, algo como Int, Int, Int -> Int.

Tuplas:

Poseen tipos, pero dependen de su longitud y del tipo de sus componentes, así que teóricamente existe una infinidad de tipos de tuplas y eso son demasiados tipos como para cubrirlos en esta guía. La tupla vacía es también un tipo () el cual solo puede contener un valor: ().

Las tuplas son parecidas a las listas. Ambas son una forma de almacenar varios valores en un solo valor. Sin embargo, hay unas cuantas diferencias fundamentales. Una lista de números es una lista de números. Ese es su tipo y no importa si tiene un sólo elemento o una cantidad infinita de ellos. Las tuplas sin embargo, son utilizadas cuando sabes exactamente cuántos valores tienen que ser combinados y su tipo depende de cuantos componentes tengan y del tipo de estos componentes. Las tuplas se denotan con paréntesis y sus valores se separan con comas.

Utilizamos la tuplas cuando sabemos de antemano cuantos componentes de algún dato debemos tener. Las tuplas son mucho más rígidas que las listas ya que para cada tamaño tienen su propio tipo, así que no podemos escribir una función general que añada un elemento a una tupla: tenemos

que escribir una función para añadir duplas, otra función para añadir triplas, otra función para añadir cuádruplas, etc.

Como las listas, las tuplas pueden ser comparadas si sus elementos pueden ser comparados. Solo que no podemos comparar dos tuplas de diferentes tamaños mientras que si podemos comparar dos listas de diferentes tamaños. Dos funciones útiles para operar con duplas son:

- **Fst.** Toma una dupla y devuelve su primer componente.

  ghci> fst (8,11)

  8

  ghci> fst ("Wow", False)

  "Wow"

- **Snd.** Toma una dupla y devuelve su segundo componente. ¡Sorpresa!

  ghci> snd (8,11)

  11

  ghci> snd ("Wow", False)

  False

- **Zip.** Esta función toma dos listas y las une en un lista uniendo sus elementos en una dupla. Es una función realmente simple pero tiene montones de usos. Es especialmente útil cuando queremos combinar dos listas de alguna forma o recorrer dos listas simultáneamente.

  ghci> zip [1,2,3,4,5] [5,5,5,5,5]

  [(1,5),(2,5),(3,5),(4,5),(5,5)]

  ghci> zip [1 .. 5] ["uno","dos","tres","cuatro","cinco"]

  [(1,"uno"),(2,"dos"),(3,"tres"),(4,"cuatro"),(5,"cinco")]

Listas:

Las listas en Haskell son muy útiles. Es la estructura de datos más utilizada y pueden ser utilizadas de diferentes formas para modelar y resolver un montón de problemas. Las listas son MUY importantes. En esta sección daremos un vistazo a las bases sobre las listas, cadenas de texto (las cuales son listas) y listas intencionales

En Haskell, las listas son una estructura de datos homogénea. Almacena varios elementos del mismo tipo. Esto significa que podemos crear una lista de enteros o una lista de caracteres, pero no podemos crear una lista que tenga unos cuantos enteros y otros cuantos caracteres.

Podemos usar la palabra reservada let para definir un nombre en GHCi. Hacer let a = 1 dentro de GHCi es equivalente ha escribir a = 1 en un fichero y luego cargarlo.

```
ghci> let lostNumbers = [4,8,15,16,23,42]

ghci> lostNumbers

[4,8,15,16,23,42]
```

Una tarea común es concatenar dos listas. Cosa que conseguimos con el operador ++.

```
ghci> [1,2,3,4] ++ [9,10,11,12]

[1,2,3,4,9,10,11,12]

ghci> "hello" ++ " " ++ "world"

"hello world"

ghci> ['w','o'] ++ ['o','t']

"woot"
```

Cuando concatenamos dos listas (incluso si añadimos una lista de un elemento a otra lista, por ejemplo [1,2,3] ++ [4], internamente, Haskell tiene que recorrer la lista entera desde la parte izquierda del operador ++. Esto no supone ningún problema cuando trabajamos con listas que no son demasiado grandes. Pero concatenar algo al final de una lista que tiene cincuenta millones de elementos llevará un rato. Sin embargo, concatenar algo al principio de una lista utilizando el operador : (también llamado operador cons) es instantáneo.

```
ghci> 'U':"n gato negro"

"Un gato negro"

ghci> 5:[1,2,3,4,5]

[5,1,2,3,4,5]
```

[], [[]] y [[],[],[]] son cosas diferentes entre si. La primera es una lista vacía, la segunda es una lista que contiene un elemento (una lista vacía) y la tercera es una lista que contiene tres elementos (tres listas vacías).

