Entrada y Salida en Haskell

Pablo López
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

25 de marzo de 2011

1. Transparencia referencial, efectos laterales y acciones

Haskell es un lenguaje funcional puro; esto es, libre de efectos laterales. El valor devuelto por una función depende exclusivamente de sus argumentos, nunca de un estado —local o global— donde pueda quedar reflejada una historia de la ejecución. En un lenguaje puro, siempre podemos asegurar que `f x == f x`.

Gracias a esta pureza, Haskell goza de **transparencia referencial**: una expresión denota siempre el mismo valor, por lo que puede ser reemplazada por su valor o viceversa sin alterar el resultado final el programa. De la misma forma, una expresión puede reemplazarse por cualquier reducción parcial de la misma sin alterar el resultado del programa. Como mucho estas sustituciones afectarían a la eficiencia del programa; pero nunca al resultado obtenido.

La transparencia referencial facilita el razonamiento sobre los programas y permite transformarlos en otros más legibles o eficientes por simple manipulación algebraica. Por ejemplo, si tenemos las funciones

```
f :: Integer -> Integer
g :: Integer -> Integer
```

entonces podemos reemplazar la expresión

```
f (g x) + f (g x)
```

por su equivalente (pero más eficiente)

```
2 * f (g x)
```

Otra consecuencia de la transparencia referencial es que las subexpresiones que aparecen varias veces en una expresión se pueden evaluar una sola vez en una definición local introducida con un `where` o un `let ... in`.

Las funciones de entrada/salida, sin embargo, provocan **efectos laterales**. Cada vez que se ejecutan afectan al estado de un *buffer* o a la posición de un puntero de lectura o escritura, de forma que cada ejecución puede devolver un resultado diferente. Por lo tanto, las funciones de entrada/salida no son puras y sacrifican la transparencia referencial. Una «función» Haskell como:

```
leerEntero :: Integer
```

no sería pura, pues debería ser capaz de devolver valores diferentes para cada invocación. No sería posible reemplazar:

```
leerEntero + leerEntero
```

por

```
2 * leerEntero
```

Sin embargo, si `leerEntero` fuese una función pura, se comportaría como una función constante, por lo que no tendría utilidad alguna.

¿Es posible reconciliar la pureza con los efectos laterales intrínsecos de la entrada/salida? En efecto, podemos utilizar el potente sistema de tipos de Haskell para asignar a la función `leerEntero` un tipo más complejo que permita preservar la pureza de la función. Si un valor del tipo `World` representa el estado del mundo, entonces podemos declarar `leerEntero` de la siguiente manera:

```
leerEntero :: World -> (Integer, World)
```

Es decir, la función `leerEntero` toma el estado actual del mundo, y devuelve un entero y un nuevo estado del mundo. El cambio de estado del mundo es producto del efecto lateral de la operación de entrada que se acaba de ejecutar.

Resultaría demasiado tedioso escribir explícitamente el estado del mundo en los tipos de las funciones de entrada/salida. Para evitarlo, introducimos el siguiente sinónimo de tipo parametrizado:

```
type IO a = World -> (a, World)
```

Un valor de tipo `IO a` es una función de entrada/salida que modifica el estado del mundo y devuelve, como resultado de esa modificación, un resultado de tipo `a`. A este tipo de funciones las llamaremos **acciones de entrada/salida**. En general, usaremos dos tipos para las acciones de entrada/salida:[1]

Tipo	Significado
`IO a`	acción de **entrada** que devuelve un resultado de tipo `a`
`IO ()`	acción de **salida** que no devuelve resultado alguno

Ahora podemos declarar `leerEntero` como:

```
leerEntero :: IO Integer
```

Según esta definición, `leerEntero` es una función pura que devuelve una constante: la acción de entrada que lee un entero.

La anterior definición del tipo `IO` es meramente conceptual. Obviamente, el estado del mundo no se almacena explícitamente en un argumento de tipo `World` —no es necesario— y, de hecho, ni siquiera existe el tipo `World`. El estado del mundo está implícito en cualquier cómputo; simplemente se trata del contenido almacenado en el *buffer* de teclado, de la información mostrada en la pantalla, del contenido almacenado en los ficheros, etc.

