

SESIÓN 6

INTRODUCCIÓN WINHUGS



Universidad de Huelva

Currificación de funciones

```
Hugs> :t (+) 3
(3 +) :: Num a => a -> a
Hugs> :t (+) 3 4
3 + 4 :: Num a => a
```

Principio de inducción y recursividad

- 1) P es cierta para el n_0 (el elemento mas pequeño)
- 2) Si P es cierta para $n-1$ entonces puedo afirmar que puede ser cierta para n : $P(n-1) \rightarrow P(n)$

Listas intensionales

```
[(x,True) | x <- [1..20], even x, x < 15]
```

¿Qué vamos a ver?

Entrada y Salida

Main

Bloque Do y módulos

Acciones básicas

Ejercicios

6.3. ENTRADA Y SALIDA

Entrada y salida

Haskell es un lenguaje funcional puro. ¿qué significa esto?

- Una función siempre devuelve el mismo resultado. ¿Es la función `readFile "file"` una función pura?

Los procedimientos de entrada y salida no corresponden a funciones puras ya que el resultado de estos procedimientos depende del entorno de ejecución y **no siempre será el mismo**.

Por ejemplo, el procedimiento “leer carácter” debe devolver valores diferentes en cada ejecución, que es justo lo que no puede hacer una función pura.

¿Qué ocurre entonces?

Entrada y salida

Haskell utiliza un mecanismo especial para definir funciones de entrada/salida que consiste en introducir un **tipo de dato especial denominado IO**.

IO define un tipo de dato que se utiliza para representar y manejar operaciones de entrada y salida (Input/Output). Es una forma de indicar que una expresión produce un efecto colateral y que, por tanto, **no es una función pura**.

Para definir una expresión que provoca una acción de entrada/salida y que devuelve un String se utilizaría por ejemplo el tipo ***IO String***.

Entrada y salida

IO permite a Haskell gestionar estos efectos mientras mantiene la pureza del lenguaje.

Puede ser:

- pasado como argumento
- devuelto como salida de una función
- guardado en una estructura de datos,
- o combinado con otros valores IO

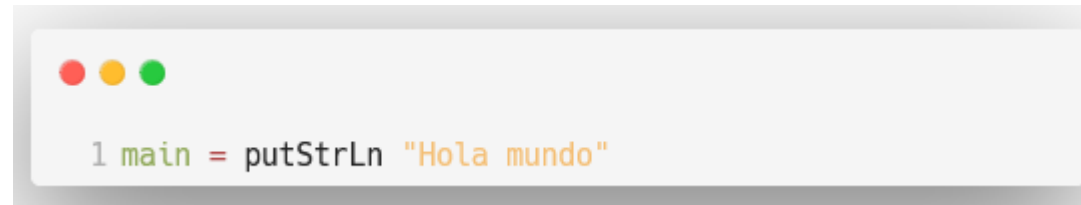
Entrada y salida: encadenamiento

Las acciones IO pueden encadenarse usando el operador `>>=` (llamado "bind") o usando la notación `do`, que permite secuenciar varias operaciones de entrada/salida.

¿pero ahora... cómo se ejecuta y usa un tipo IO?

Entrada y salida: main

El compilador Haskell busca un valor especial: `main :: IO()` que va a ser entregado al runtime y ejecutado (cuando tenemos un ejecutable. El siguiente programa escribiría en consola “Hola mundo”.



```
1 main = putStrLn "Hola mundo"
```

Si analizamos el tipo, obtendremos lo siguiente:

```
Main> :type main  
main :: IO ()
```

Esto quiere decir que la evaluación de `main` desarrollará una acción de entrada/salida y devolverá el **valor ()**.

Entrada y salida

Las acciones IO se ejecutan cuando son lanzadas desde un contexto de entrada/salida, es decir, no se ejecutan automáticamente.

La evaluación de una acción solo produce efectos de entrada/salida cuando se realiza desde otra acción o desde main.

La función main es una acción de entrada/salida que representa el efecto completo del programa.

¿Cómo se pueden ejecutar más acciones sino tenemos el objeto main?

6.3.2 ENTRADA Y SALIDA “DO”

Entrada y salida: tipos IO

Unit: (). No transmite ninguna información y tiene solo un constructor sin argumentos.

¿para qué sirve?

Lo necesitamos para ciertas concatenaciones de acciones. Es como el void en otros lenguajes y necesario porque Haskell siempre tiene que devolver algo (aunque no se para nada).

