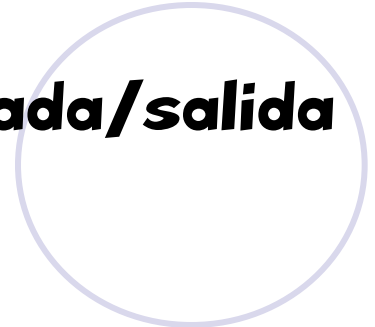


Acciones de entrada/salida





Acciones de entrada/salida



- Una **acción de entrada/salida** es una función que cambia el estado y además produce un resultado

data **IO a** \approx **World** \rightarrow **(a, World)**

- Una función f que toma como dato un caracter produce un efecto de entrada/salida y un resultado de tipo entero

$f :: \text{Char} \rightarrow \text{IO Int}$

$-- :: \text{Char} \rightarrow \text{World} \rightarrow (\text{Int}, \text{World})$



El tipo de datos predefinido IO a

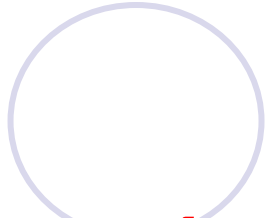
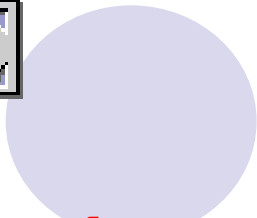


- IO a acciones de entrada/salida con resultado de tipo a
- IO () acciones de entrada/salida con resultado ()
- Acciones básicas (primitivas) de entrada/sálida :

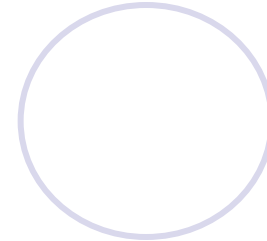
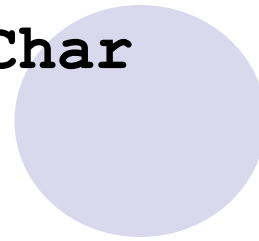
1. `getChar :: IO Char`

2. `putChar :: Char -> IO ()`

3. `return :: a -> IO a`



getChar



getChar :: IO Char

-- Lee el carácter introducido por el teclado,
-- lo produce como resultado

```
Prelude> getChar
a :: IO Char

Prelude> getChar

:: IO Char
```

Este es el carácter que se ha tecleado,

Aquí se ha pulsado la tecla correspondiente a '`\n`',



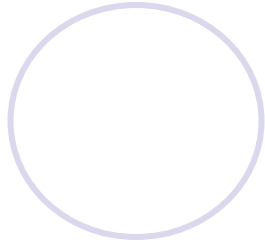
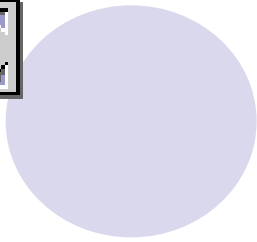
putChar

putChar :: Char -> IO ()

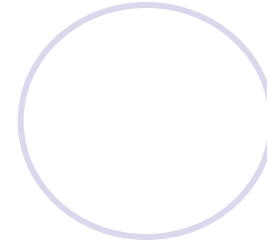
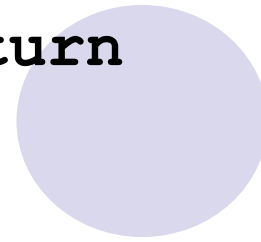
-- escribe un carácter y produce la tupla
-- vacía () como resultado.

```
Prelude> 'a'  
'a' :: Char  
  
Prelude> putChar 'a'  
a :: IO ()
```

```
Prelude> '\n'  
'\n' :: Char  
  
Prelude> putChar '\n'  
:: IO ()
```



return



return :: a -> IO a

-- :: a -> World -> (a, World)

-- (return e) produce el resultado que
-- resulta de evaluar e,
-- sin cambiar el mundo (sin interacción).

-- return valor ≈ \world -> (valor, world)

No tiene sentido evaluar (return exp) en Hugs interactivo.



Componiendo acciones primitivas



```
echo :: IO ()
```

```
echo = getChar >>= (\c -> putChar c)
```

```
echo' :: IO ()
```

```
echo' = getChar >>=
      (\c -> putChar '\n' >>=
        (\_ -> putChar c))
```

```
echoR :: IO Char
```

```
echoR = getChar >>=
      (\c -> (putChar c >>=
        \_ -> return c))
```

```
Main> echo
aa :: IO ()
```

```
Main> echo'
a
a :: IO ()
```

```
Main> echoR
aa :: IO Char
```



Composición secuencial de dos acciones: $>>=$



$$a1 \gg= \lambda x \rightarrow a2$$

Acción que resulta de realizar **a1** seguida de **a2** sobre el resultado de **a1**

La lambda sirve para dar nombre (**x**) al resultado de **a1** y así poder usarlo en **a2**.



