PRÁCTICA 7

INTRODUCCIÓN WINHUGS

Universidad de Huelva

7.1. REPASO

7.1. Repaso

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 n0 (el elemento mas pequeño)

2) Si P es cierta para n-1 entonces puedo afirmar que puede ser cierta para n: P(n-1) -> P(n)

Listas intensionales

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

7.2. EJERCICIOS PROPUESTOS S5

7.2. Ejercicios

Listas intensionales

1. `[11,12,13,14,15,16,17,18,19,20]`
   1) `Hugs> [x+10|x<-[1..10]]`
2. `[[2],[4],[6],[8],[10]]`
   2) `Hugs> [[x]|x<-[1..10],even x]`
3. `[[10],[9],[8],[7],[6],[5],[4],[3],[2],[1]]`
   3) `Hugs> [[11-x]|x<-[1..10]]`
   3) `Hugs> [[11-x] | x<-[1..10], f<-[(11-)]]` **???? Tiene efecto la función**
   3) `Hugs> [[f x] | x<-[1..10], f<-[(11-)]]`
4. `[True,False,True,False,True, False,True,False,True,False]`
   4) `Hugs> [(mod x 2) /= 0|x<-[1..10]]`
   4) `Hugs> [odd x | x <- [1..10]]`
   4) `Hugs> [`**`f x`**` | x <- [1..10],` **`f<-[odd]`**`]`

7.2. Ejercicios

Listas intensionales

5. `[(3,True),(6,True),(9,True),(12,False),(15,False), (18,False)]`

```
5) Hugs> [(3*x,3*x<12)| x <- [1..10], x<7]
5) Hugs> [(3*x, x<=3) | x <- [1..10], x<=6]
```

6. `[(5,False),(10,True),(15,False),(40,False)]`

```
6) Hugs> [(x*5,x*5==10)| x <- [1..10], x<4  || x == 8]
6) Hugs> [(5*x, False) | x <- [1..10], x<=3 || x == 8]
```

7. `[(11,12),(13,14),(15,16),(17,18),(19,20)]`

```
7) Hugs> [(10+x,11+x)|x<-[1..10], mod x 2 /= 0]
7) Hugs> [(x+1,x+2)|x<-[10..18],even x]
```

8. `[[5,6,7],[5,6,7,8,9],[5,6,7,8,9,10, 11],[5,6,7,8,9,10,11,12,13]]`

```
8) Hugs> [map (+4) (take x [1..10])|x<-[1..10],odd x, x>1]
8) Hugs> [[5..x*2+1]|x<-[1..10],x>2 && x<7]
```

7.2. Ejercicios

Listas intensionales

9. `[21,16,11,6,1]`

```
9) Hugs> [(50-(5*x)) +1 | x<-[1..10], x >5 ]
9) Hugs> [((11-x)*5 - 29) | x<-[1..10], x < 6]
9) Hugs> [ 5*(11-x)-4 | x <- [1..10], x>5]
9) Hugs> [(10 - x) * 5 + 1 | x <- [1 .. 10], x >= 6]
```

10. `[[4],[6,4],[8,6,4],[10,8,6,4],[12,10,8,6,4]]`

```
10) Hugs> [[x*2,x*2-2..4]|x<-[1..10], x>1 && x <7]
10) Hugs> [[2 * x, 2 * (x - 1) .. 4] | x <- [1 .. 10], x >= 2 && x <= 6]
10) Hugs> [[(x+2),x .. 4] | x <- [1..10], even x]
10) Hugs> [[x+2,x..4]|x<-[1..10], even x]
10) Hugs> [[(f x),(f x-2)..x] | x<-[1..10], f<-[(+0),(+2),(+4),(+6),(+8)],  x==4]
```

Índice

¿Qué vamos a ver?

Entrada y Salida

Main

Bloque Do y módulos

Acciones básicas

Ejercicios

7.3. ENTRADA Y SALIDA

7.3.1. Entrada y salida

Entrada y salida

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

- Una función siempre devuelve el mismo resultado

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?

7.3.1. Entrada y salida

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 cuya ejecución produce una acción de entrada/salida. 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 el tipo ***IO String***.

7.3.1. Entrada y salida

Entrada y salida

Si definimos:

```
t :: huevo_frito
```

¿qué tenemos?

```
t1 :: como_hacer_huevo_frito
```

¿y ahora?

Tenemos la definición de algo que no tiene “efecto” sobre nada, pero tenemos las instrucciones de como hacerlo.

Del mismo modo, un valor de **tipo IO** son las instrucciones para producir algún valor de tipo a (cualquier clase)

7.3.1. Entrada y salida

Entrada y salida

El tipo IO pude ser:

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

¿Cómo se ejecuta un tipo IO?