`putStrLn :: String -> IO()`. Acción que se ejecuta y devuelve ()

`getLine :: IO a`. Acción que produce un valor de tipo a.

Entrada y salida: bloque do o secuenciación.

Para poder ejecutar más de una acción y por lo tanto componer un “programa”, necesitamos algo más que el main.

El bloque **do** permite definir una acción compleja como una secuencia de acciones.

Líneas desnudas



```
1 main = do
2   putStrLn "Hola, Como te llamas?"
3   name <- getLine
4   putStrLn ("Ok " ++ name ++ ", encantado!")
```

Entrada y salida: bloque do o secuenciación

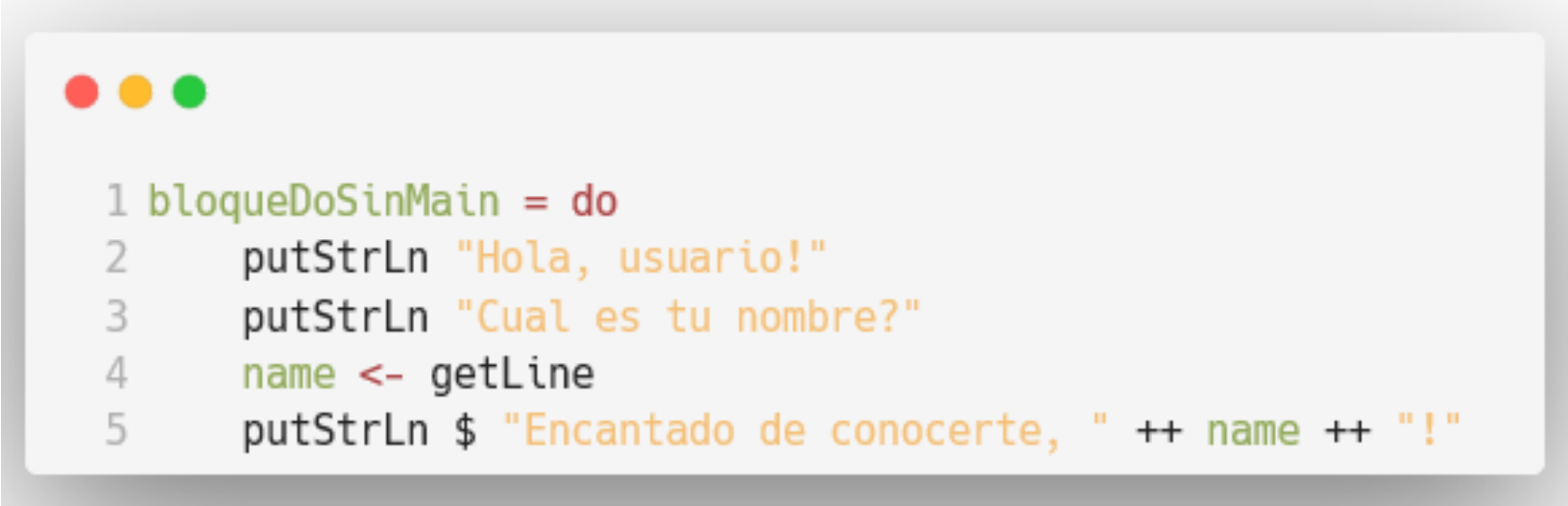
Para poder ejecutar más de una acción y por lo tanto componer un “programa”, necesitamos algo más que el main.

El bloque **do** permite definir una acción compleja como una secuencia de acciones.

```
Main> main
Hola, Como te llamas?
Antonio
Ok Antonio, encantado!
:: IO ()
```

Entrada y salida: bloque do

También podemos crear una secuencia de acciones que no esté asociada al elemento main o bloque principal del programa, para ello, haríamos lo siguiente:



```
1 bloqueDoSinMain = do
2   putStrLn "Hola, usuario!"
3   putStrLn "Cual es tu nombre?"
4   name <- getLine
5   putStrLn $ "Encantado de conocerte, " ++ name ++ "!"
```