Definición del operador $\gg=$



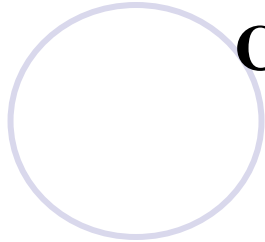
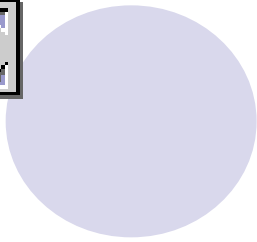
`infixl 1 >>=`

$(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

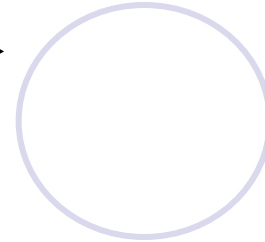
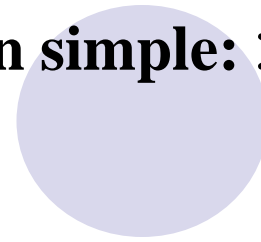
-- definida como primitiva, equivalente a

```
f >>= g = \w -> let (v,w') = f w
                  in g v w'
```

```
f >>= g = (uncurry g) . f
```



Composición simple: >>

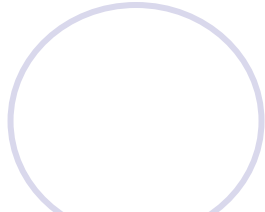


- Cuando la segunda acción “no utiliza” el resultado de la primera
- Se define en función de >>=

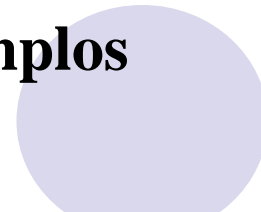
```
infixl 1 >>
```

```
(>>) :: IO a -> IO b -> IO b
```

```
a1 >> a2 = a1 >>= \ _ -> a2
```



Ejemplos



- `echo' :: IO ()`
`echo' = getChar >>=`
`(\c -> putChar '\n' >> putChar c)`

- `echoR :: IO Char`
`echoR = getChar >>=`
`(\c -> (putChar c >> return c))`

- `Main> getChar >> getChar >> getChar`
`ywz :: IO Char`

¿Cuál de los tres caracteres es el resultado de la acción?



Componer k acciones: **do**



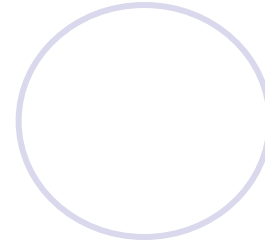
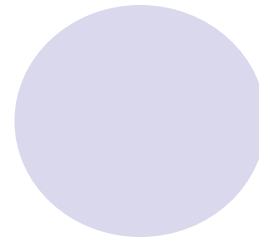
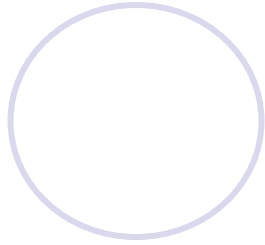
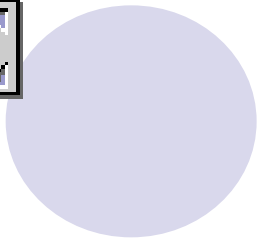
• do
 accion₁
 accion₂
 ⋮
 accion_k

donde accion_i ::= exp | p <- expr
 exp :: IO a
 p es un patrón

1. do e ≡ e

2. do
 e
 exps

≡ e >> do exps



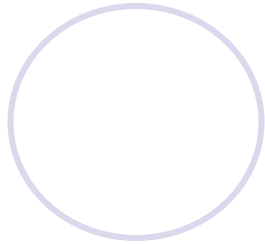
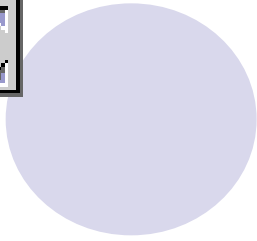
$$\left. \begin{array}{l} 3. \text{ do} \\ p \leftarrow e \\ \text{exps} \end{array} \right\} \equiv e \gg= \backslash p \rightarrow \text{do exps}$$

- Si la acción $e :: \text{IO } a$ produce un resultado, entonces $p \leftarrow e$ es la acción que realiza e y deja en p el resultado de la acción $e :: \text{IO } a$.
- La última acción de un `do` no puede ser de tipo $p \leftarrow e$

