Universidad de Huelva

Práctica 5

Entrada y salida

Índice

- Haskell es un lenguaje funcional puro. Esto quiere decir que las funciones definidas en Haskell son funciones puras en sentido matemático.
- 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.

- 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 devuelve un String y que provoca una acción de entrada/salida se utilizaría el tipo

IO String

- Para definir una expresión que provoca una acción de entrada/salida y no devuelve nada se utilizaría el tipo

IO ()

- El tipo () se denomina ***unit*** y podemos interpretarlo como una tupla sin elementos. Por tanto es una forma de expresar "nada". Es el equivalente en Haskell del tipo de dato *void* utilizado en otros lenguajes.

- El siguiente programa escribe en consola “Hola mundo”

```
main = putStrLn  “Hola mundo”
```

- Si comprobamos el tipo de dato de la función *main* obtenemos

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

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

- 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.

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

```
main =  do
     putStrLn "Hello, what's your name?"
     name <- getLine
     putStrLn ("Hey " ++ name ++ ", you rock!")
```

- En este caso la función *main* realiza una secuencia de tres acciones en la que primero se escribe un mensaje, a continuación se lee una línea de la consola y por último se vuelve a escribir un mensaje.

- La sintaxis del bloque puede utilizar llaves y punto y coma como separadores.

```
main =  do
 {
     putStrLn "Hello, what's your name?" ;
     name <- getLine ;
     putStrLn ("Hey " ++ name ++ ", you rock!")
 }
```

- El resultado de un bloque *do* es el resultado de la última acción del bloque.

- 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*.
- 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*, que en este caso no requiere la palabra clave *in*.

```
import Data.Char
main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn ("hey " ++ bigFirstName ++ " " ++ bigLastName ++ "!")
```

- 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
   return ()
   return "HAHAHA"
   line <- getLine
   return "BLAH BLAH BLAH"
   return 4
   putStrLn line
```

- La instrucción *if* también puede utilizarse sobre acciones.

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

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

Acciones sobre la entrada y salida estándar

- **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

Acciones sobre la entrada y salida estándar

- **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.

- 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.
- Por ejemplo,

```
> 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.

Funciones que operan sobre acciones

- **when :: Bool -> IO () -> IO ()**
  - Está incluida en el módulo Control.Monad. El primer argumento es una condición. Si la condición es cierta se devuelve el segundo argumento, si no se ejecuta "return ()". En realidad el segundo argumento debe ser un instancia de la clase Applicative, aunque normalmente se utiliza con IO.
- **unless :: Bool -> IO () -> IO ()**
  - Es semejante a when, pero verifica que la condición sea falsa.

Funciones que operan sobre acciones

- **sequence :: [IO a] -> IO [a]**
  - Toma una lista de acciones devuelve una acción que realiza todas esas acciones y obtiene como resultado una lista de los resultados de estas acciones. Suele utilizarse junto a *map*.

```
> sequence (map print [1,2,3])
1
2
3
[(),(),()]
```

Funciones que operan sobre acciones

- **mapM :: (a -> IO b) -> [a] -> IO [b]**
  - Mapea una acción sobre una lista. Es equivalente a secuenciar el resultado de mapear una acción sobre una lista.

```
> mapM print [1,2,3]
1
2
3
[(),(),()]
```

- **mapM_ :: (a -> IO b) -> [a] -> IO ()**
  - Es equivalente a *mapM* pero desechando el resultado

Funciones que operan sobre acciones

- **forever :: IO a -> IO b**
  - Definida en el módulo Control.Monad. Toma una acción y la repite indefinidamente, es decir, genera una secuencia infinita repitiendo la acción.
- **interact :: (String -> String) -> IO ()**
  - Toma una función que transforma cadenas y la aplica a todo el contenido de la entrada estándar, volcando el resultado sobre la salida estándar.