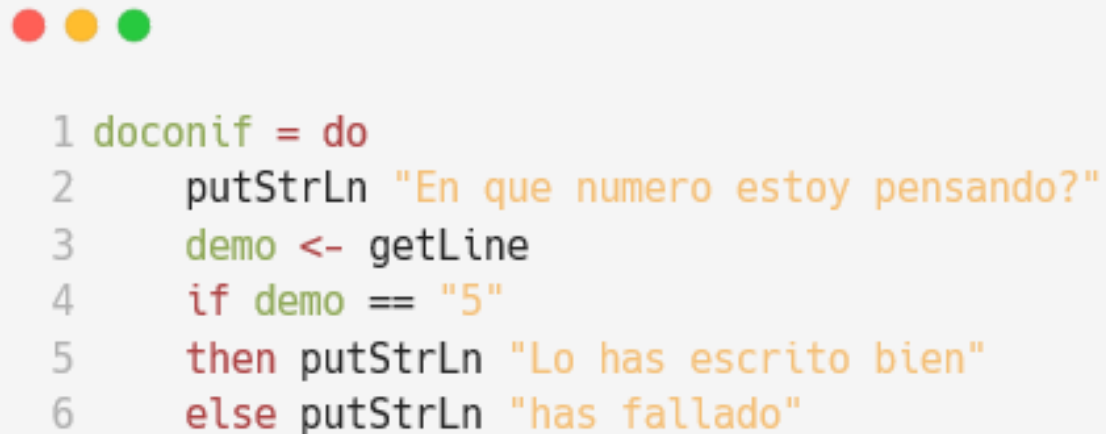
Entrada y salida: bloque do

También podemos crear una secuencia de accesiones que no esté asociada al elemento main o bloque principal del programa, para ello, haríamos lo siguiente:

```
Main> bloqueDoSinMain  
Hola, usuario!  
Cual es tu nombre?  
Antonio  
Encantado de conocerte, Antonio!  
:: IO ()
```


Entrada y salida: bloque do con if

Se pueden realizar combinaciones de bifurcaciones dentro del mismo bloque, y aplicar lo aprendido en la creación de funciones visto hasta ahora.



```
1 doconif = do
2   putStrLn "En que numero estoy pensando?"
3   demo <- getLine
4   if demo == "5"
5   then putStrLn "Lo has escrito bien"
6   else putStrLn "has fallado"
```

Entrada y salida: bloque do con if

Se pueden realizar combinaciones de bifurcaciones dentro del mismo bloque, y aplicar lo aprendido en la creación de funciones visto hasta ahora.

```
Main> doconif
```

```
En que numero estoy pensando?
```

```
6
```

```
has fallado
```

```
:: IO ()
```

```
Main> doconif
```

```
En que numero estoy pensando?
```

```
5
```

```
Lo has escrito bien
```

```
:: IO ()
```

¿Qué está pasando?

Entrada y salida: main infinito

Otro ejemplo:

```
Main> main
```

```
cadena uno
```

```
anedac onu
```

```
cadena dos
```

```
anedac sod
```

```
esto no para hasta que pulse intro  
otse on arap atsah euq eslup ortni
```

```
:: IO ()
```

```
1 main = do  
2     line <- getLine  
3     if null line  
4         then return()  
5     else do  
6         putStrLn (reverseWords line)  
7         main  
8  
9 reverseWords :: String -> String  
10 reverseWords = unwords . map reverse . words
```

Entrada y salida: getline, <- y let

Hemos visto que: la función *getline* es de tipo IO String. Esto significa que produce una acción de entrada/salida (en concreto la acción consiste en leer una línea de la entrada estándar) y devuelve un valor String.

Hemos visto que: la instrucción <- permite almacenar en una variable el valor devuelto por una acción. Podemos ver IO como un contenedor de valores y la instrucción <- como una forma de extraer el valor de ese contenedor.

En bloques do, = asigna directamente valores o expresiones, mientras que <- trabaja con valores envueltos en un contexto monádico.

Entrada y salida: getline, <- y let

```
1 main :: IO ()
2 main = do
3     nombre <- getLine           -- Extrae el valor de `IO String`
4     saludo = "Hola, " ++ nombre -- `=` solo asigna el resultado de la concatenación
5     putStrLn saludo
```

Entrada y salida: getline, <- y let

Los bloques do pueden contener también ligaduras formadas con la instrucción **let**.

En Haskell, **let** se utiliza para definir expresiones locales y asociarles nombres dentro de un bloque o un contexto.

A diferencia de <-, que trabaja con valores en un contexto monádico (como IO), **let** es puramente funcional y solo se usa para introducir nombres locales o hacer cálculos dentro de expresiones.