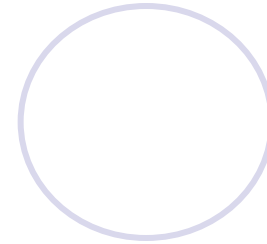
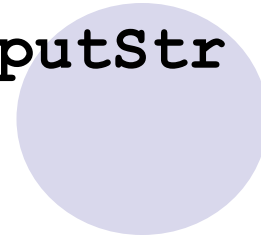


Ejemplos

- `putStr :: String -> IO ()` -- predefinida
`putStr (c:cs) = do`
 `putChar c`
 `putStr cs`
- `putStrLn :: String -> IO ()` -- predefinida
`putStrLn s = do`
 `putStr s`
 `putChar '\n'`
- `echo' :: IO ()`
`echo' = do` `c <- getChar`
 `putChar '\n'`
 `putChar c`



Uso de `putStr`



```
Prelude> "Esto \n es un salto de linea"  
"Esto \n es un salto de linea" :: String
```

```
Prelude> putStrLn "Esto \n es un salto de linea"  
Esto  
    es un salto de linea  
:: IO ()
```

```
Prelude> putStr "Esto \t es un tabulador"  
Esto      es un tabulador :: IO ()
```



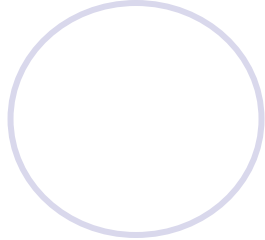
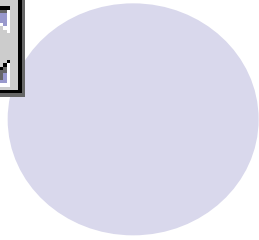
La función predefinida `getLine`



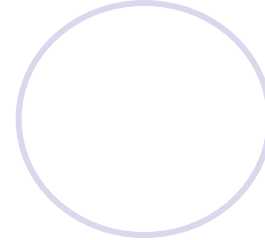
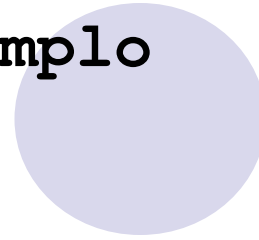
`getLine :: IO String`

-- lee el string de entrada hasta '\n'
-- está definida (primitiva) equivalente a

```
getLine =  
    do  
    c <- getChar  
    if c=='\n' then return ""  
        else do  
            cs <- getLine  
            return (c:cs)
```

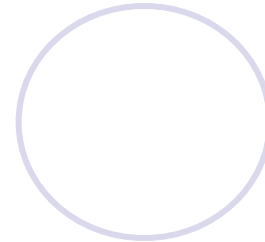
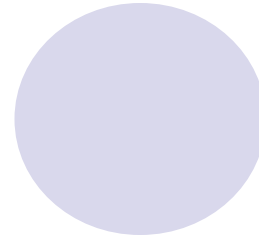
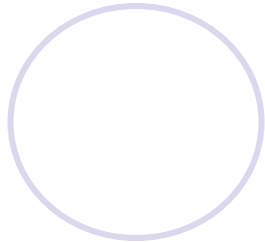
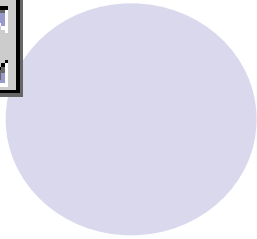


Ejemplo



```
pedirLeer :: IO String
-- pide y produce como resultado el string que se teclee
pedirLeer = do
    putStrLn "Teclee password + Enter"
    getLine

coincide :: String -> IO Bool
-- (coincide pass) lee un string y decide si es
-- es idéntico a pass
coincide pass = do
    tecleado <- pedirLeer
    return (tecleado == pass)
```



```
pedirComprobar :: String -> IO ()
-- (pedirComprobar p) pide al usuario un caracter,
--- lo lee y escribe un mensaje indicando si coincide
-- con p
```

```
pedirComprobar p =
  do
  sonIg <- coincide p
  putchar '\n'
  if sonIg then putStrLn "Correcto"
  else putStrLn "Incorrecto"
```



Las funciones predefinidas `print` y `read`



- `print :: Show a => a -> IO ()`
`print = putStrLn . show`
- `read :: Read a => String -> a`

```
Main> print 5
5 :: IO ()
```

```
Main> read "False" :: Bool
False :: Bool
Main> read "Tru" :: Bool
Program error: Prelude.read: no parse
Main> read "[1,2,3]" :: [Int]
[1,2,3] :: [Int]
```



