



Introducción a Haskell (Parte II)

Funciones:

Del mismo modo que los datos tienen tipos, también lo tienen las funciones. Los tipos de las funciones y los tipos de los argumentos son importantes porque indican cuál es el tipo de los argumentos de la función y cuál es el tipo del resultado que devuelve.

Una función que toma un argumento de tipo A y devuelve un resultado de tipo B tiene tipo $A \rightarrow B$ ("A flecha B", "A en B", etc). Si f tiene este tipo, escribimos $f : A \rightarrow B$. En Haskell se escribe:
`f :: A -> B.`

Ejemplos, los tipos de las siguientes funciones definidas en Haskell son:

- `max :: Integer -> Integer -> Integer`
- `not :: Bool -> Bool`
- `toUpper :: Char -> Char` (averiguar cuál es el comportamiento de esta función)

Haskell hace una cierta diferencia entre operador y función: Un operador en Haskell es una función, pero debe tener exactamente dos argumentos (función binaria), y se escribe entre los argumentos. Por ejemplo, para el operador + que toma dos números y devuelve su suma, se utiliza la notación explicada: `2 + 3`.

Como un operador es una función, tiene un tipo.

- Indique cuál es el tipo de +.

Cuando se desea usar el operador solo, por ejemplo, para ver su tipo, se deberá escribir entre paréntesis.

También es una forma alternativa de escribir los operadores cuando se desea utilizar notación prefija. Por ejemplo, una forma de escribir `2 + 3` es `(+) 2 3`, que indica que la función (+) se aplica a dos argumentos, 2 y 3.

¿Cómo definir nuevas funciones?

Se define una función indicando el tipo de la misma (declaración de tipo) y dando la expresión en forma de ecuación que indica lo que la función hace (el programa).

La declaración del tipo tiene la forma:

`nombre_de_función :: tipoArg1 -> tipoArg2 ->..... -> tipoArgn -> tipoResultado`

La definición de la ecuación tiene la forma:

`nombre_de_función arg1 arg2. . . argn = expresión que puede utilizar los argumentos o parámetros.`

Ejemplo:

Definir la función que dado un entero devuelve su cuadrado decrementado en 1.

Declaración del tipo:

`cuadrado_menos_uno :: Integer -> Integer`

Definición de la expresión correspondiente a la función:

`cuadrado_menos_uno x = x*x-1`

Probar esta definición en Haskell.

- Definir en Haskell una función que dado un entero devuelva su cuadrado decrementado en un entero dado y probarla.

Definición de funciones por casos

Queremos construir una función que devuelva True si el parámetro o argumento es 3 y False en otro caso. Debemos definir dos ecuaciones:

`esTres :: Integer -> Bool`

`esTres 3 = True`

`esTres x = False`

¿Cómo se comporta esta función? Dado el parámetro se fija si coincide con el caso que se presenta en la primer ecuación, entonces si el parámetro es 3, devuelve True y termina. Si no coincide sigue buscando, encuentra que sí coincide con el caso de la segunda ecuación que x representa un entero cualquiera, por lo tanto devolverá False.

- Probar dicha función en Haskell con diferentes parámetros.
- Escribir una función que toma un caracter y devuelve True si el caracter es 'a' y False en caso contrario.

Ejercicios:

1. Definir una función `maximo :: (Integer, Integer) → Integer` que devuelve el mayor de sus dos argumentos.
2. Definir una función `par :: Integer → Bool` que indica si su argumento es par (Sugerencia: Utilizar el operador 'mod').
3. Definir una función `max3 :: (Integer, Integer, Integer) → Integer` que devuelve el máximo de sus argumentos.
4. Definir la función `sgn :: Int → Int` que dado un número devuelve 1, 0 ó -1, en caso que el número sea positivo, cero o negativo respectivamente.
5. Definir la función `abs :: Int → Int` que calcula el valor absoluto de un número.
6. Definir el predicado `bisiesto :: Int → Bool` que determina si un año es bisiesto. Los años bisiestos son aquellos que son divisibles por 4 pero no por 100 a menos que también lo sean por 400. Por ejemplo, 1900 no es bisiesto pero 2000 sí lo es.
7. Tres números positivos pueden ser la medida de los lados de un triángulo si y sólo si el mayor de ellos es menor que la suma de los otros dos. Definir una función `lados_triangulo :: (Float, Float, Float) → Bool` que devuelva True si los tres números que se le pasan verifican esta condición, y False en caso contrario.
8. Definir una función `es_rectangulo :: (Integer, Integer, Integer) → Bool` que devuelva True si los números que se le pasan pueden ser los lados de un triángulo rectángulo, y False en caso contrario. Sugerencia: una manera sería ordenar los tres números y verificar si el cuadrado del mayor de ellos es igual a la suma de los cuadrados de los otros dos. Sin embargo existe otra manera más fácil, utilizando (sólo una vez) la función `max_3 :: (Integer, Integer, Integer) → Integer` que devuelva el máximo de tres enteros. ¿Se te ocurre? Probar la función con las entradas (3,5,4), (5,13,12) y (7,3,5).

Funciones de orden superior

Las funciones en Haskell son "objetos de primera clase", es decir, se pueden almacenar en estructuras de datos, pasarlas como argumentos a otras funciones, y crear de esta forma nuevas funciones.

Una función se denomina de primer orden si sus argumentos y los resultados son valores de tipos de datos simples, y se llama de orden superior si tiene como argumento otra/s función/es o si se devuelve una función como resultado. Las funciones de orden superior hacen posible utilizar una variedad de potentes técnicas de programación.

Ejemplo, dada una función de enteros en enteros y un entero, escribir una función que aplique dos veces la función dada al argumento dado:

Tipo de esta función, **recibe una función y un entero:**

`aplicaDos :: (Integer → Integer) → Integer → Integer`

`aplicaDos f x = f (f x)`

- Probar esta función en Haskell.