Universidad de Huelva

Práctica 3

Definición de funciones

Índice

- Declaración de tipo
  - Como cualquier otra expresión en Haskell, el tipo de una función puede ser inferido automáticamente por el compilador o puede ser asignado explícitamente mediante el operador de coerción (::).
  - Por ejemplo, la función *media* que calcula el valor medio entre dos datos de tipo *Int* puede declararse como

```
media :: Int -> Int -> Int
```

- Declaración de tipo
  - Haskell admite polimorfismo en la definición de funciones. Esto quiere decir que la misma definición de función puede aplicarse sobre diferentes tipos de datos.
  - Para indicar un tipo de dato indefinido en una expresión de tipo se utilizan *variables de tipo*.
  - Por ejemplo, la función *primeroDeTres* que obtiene el primer elemento de cualquier 3-tupla se puede declarar así.

```
primeroDeTres :: (a,b,c)  ->  a
```

  - Esta declaración deja abierto el tipo de dato concreto de los tres elementos de la tupla.

- Declaración de tipo
  - Podemos indicar que una cierta variable de tipo debe ser de una determinada clase. Esto permite precisar más el tipo de dato dejando aún abierta la condición de polimorfismo. Para indicar las restricciones de una variable de tipo hay que utilizar la clausula (=>) antes de la expresión de tipo.
  - Por ejemplo, la función *media* se puede aplicar a números que soporten los operadores + y /. Esto corresponde a la clase *Fractional*. Por tanto, podemos definr el tipo de la función media como

```
media :: (Fractional  a) =>  a ->  a ->  a
```

  - Esta expresión puede leerse como "siendo *a* un tipo de dato *Fractional*, la función *media* toma dos datos de tipo *a* y devuelve un dato de tipo *a*".

- Una declaración de función tiene la siguiente sintaxis:

```
identificador   patrones  =  expresión
```

- El identificador se refiere al nombre de la función, que debe comenzar por una letra minúscula o _ .
- En su versión más simple, los patrones son variables asociadas a los posibles valores de los argumentos. Por ejemplo, la función *media* podemos declararla como

```
media  a  b  =  ( a + b ) / 2
```

- La aplicación de la función consiste simplemente en sustituir los valores de las variables en la expresión.

- La definición de una función puede dividirse en varias declaraciones.

```
identificador   patrones  =  expresión
identificador   patrones  =  expresión
...
```

- En este caso, al aplicar la función se estudia de manera consecutiva si los argumentos de la función verifican los patrones de alguna declaración (matching) y se sustituye la primera declaración que empareja correctamente los valores.
- Si no se logra ningún emparejamiento se produce un error en tiempo de ejecución.

- Por ejemplo, podríamos definir la longitud de una lista como.

```
myLength []  =  0
myLength x:xs  =  1 + (myLength xs)
```

- En este ejemplo, al aplicar la función *myLength* a una lista se estudia si se empareja con la lista vacía o si se empareja con una lista con un elemento de cabecera.

- Si una variable incluida en los patrones no aparece en la expresión podemos definirla como una variable "sin nombre" (*wild card*) utilizando el carácter '_'.

```
myLength []  =  0
myLength _:xs  =  1 + (myLength xs)
```

- Es bastante frecuente utilizar *wild cards* como última declaración de una función para cubrir cualquier caso no emparejado en declaraciones anteriores y evitar los errores en tiempo de ejecución.

- En muchas ocasiones, para definir un patrón no es posible utilizar tan solo variables y constantes sino que se necesita indicar las condiciones que deben cumplir las variables del patrón (*guards*).
- Para definir las condiciones de un patrón se utiliza la clausula ‘|’ que podemos leer como “tal que”.
- Por ejemplo, la función valor absoluto podemos definirla como

```
myAbs  x
        | x < 0   =   (- x)
        | x >= 0 = x
```

- El operador ‘|’ debe tener una sangría mayor que el comienzo de la función y si queremos incluir más de una condición todos los operadores ‘|’ deben tener la misma sangría.

Índice

- La sintaxis de las expresiones admite el uso de la instrucción "*if-then-else*".

```
IfSent    ::=     if exp [;] then exp [;] else exp
```

- Por ejemplo

```
myAbs  x = if x < 0 then (-x) else x
```

- Se pueden escribir las clausulas *then* y *else* en otra línea, pero el sangrado debe ser siempre mayor o igual a la posición del *if*.
- Las normas de estilo de los programadores de Haskell tienden a sustituir la instrucción *if-then-else* por el uso de guardas en las declaraciones.

- La instrucción *case-of* representa un condicional múltiple. La sintaxis es la siguiente:

CaseSent ::= case Exp of { Alts }

Alts ::= Alt_1 [;] Alt_2 [;] … [;] Alt_n

Alt ::= Patrón [Guardas] -> Exp

- Por ejemplo:

```
myNull :: [a] -> Bool
myNull list = case list of
                  [] -> True
                  x:xs -> False
```

- La clausula *where* se introduce al final de una expresión para indicar el valor de una variable o función utilizada en la expresión.
- Por ejemplo

```
myRadians  angle  =    myPi * angle / 180
                              where myPi = 3.141592653589793
```

- La clausula puede incluir varias definiciones entre llaves y separadas por ‘;’ o mediante sangrado del texto.

- La clausula let-in se introduce al comienzo de una expresión para indicar el valor de una variable o función utilizada en la expresión.
- Por ejemplo

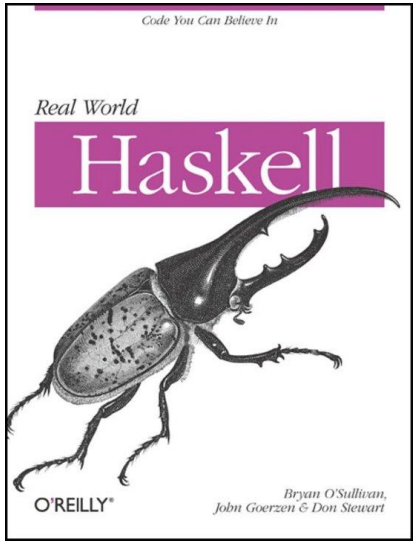
```
myRadians  angle  = let myPi = 3.141592653589793
                     in  myPi * angle / 180
```

- Se pueden definir varias variables o funciones dentro de la clausula l*et-in* utilizando definiciones entre llaves y separadas por ‘;’ o mediante sangrado.

- Ejercicios
  - Desarrollar la función **factorizar** que tome un valor entero y devuelva su factorización, es decir, la lista de factores en la que se puede descomponer el número. Por ejemplo, factorizar 60 = [5,3,2,2,1]
  - Desarrollar la función **esPrimo** que verifique si un número entero es primo.
  - Aplique la función anterior para obtener la secuencia de números primos entre el 1 y el 100.

Ejercicios:

- Probar en el intérprete de Haskelll los diferentes ejemplos incluidos en el capítulo "Defining Types, Streamlining Functions" del libro "Real World Haskell"



http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html