7.3.1. Entrada y salida “main”

Entrada y salida: main

El compilador Haskell busca un valor especial: main :: IO() que va a ser entregado al runtime y ejecutado

El siguiente programa escribe en consola “Hola mundo”

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

Si comprobamos el tipo de dato de la función main obtenemos:

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

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

7.3.1. Entrada y salida “main”

Entrada y salida

Las acciones IO se ejecutan cuando son lanzadas desde un contexto de entrada/salida.

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?

7.3.2 ENTRADA Y SALIDA “DO”

7.3.2. 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 Haskel 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. Solo se puede ejecutar dentro de un bloque de acciones.

7.3.2. Entrada y salida “do”

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

7.3.2. Entrada y salida “do”

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 = do
    {
        putStrLn "Hola, como te llamas?";
        name <- getLine;
        putStrLn ("ok " ++ name ++ ", encantado!")
    }
```

7.3.2. Entrada y salida “do”

Entrada y salida: bloque do

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 ()
```

7.3.2. Entrada y salida “do”

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:

```
accionesSinMain = do
  putStrLn "Hola, usuario!"
  putStrLn "Cual es tu nombre?"
  name <- getLine
  putStrLn $ "Encantado de conocerte, " ++ name ++ "!"
```

7.3.2. Entrada y salida “do”

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

7.3.2. Entrada y salida “do”

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.

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

7.3.2. Entrada y salida “do”

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?

7.3.2. Entrada y salida “do”

Entrada y salida: bloque do con if

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 ()
```

```haskell
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn (reverseWords line)
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

7.3.3. Entrada y salida “let”

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.

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

7.3.3. Entrada y salida “let”

Entrada y salida: getline, <- y let

```
import Data.Char
main = do
    putStrLn "Cual es tu nombre?"
    firstName <- getLine
    putStrLn "Y tus apellidos?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
    let bigLastName = map toUpper lastName
    putStrLn ("Ok " ++ bigFirstName ++ " " ++ bigLastName ++ "!")
```

7.3.3. Entrada y salida “let”

Entrada y salida: getline, <- y let

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

7.3.3. Entrada y salida “return”

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 = do
    putStrLn "Escribe algo:"
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn ("Lo que he escrito es: \n" ++ line)
```

7.3.3. Entrada y salida “return”

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

7.3.3 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.

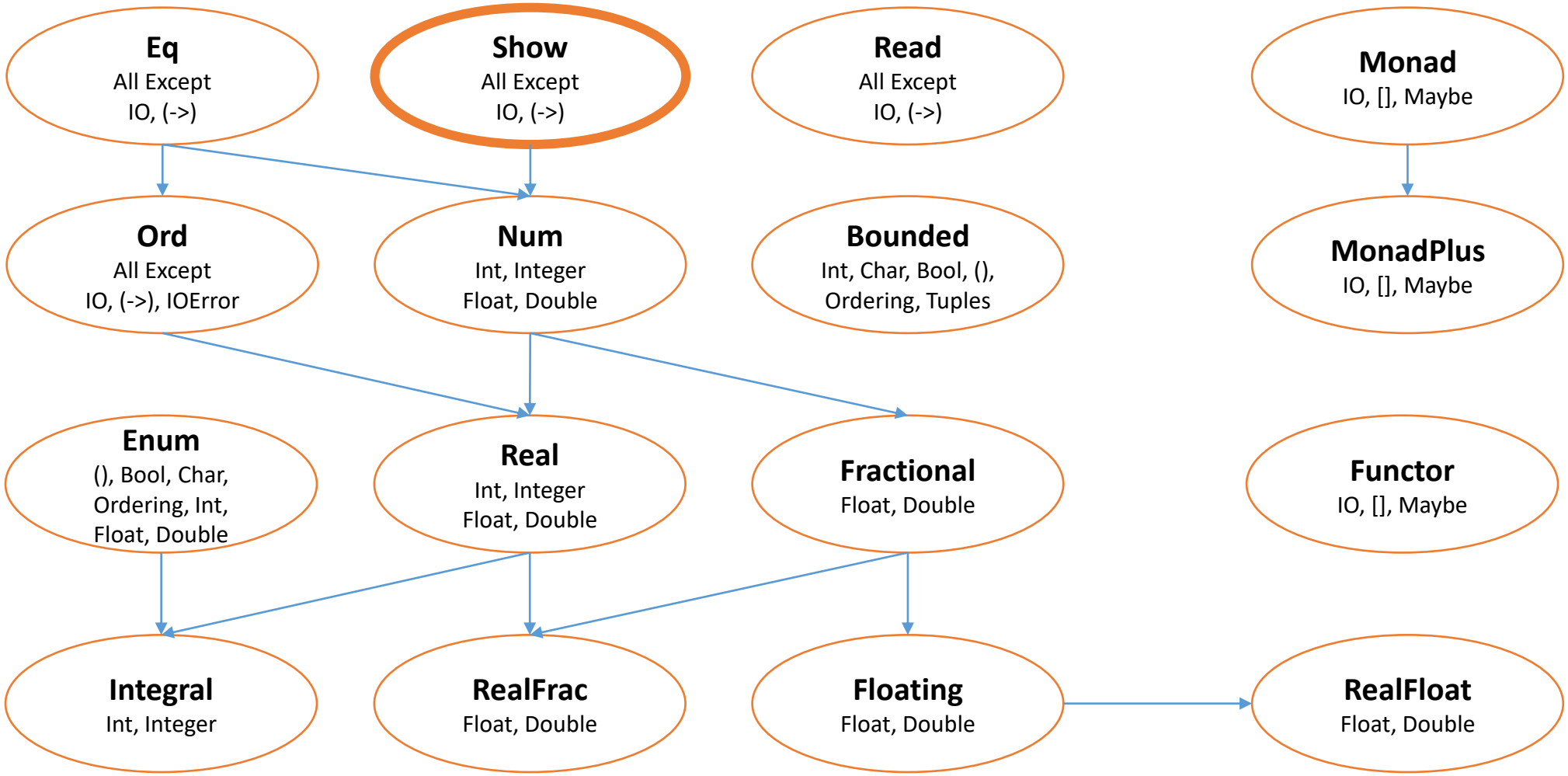
putChar :: Char -> IO ()

Muestra un carácter en la salida estándar

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

Muestra cualquier valor de un tipo que desarrolle Show

6.3.4. Entrada y salida: acciones



7.3.4. Entrada y salida: acciones

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.

7.3.4. Entrada y salida: acciones

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

7.4 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)
```