Si queremos obtener un elemento de la lista sabiendo su índice, utilizamos !!. Los índices empiezan por 0.

```
ghci> "Steve Buscemi" !! 6

'B'

ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
```

33.2

Las listas también pueden contener listas. Estas también pueden contener a su vez listas que contengan listas, que contengan listas...

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

Funciones básicas que pueden operar con las listas:

- **Head.** Toma una lista y devuelve su cabeza. La cabeza de una lista es básicamente el primer elemento.

  ```
  ghci> head [5,4,3,2,1]
  5
  ```

- **Tail.** Toma una lista y devuelve su cola. En otras palabras, corta la cabeza de la lista.

  ```
  ghci> tail [5,4,3,2,1]
  [4,3,2,1]
  ```

- **Last.** Toma una lista y devuelve su último elemento.

  ```
  ghci> last [5,4,3,2,1]
  1
  ```

- **Init.** Toma una lista y devuelve toda la lista excepto su último elemento.

  ```
  ghci> init [5,4,3,2,1]
  [5,4,3,2]
  ```

¿Pero que pasa si intentamos obtener la cabeza de una lista vacía?

ghci> head []

*** Exception: Prelude.head: empty list

Cuando usamos head, tail, last e init debemos tener precaución de no usar con ellas listas vacías. Este error no puede ser capturado en tiempo de compilación así que siempre es una buena práctica tomar precauciones antes de decir a Haskell que te devuelva algunos elementos de una lista vacía.

- **Length.** Toma una lista y obviamente devuelve su tamaño.

  ghci> length [5,4,3,2,1]

  5

- **Null.** Comprueba si una lista está vacía. Si lo está, devuelve True, en caso contrario devuelve False. Usa esta función en lugar de xs == [] (si tienes una lista que se llame xs).

  ghci> null [1,2,3]

  False

  ghci> null []

  True

  reverse pone del revés una lista.

  ghci> reverse [5,4,3,2,1]

  [1,2,3,4,5]

- **Take.** Toma un número y una lista y extrae dicho número de elementos de una lista.
  ghci> take 3 [5,4,3,2,1]

  [5,4,3]

  ghci> take 1 [3,9,3]

  [3]

  ghci> take 5 [1,2]

  [1,2]

  ghci> take 0 [6,6,6]

  []

Si intentamos tomar más elementos de los que hay en una lista, simplemente devuelve la lista. Si tomamos 0 elementos, obtenemos una lista vacía.

- **Drop.** Funciona de forma similar, solo que quita un número de elementos del comienzo de la lista.

  ghci> drop 3 [8,4,2,1,5,6]

  [1,5,6]

  ghci> drop 0 [1,2,3,4]

  [1,2,3,4]

  ghci> drop 100 [1,2,3,4]

  []

- **Maximum.** Toma una lista de cosas que se pueden poner en algún tipo de orden y devuelve el elemento más grande.

- **Minimum.** Devuelve el más pequeño.

  ghci> minimum [8,4,2,1,5,6]

  1

  ghci> maximum [1,9,2,3,4]

  9

- **Sum.** Toma una lista de números y devuelve su suma.

- **Product.** Toma una lista de números y devuelve su producto.

  ghci> sum [5,2,1,6,3,2,5,7]

  31

  ghci> product [6,2,1,2]

  24

  ghci> product [1,2,5,6,7,9,2,0]

  0

- **Elem.** Toma una cosa y una lista de cosas y nos dice si dicha cosa es un elemento de la lista. Normalmente, esta función es llamada de forma infija porque resulta más fácil de leer.

  ghci> 4 `elem` [3,4,5,6]

  True

  ghci> 10 `elem` [3,4,5,6]

  False.