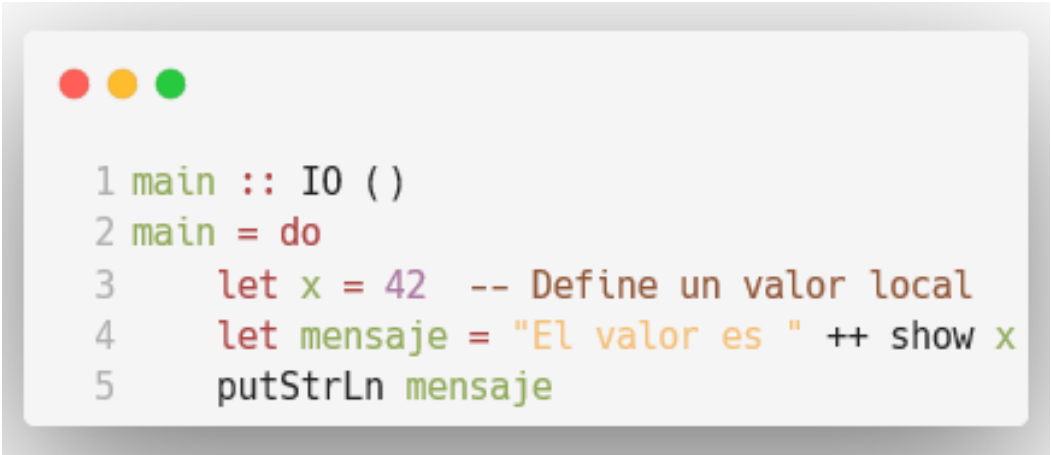
let nombre = expresión

Entrada y salida: getline, <- y let

Evalúa la expresión asociada en el momento en que se necesita (lazy evaluation).

Asigna el resultado al nombre.

El nombre solo es visible en el ámbito donde se definió.



```
1 main :: IO ()
2 main = do
3     let x = 42 -- Define un valor local
4     let mensaje = "El valor es " ++ show x
5     putStrLn mensaje
```

Entrada y salida: diferencias entre <- y let

let	<-
Define nombres para valores o expresiones.	Extrae valores de una mónada (IO, Maybe).
Usado en cualquier expresión, incluidas funciones puras.	Solo se usa dentro de un bloque do.
No interactúa con IO ni otro contexto monádico.	Está relacionado con operaciones monádicas.
No necesita bloques do para usarse.	Necesita bloques do o notación monódica.

Entrada y salida: ejemplo combinado

```
Main> main
Cual es tu nombre?
Antonio
Y tus apellidos?
Palanco Salguero
Ok ANTONIO PALANCO SALGUERO!
:: IO ()
```

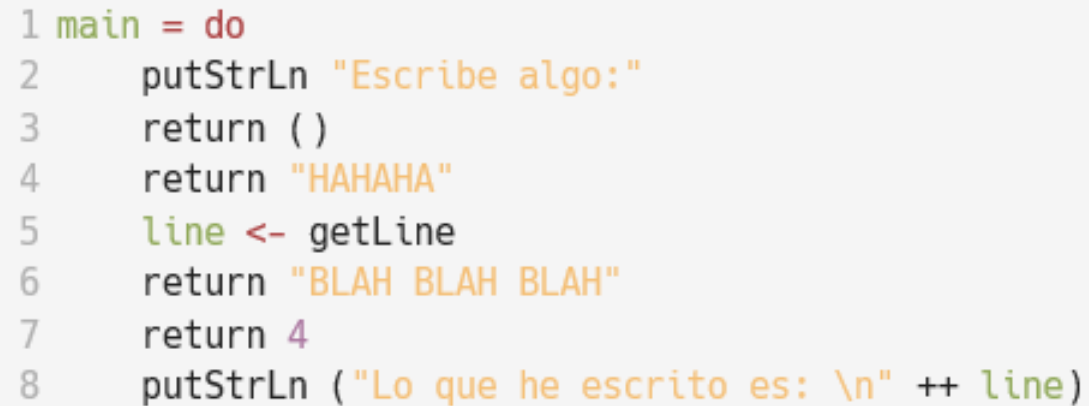


```
1 import Data.Char
2 main = do
3     putStrLn "Cual es tu nombre?"
4     firstName <- getLine
5     putStrLn "Y tus apellidos?"
6     lastName <- getLine
7     let bigFirstName = map toUpper firstName
8     let bigLastName = map toUpper lastName
9     putStrLn ("Ok " ++ bigFirstName ++ " " ++ bigLastName ++ "!" )
```

Entrada y salida: return

La instrucción return encapsula un valor dentro de una acción IO.