7.4. Módulos

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
```

7.5 EJERCIOS

7.5.1. Entrada y salida: acciones

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.

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

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

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

7.5.1. Entrada y salida: ejercicio1

Ejercicio 1. Búsqueda secuencial

Teniendo en cuenta la siguiente implementación, definir la función busca_número para encontrar el número buscado. E NO En caso de ser el n

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

busca_numero :: Int -> Int -> IO ()
```

Nota: para

Nota: Los

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

7.5.1. Entrada y salida: ejercicio1

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

busca_numero :: Int -> Int -> IO ()
busca_numero a b =
    do putStr ("Es " ++ show proximo ++ " el numero ? [SI/NO] ")
       s <- getLine
       case s of
         "NO"    -> busca_numero (proximo+1) b
         "SI"    -> return ()
         _       -> do putStr ("Error en la entrada, responsa SI o NO: Es " ++ show proximo ++ " el numero ? [SI/NO] ")
                       busca_numero a b
    where
      proximo = a
```

7.5.1. Entrada y salida: ejercicio1

Ejercicio 1. Búsqueda secuencial

Solución aportada por Alberto Rodero

```
--ALBERTO RODERO

--juego::IO()
juego = do
    putStrLn "Piensa un n entre el 1 y el 100"
    buscan 1 100
    putStrLn "Fin"

--buscan :: Int->Int->IO()
buscan a b = do
    putStrLn ("es tu numero "++(show a))
    if a==b+1 then putStrLn "sacabao"
    else do
        linea <- getLine
        if linea == "si" then putStrLn "ta weno"
        else if linea=="no" then buscan (a+1) b
        else do putStrLn "pon un si o un no"
            buscan a b
```

7.5.1. Entrada y salida: ejercicio1

Ejercicio 1. Búsqueda secuencial

Solución aportada por Jesús Valeo

```
--JESUS VALEO

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

busca_numero :: Int -> Int -> IO ()
busca_numero ini fin = do
    if ini == fin then
        return ()
    else do
        putStrLn ("Tu numero es el " ++ show ini ++ "?")
        line <- getLine
        if line == "SI" || line == "NO"
        then
            if line == "SI"
            then putStrLn ("Tu numero era el " ++ show ini)
            else busca_numero (ini+1) fin
        else do
            putStrLn "Introduce solo 'SI' o 'NO'"
            busca_numero ini fin
```

7.5.2. Entrada y salida: Práctica 3

Práctica 3: Implementación del juego mejorada.

El programa le pide al jugador humano que piense un número entre 1 y 100 y tratara de acertar el número que ha pensado preguntando al jugador. El jugador responderá encontrado, mayor o menor y en función de la respuesta, se realizara una modificación del número buscado mejorando el ejercicio que vimos en clase que realizaba una búsqueda secuencial. Esta modificación se realizará calculando el nuevo número de la siguiente forma:

proximo = (x+y) div 2

El programa finalizará su ejecución cuando el número pensado por el jugador haya sido encontrado.