- Las funciones que hemos visto hasta ahora realizan acciones sobre la entrada y salida estándar.
- Las acciones sobre ficheros son similares, pero necesitan un manejador de fichero (*handle*) para indicar el direccionamiento de la acción.

```
import System.IO
main = do
    handle <- openFile "girlfriend.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

- **openFile :: FilePath-> IOMode -> IO Handle**
  - Acción que abre un fichero. Está definida en el módulo *System.IO*. El primer argumento es la ruta de acceso al fichero (*FilePath* es un sinónimo de *String*). El segundo argumento es el modo de apertura del fichero, que puede ser *ReadMode*, *WriteMode*, *AppendMode* o *ReadWriteMode*. El resultado es el manejador del fichero.

- **hGetContents :: Handle -> IO String**
  - Obtiene el contenido completo de un fichero. Está definida en el módulo *System.IO*. Hay que tener en cuenta que la evaluación perezosa provoca que el contenido no se lea realmente hasta que no sea necesario.
- **hClose :: Handle -> IO ()**
  - Cierra un fichero. Está definida en el módulo *System.IO*. Es responsabilidad del programador el cierre de los ficheros que se abren mediante *openFile*.

- **withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r**
  - Encadena las acciones de abrir fichero, tratarlo y cerrarlo. El primer argumento es la ruta del fichero. El segundo argumento es el modo de apertura. El tercer argumento es la acción a realizar sobre el fichero.

- **hGetChar :: Handle -> IO Char**
  - Lee un carácter del fichero.
- **hGetLine :: Handle -> IO String**
  - Lee una línea del fichero.
- **hPutChar :: Handle -> Char -> IO ()**
  - Escribe un carácter en un fichero.
- **hPutStr :: Handle -> String -> IO ()**
  - Escribe una línea en un fichero.
- **hPutStrLn :: Handle -> String -> IO ()**
  - Escribe una línea en un fichero terminada en un salto de línea.

- **readFile :: FilePath -> IO String**
  - Abre un fichero y obtiene una acción que lee su contenido. Podemos obtener este contenido con la instrucción <-. Cuando se utiliza esta función no se obtiene el manejador así que el ciere del fichero es responsabilidad de Haskell.
- **writeFile :: FilePath -> String -> IO ()**
  - Abre un fichero en modo escritura y genera una acción de escribir el segundo argumento en el fichero.
- **appendFile :: FilePath -> String -> IO ()**
  - Abre un fichero en modo añadir y genera una acción de escribir el segundo argumento en el fichero.

- El acceso a los ficheros se realiza por medio de buffers. Esto quiere decir que las lecturas no se hacen efectivas hasta que el buffer está lleno.
- Por defecto el tamaño del buffer es una línea, es decir, el salto de línea provoca un volcado del buffer.
- En los ficheros binarios el tamaño del buffer suele ser un bloque de 64Kb.

- **hSetBuffering :: Handle -> BufferMode -> IO ()**
  - Asigna el tamaño del buffer. Los valores de *BufferMode* pueden ser *NoBuffering*, *LineBuffering* o *BlockBuffering* (*Maybe Int*). En el último caso, si se indica *BlockBuffering Nothing* se asigna el valor por defecto del sistema operativo. Para asignar un tamaño concreto en bytes se utilizaría *BlockBuffering Just tamaño*.
- **hFlush :: Handle -> IO ()**
  - Provoca el volcado del buffer.

- Las funciones asociadas al sistema de archivos se encuentran en el módulo *System.Directory*

```
Prelude> import System.Directory
Prelude  System.Directory> :browse System.Directory
canonicalizePath :: FilePath -> IO FilePath
copyFile :: FilePath -> FilePath -> IO ()
copyFileWithMetadata :: FilePath -> FilePath -> IO ()
copyPermissions :: FilePath -> FilePath -> IO ()
createDirectory :: FilePath -> IO ()
createDirectoryIfMissing :: Bool -> FilePath -> IO ()
createDirectoryLink :: FilePath -> FilePath -> IO ()
createFileLink :: FilePath -> FilePath -> IO ()
...
```

- Puede obtenerse una descripción más exahustiva del contenido del módulo System.IO en la descripción oficial de Haskell 2010.

https://www.haskell.org/onlinereport/haskell2010/haskellch41.html#x49-32000041

- Ver capítulo 9 del manual "*Aprende Haskell por el bien de todos*".

http://aprendehaskell.es/content/ClasesDeTipos.html