Ejemplo read and `print`

```
leerLisEnt :: IO [Int]
```

```
leerLisEnt = do  
    lin <- getLine  
    return (read lin)
```

```
mapLisEnt :: Show a => (Int -> a) -> IO ()
```

```
mapLisEnt f = do  
    l<-leerLisEnt  
    print (map f l)
```

```
Main> mapLisEnt (*2)  
[1,2,3]  
[2,4,6]  
:: IO ()
```



Un juego interactivo



- *Juego para dos jugadores:*
 - *El jugador 1 elige un número **clave** y teclea **juego clave**.*
 - *El jugador 2 tiene que adivinar la **clave** a base de introducir números por el teclado y que el programa le conteste si la clave es menor, mayor o igual que ellos.*
 - *El juego termina cuando jugador 2 introduce el número **clave**.*

```
Adivina un número entre 1 y 100
45
Mi número es menor que 45
12
Mi número es mayor que 12
30
Lo adivinaste es: 30
```

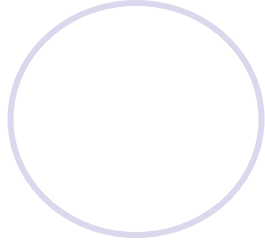
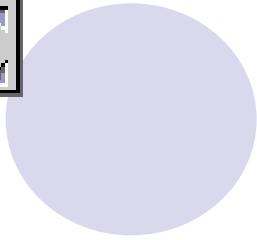


juego y limpiar pantalla

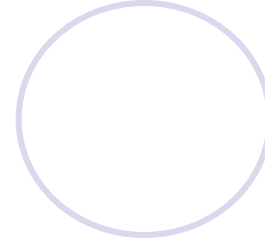
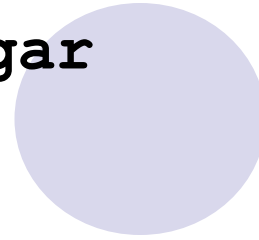


```
juego :: Int -> IO ()
-- Comprueba que la clave esté entre 1 y 100.
-- En caso afirmativo, inicia el juego y lo gestiona
juego clave =
    if not (elem clave [1..100])
    then putStrLn "La clave debe estar entre 1 y 100"
    else    do
        limpiarPantalla
        putStrLn "Adivina un número entre 1 y 100"
        n <- getLine
        jugar clave (read n)

limpiarPantalla :: IO ()
limpiarPantalla = putStr ['\n' | i<-[1..24]]]
```



jugar



```
jugar :: Int -> Int -> IO ()
-- gestiona el juego del jugador 2
jugar clave n =
  if not (elem n [1..100])
  then putStrLn "Debe estar entre 1 y 100"
  else if n == clave
    then putStrLn ("Lo adivinaste es: "++ show n)
    else do
      n' <- pedirNuevoNum n clave
      jugar clave n'
```



pedirNuevoNum



```
pedirNuevoNum :: Int -> Int -> IO Int
-- (pedir n clave) informa al usuario sobre
-- el número anterior y le pide uno nuevo
pedirNuevoNum n clave =
  let
    info True = "menor"
    info False = "mayor"
  in do
    putStrLn ("Mi número es " ++
              info (clave<n) ++
              " que " ++ show n)
    s <- getLine
    return (read s)
```



Ficheros de entrada/salida



```
readFile :: FilePath -> IO String
```

```
-- lee y da como resultado un string
```

```
writeFile :: FilePath -> String -> IO ()
```

```
-- escribe el string en el fichero
```

```
appendFile :: FilePath -> String -> IO ()
```

```
-- añade el string al final del fichero
```

donde

- `type FilePath = String`
usando / (en lugar de \) para describir el camino
- los ficheros tiene que tener extensión `txt`
- los ficheros de escritura son creados (si no existen ya)



Ejemplo: Convertir a mayúsculas un fichero de texto



```
convertirMay =  
  do  
  putStrLn "Teclee el nombre del fichero  
           que quiere convertir a mayúsculas"  
  ficheroIn <- getLine  
  s <- readFile ficheroIn  
  putStrLn "Teclee el nombre del fichero donde quiere  
           dejar el resultado"  
  ficheroOut <- getLine  
  writeFile  ficheroOut  (map toUpper s)  
  putStrLn (ficheroIn ++ " ha sido convertido a  
           mayúsculas en " ++ ficheroOut)
```



Versión “en el mismo fichero”



```
seguidoMay =  
  do  
  putStrLn "Teclee el nombre del fichero"  
  fichero <- getLine  
  s <- readFile fichero  
  appendFile fichero ("\n\n\t -- EL TEXTO  
                        ANTERIOR EN MAYÚSCULAS  
                        ES: -- \n\n"  
                    ++ (map toUpper s))  
  putStrLn ("COMPLETADA OPERACIÓN EN " ++  
            fichero)
```