No supone ninguna ruptura de flujo. Puede entenderse como lo contrario de la instrucción <-.



```
1 main = do
2   putStrLn "Escribe algo:"
3   return ()
4   return "HAHAHA"
5   line <- getLine
6   return "BLAH BLAH BLAH"
7   return 4
8   putStrLn ("Lo que he escrito es: \n" ++ line)
```

Entrada y salida: return

La instrucción return encapsula un valor dentro de una acción IO.

No supone ninguna ruptura de flujo. Puede entenderse como lo contrario de la instrucción <-.

```
Main> main
```

```
Escribe algo:
```

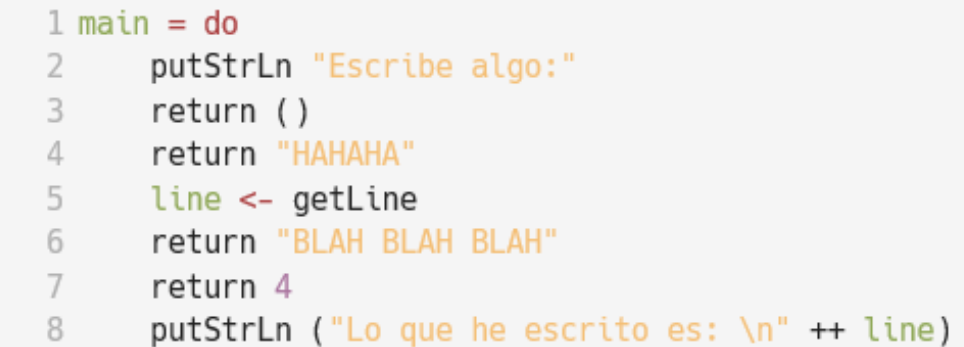
```
esto es lo que he escrito
```

```
Lo que he escrito es:
```

```
esto es lo que he escrito
```

```
:: IO ()
```

return no produce ningún efecto en la salida



```
1 main = do
2   putStrLn "Escribe algo:"
3   return ()
4   return "HAHAHA"
5   line <- getLine
6   return "BLAH BLAH BLAH"
7   return 4
8   putStrLn ("Lo que he escrito es: \n" ++ line)
```

Entrada y salida: \$

\$ es la función “aplicar a”.

$(\$)\ ::\ (a \rightarrow b) \rightarrow a \rightarrow b$ o lo que es lo mismo $f \$ x = f x$

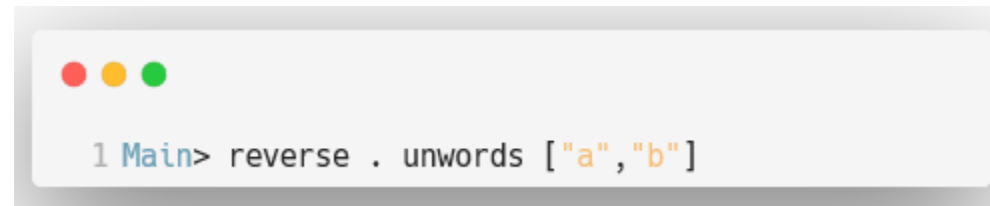
\$ es de baja precedencia, lo que permite escribir expresiones sin necesidad de anidar paréntesis, haciendo que las expresiones largas sean más fáciles de leer, especialmente cuando hay múltiples funciones aplicadas secuencialmente.

```
main = putStrLn (map toUpper (reverse "Haskell"))
```

```
main = putStrLn $ map toUpper $ reverse "Haskell"
```

Entrada y salida: \$

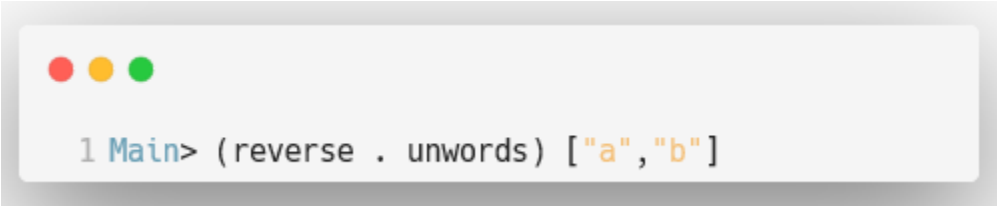
```
Main> reverse . unwords ["a","b"]  
ERROR - Type error in application  
*** Expression      : reverse . unwords ["a","b"]  
*** Term            : unwords ["a","b"]  
*** Type            : [Char]  
*** Does not match  : a -> b
```



Entrada y salida: \$

```
Main> (reverse . unwords) ["a", "b"]
```

```
"b a" :: [Char]
```



```
1 Main> (reverse . unwords) ["a", "b"]
```

```
Main> reverse $ unwords ["a", "b"]
```

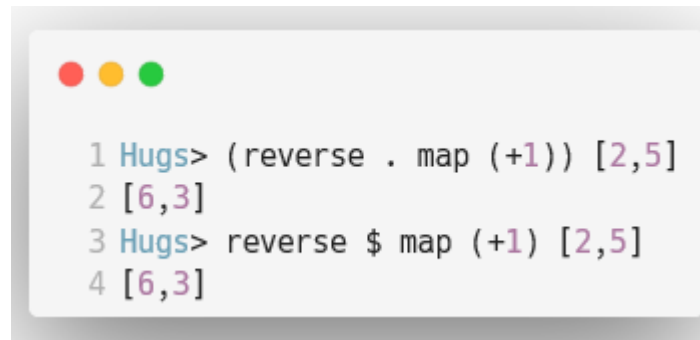
```
"b a" :: [Char]
```



```
1 Main> reverse $ unwords ["a", "b"]
```

Entrada y salida: \$

```
Hugs> reverse . map (+1) [2,2]
ERROR - Type error in application
*** Expression      : reverse . map (flip (+) 1) [2,2]
*** Term            : map (flip (+) 1) [2,2]
*** Type            : [c]
*** Does not match : a -> b
```



```
1 Hugs> (reverse . map (+1)) [2,5]
2 [6,3]
3 Hugs> reverse $ map (+1) [2,5]
4 [6,3]
```

6.3.3 E/S “ACCIONES”

Entrada y salida: acciones

putStrLn :: String -> IO ()

Toma una cadena y la muestra en la salida estándar con un salto de línea final.

putStr :: String -> IO ()

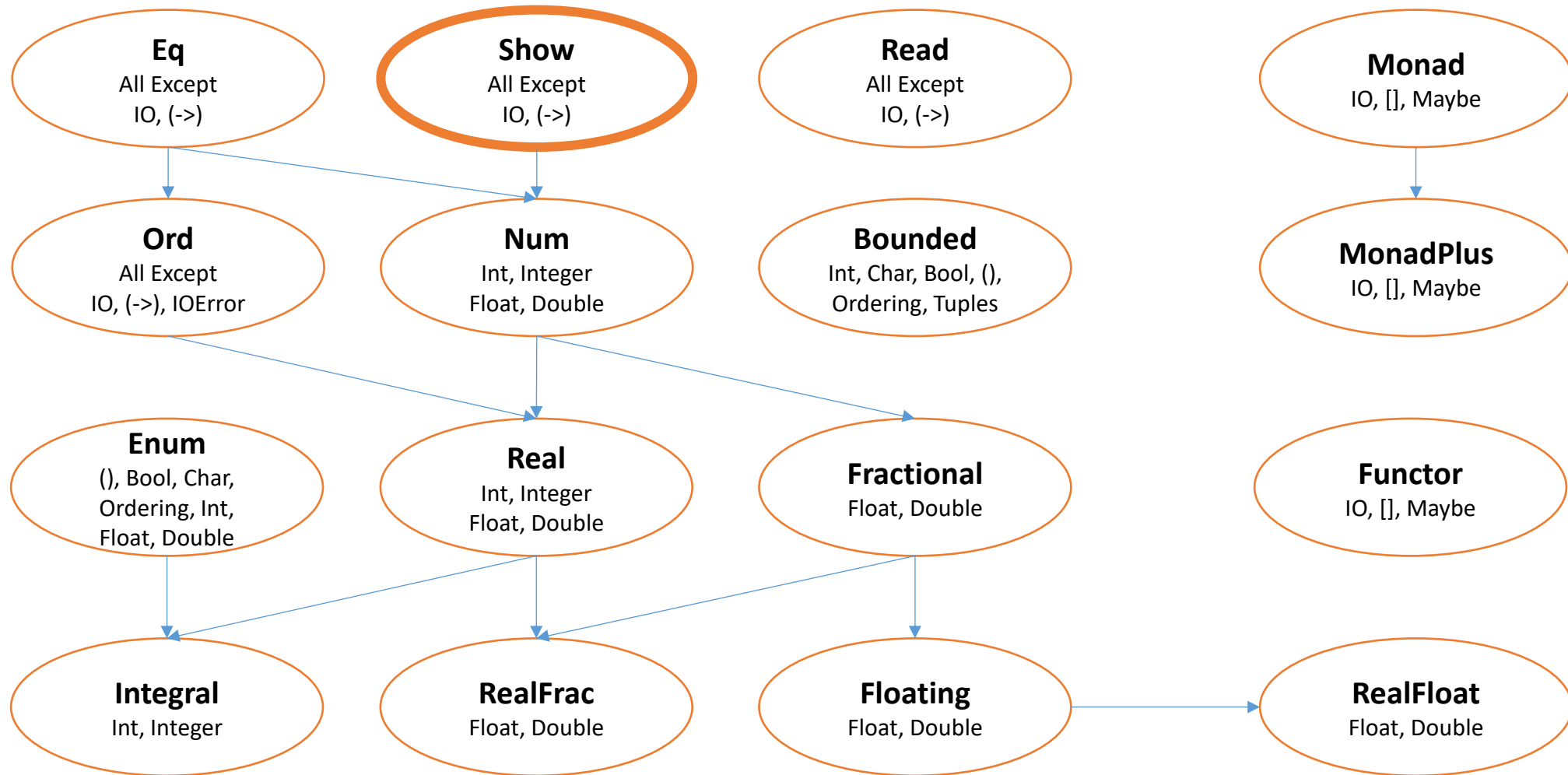
Similar a putStrLn sin el salto de línea final.

putChar :: Char -> IO ()

Muestra un carácter en la salida estándar

print :: Show a => a -> IO ()

Muestra cualquier valor de un tipo que desarrolle Show



Entrada y salida: acciones

`getChar :: IO Char`

Lee un carácter de la entrada estándar. Debido al buffering **la lectura no es efectiva hasta que no se vuelca el buffer, es decir, hasta que no se introduce un salto de línea en la consola.**

`getLine :: IO String`

Lee una línea completa hasta el salto de línea

`getContents :: IO String`

Lee la entrada estándar completa. Si se la entrada estándar se ha redirigido desde un fichero, la acción lee el fichero completo. Si la entrada estándar es la consola la acción lee la entrada indefinidamente hasta interrumpir la ejecución.

Entrada y salida: acciones

La evaluación perezosa se sigue manteniendo con las acciones. Esto quiere decir que una acción no se ejecuta hasta que no necesita ser evaluada y solo si la ejecución se encuentra en un contexto de entrada/salida.

```
> head [ print "hola", print 5, print 'C']  
"hola"  
it :: ()
```

Se ha creado una lista con tres acciones. La función head obtiene la primera acción. El intérprete debe evaluar la acción por lo que provoca que se escriba "hola" en la consola

6.4 MODULOS

Modulos

Es una colección de funciones, tipos y clases de tipos relacionadas entre sí.

Para importar un módulo debemos hacerlo: *import nombre_modulo*

Ejemplo: *módulo para trabajar con listas. Nub (elimina duplicados).*

```
1 import Data.List
2 numUniques :: (Eq a) => [a] -> Int
3 numUniques = length . nub
```

```
import Data.List
import Data.Char
```

```
--nub $ map (\p -> map toLower p) ["hola", "HOLA", "hOLA"]
```

Módulos

¿Y si tenemos funciones con el mismo nombre que algunas del módulo?

Podemos cargar algunas funciones

```
import Data.List (nub, sort)
```

O todo el módulo menos algunas funciones

```
import Data.List hiding (nub)
```

Módulos: creación de módulos propios y submódulos

Crearemos un fichero con el nombre del módulo: NombreDelModulo.hs

declaramos con ***module NombreModulo*** las funciones y tipos a exponer y definiremos las funciones.

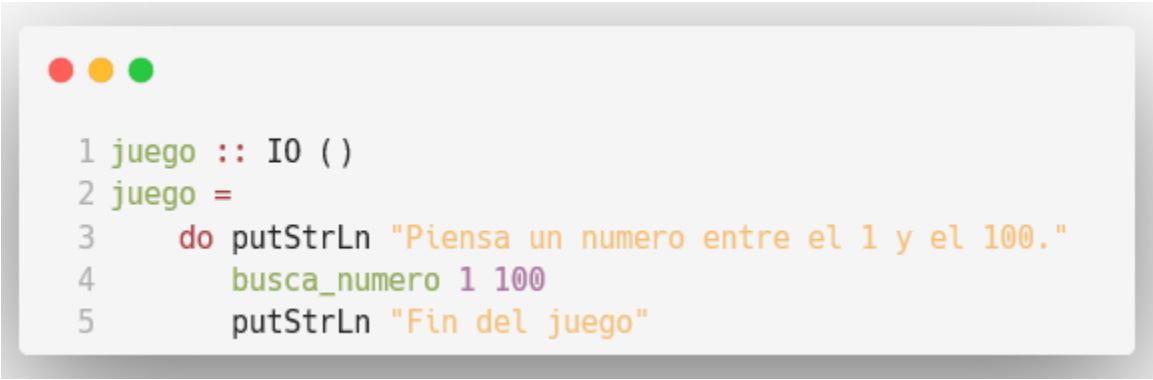
```
module MiModulo
( funcion1
, funcion2
, funcion3
, funcion4
) where

module MiModulo.SubModulo
( funcion5
, funcion6
) where
funcion5 :: Float -> Float
funcion5 a = loquesea
funcion6 :: Float -> Float
funcion6 a = loquesea
```


6.5 EJERCICIOS

Ejercicio 1. Búsqueda secuencial

Teniendo en cuenta la siguiente implementación, definir la función busca número para encontrar el número buscado. El programa preguntará por pantalla si el valor 1 es el correcto y se responderá SI o NO En caso de ser el número, el programa termina, en caso contrario, se sumará 1 y se sigue.

A screenshot of a code editor window with a light gray background and a title bar with three colored circles (red, yellow, green). The code is written in Haskell and is as follows:

```
1 juego :: IO ()
2 juego =
3     do putStrLn "Piensa un numero entre el 1 y el 100."
4       busca_numero 1 100
5       putStrLn "Fin del juego"
```

Ejercicio 1. Búsqueda secuencial

Nota: para mostrar por pantalla el valor que devuelve una función se utiliza show

Nota: Los números se buscarán en orden desde el 1 al 100.

Nota: cuando se introduzca un valor diferente a SI o NO mostrará error advirtiéndolo y sigue

```
1 juego :: IO ()
2 juego =
3     do putStrLn "Piensa un numero entre el 1 y el 100."
4         busca_numero 1 100
5         putStrLn "Fin del juego"
6
7 busca_numero :: Int -> Int -> IO ()
8 busca_numero a b =
9     do putStr ("Es " ++ show proximo ++ " el numero ? [SI/NO] ")
10        s <- getLine
11        case s of
12            "NO"    -> busca_numero (proximo+1) b
13            "SI"    -> return ()
14            _       -> do putStr ("Error en la entrada, respuesta SI o NO: Es " ++ show proximo ++ " el numero ? [SI/NO] ")
15                       busca_numero a b
16 where
17     proximo = a
```

Ejercicio 1. Otras soluciones.

```

1 --juego::IO()
2 juego = do
3   putStrLn "Piensa un n entre el 1 y el 100"
4   buscan 1 100
5   putStrLn "Fin"
6
7 --buscan :: Int->Int->IO()
8 buscan a b = do
9   putStrLn ("es tu numero "++(show a))
10  if a==b+1 then putStrLn "sacabao"
11  else do
12    linea <- getLine
13    if linea == "si" then putStrLn "ta weno"
14    else if linea=="no" then buscan (a+1) b
15    else do putStrLn "pon un si o un no"
16          buscan a b

```

```

1 juego :: IO ()
2 juego = do
3   putStrLn "Piensa un numero entre el 1 y el 100."
4   busca_numero 1 100
5   putStrLn "Fin del juego."
6
7 busca_numero :: Int -> Int -> IO ()
8 busca_numero ini fin = do
9   if ini == fin then
10    return ()
11   else do
12     putStrLn ("Tu numero es el " ++ show ini ++ "?")
13     line <- getLine
14     if line == "SI" || line == "NO"
15     then
16       if line == "SI"
17       then putStrLn ("Tu numero era el " ++ show ini)
18       else busca_numero (ini+1) fin
19     else do
20       putStrLn "Introduce solo 'SI' o 'NO'"
21       busca_numero ini fin

```

Práctica 2:

Realizar un programa que le pedirá al jugador que ingrese el límite inferior y el límite superior del rango en el que desea que el programa adivine su número. El jugador debe pensar en un número dentro de este rango definido. El programa intentará adivinar el número que el jugador ha pensado usando una búsqueda binaria. Después de cada intento, el jugador debe responder con "encontrado", "mayor", o "menor". Si el jugador responde "mayor", el programa ajustará el rango para hacer su próxima suposición en el rango superior. Si el jugador responde "menor", el programa ajustará el rango para hacer su próxima suposición en el rango inferior. El programa continuará hasta que el jugador responda "encontrado".