Cuando una función devuelve un resultado de tipo `IO`, lo que esto realmente indica es que se trata de una función en cuyo interior ocurren efectos laterales que no son visibles desde el exterior. Para entender esto, comparemos las declaraciones de función:

[1] `()` es el tipo *unit*, el tipo que corresponde a las tuplas vacías. Es un tipo similar al tipo `void` en otros lenguajes de programación.

```
pura :: String -> Integer
impura :: String -> IO Integer
```

En el caso de la función `pura`, el `Integer` devuelto es calculado a partir exclusivamente del argumento de tipo `String`: se trata de un **valor puro** obtenido a través de un **cómputo**. Por el contrario, el uso del tipo `IO` en la función `impura` nos indica que, para calcular el `Integer` devuelto, además del argumento `String`, se ha tenido en cuenta el estado del mundo: se trata de un **valor impuro** obtenido a través de una **acción**. De esta manera tan simple, el sistema de tipos de Haskell consigue diferenciar valores puros, obtenidos a través de cómputos, de valores impuros, obtenidos a través de acciones. La disciplina de tipos de Haskell mantiene ambos mundos separados: un valor impuro siempre aparece encapsulado dentro del tipo `IO`. Cuando se escribe un programa Haskell, es muy importante estructurarlo en un conjunto de funciones puras, que se encargan de la parte computacional, y otro conjunto de acciones, que se encargan de la interacción con el mundo exterior.

2. Acciones de entrada/salida básicas

Haskell predefine un buen número de acciones de entrada/salida sobre caracteres, cadenas, etc. De entre ellas destacamos las siguientes.

Sobre caracteres:

```
getChar :: IO Char
putChar :: IO Char
```

Sobre cadenas:

```
getLine :: IO String
putStr :: IO ()
putStrLn :: IO ()
```

Sobre cualquier tipo imprimible (instancia de `Show`) o legible (instancia de `Read`):[2]

```
print :: Show a => a -> IO a
readLn :: Read a => IO a
```

Sobre ficheros:

```
type FilePath = String

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

El sinónimo de tipo `FilePath` almacena la ruta del fichero al que queremos acceder. Las rutas se pueden escribir siempre con el separador de directorios de UNIX (`/`). La función `writeFile` crea el fichero si no existe, o borra el contenido si existía; mientras que `appendFile` añade información al final de un fichero (si no existe, lo crea).

[2] La función `readLn` eleva una excepción si se produce un error sintáctico.

3. Acciones de entrada/salida compuestas

En GHCi, es posible evaluar las siguientes expresiones:

```
putChar 'a'

putStrLn "hola mundo"

print (1+2+3)

readFile "ejemplo.txt"

getChar

getLine
```

Sin embargo, no es posible evaluar la siguiente expresión:

```
(putStr "hola ", putStrLn "mundo")
```

Esto se debe a que el orden de evaluación no está preestablecido en Haskell. De hecho, por ser un lenguaje puro, el orden de evaluación no puede afectar al resultado. Si embargo, en la expresión anterior el orden de evaluación es relevante: no es lo mismo escribir `"hola mundo\n"` que `"mundo\nhola "`.

Por otro lado, la siguiente expresión también es incorrecta:

```
putChar getChar
```

En este caso no se trata de un problema de orden de evaluación, sino de un problema con los tipos. Observa que `putChar` espera un `Char` como argumento, pero le pasamos una acción `IO Char`. Para poder imprimir el carácter leído necesitaríamos una función que accediera al valor impuro `a` almacenado en la acción `IO a`; por ejemplo:

```
value :: IO a -> a
```

entonces podríamos reescribir nuestro ejemplo como:

```
putChar (value getChar)
```

Sin embargo, no existe la función `value` ni ninguna que se le parezca; es decir, no tenemos ninguna función que nos permita acceder al valor impuro devuelto por una acción. El motivo es que una función como `value` sacrificaría la transparencia referencial; por ejemplo, la siguiente expresión:

```
value getChar == value getChar
```

no sería cierta. Una función como `value` permitiría convertir un valor impuro en un valor puro, lo que traicionaría la clasificación de los valores en puros e impuros. La idea clave es que un resultado que se ha obtenido de forma impura a través de una acción siempre permanecerá impuro, encapsulado dentro del tipo `IO`. En Haskell, una vez que un valor se ha «ensuciado», no hay forma de «limpiarlo».

Las acciones de entrada/salida tienen que ser combinadas a través de un operador que:

1. indique explícitamente el **orden de evaluación**, y

2. permita el **acceso** a los valores impuros.

Este operador es >>=, pronunciado *bind* o *then*, y su tipo es:

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

Como vemos, el operador >>= es capaz de combinar dos acciones de entrada/salida de manera que se ejecutan en secuencia: primero la acción denotada por el primer argumento (`IO a`) y después la acción denotada por el segundo (`a -> IO b`). Además, observa que la segunda acción recibe como argumento el resultado de la primera. De esta manera, **dentro** del operador >>= es posible acceder al valor devuelto por una acción de entrada/salida; sin embargo, el valor devuelto por >>= sigue siendo una acción de entrada/salida, por lo que resulta inaccesible desde el exterior. Podemos pensar que el tipo `IO a` es un tipo abstracto de datos: sabemos que contiene un valor impuro de tipo `a`, pero no tenemos acceso a él. El operador >>= es una operación del tipo abstracto `IO`, por lo que tiene acceso a los valores impuros encapsulados.

Así, las anteriores acciones compuestas pueden escribirse usando el operador >>= de la siguiente manera:

```
putStr "hola " >>= \ _ -> putStrLn "mundo"   -- orden

getChar >>= \ x -> putChar x                 -- orden y acceso
```

Ambas acciones se pueden simplificar:

```
putStr "hola " >>= const (putStrLn "mundo")  -- orden

getChar >>=  putChar                         -- orden y acceso
```

Por supuesto, es posible secuenciar un número arbitrario de acciones. Por ejemplo, la siguiente acción compuesta lee tres caracteres:

```
getChar >>= \ x -> getChar >> \ y -> getChar
```

El problema es que una combinación como la anterior sólo devuelve el resultado de la última acción realizada (el último `getChar`). ¿Cómo es posible devolver el resultado de las tres acciones? Para ello, el tipo `IO` define una acción cuyo objetivo es simplemente devolver resultados; la acción `return`, cuyo tipo es:

```
return :: a -> IO a
```

es decir, `return` toma un resultado de cualquier tipo y devuelve una acción que devuelve precisamente ese resultado. Esto significa que podemos transformar el resultado de un cómputo puro `a` en una acción impura `IO a`; es decir, podemos «ensuciar» un valor. Recuerda que el paso inverso, de impuro a puro (`IO a -> a`), no está disponible.

La acción que lee tres caracteres y los devuelve como resultado se puede definir de la siguiente manera:

```
getChar >>= \ x -> getChar >> \ y -> getChar >> = \ z -> return (x, y, z)
```

donde `return` se limita a agrupar los resultados anteriores. El tipo definitivo de la acción es `IO (Char, Char, Char)`.Una forma un poco más legible de escribir la anterior acción compuesta es:

```
tres :: IO (Char, Char, Char)
tres = getChar >>= \ x ->
       getChar >>= \ y ->
       getChar >>= \ z ->
       return (x, y, z)
```

Por supuesto, `return` no tiene por qué agrupar todos los resultados en una tupla: puede agrupar solo algunos de ellos en cualquier otra estructura de datos.

Ejercicio 1. Define alternativas a la función `tres` que devuelvan:

- el primer y último caracteres leídos, almacenados en una tupla
- una cadena con los tres caracteres en el orden de lectura
- la suma de los ordinales de los tres caracteres leídos

Indica explícitamente el tipo de las tres funciones.

Ejercicio 2. Define una función `saludo` que solicite al usuario su nombre por teclado y escriba un saludo personalizado en pantalla.

Ejercicio 3. Escribe una función `factorial` que pida un entero por teclado y escriba en pantalla su factorial.

Ejercicio 4. Escribe una función recursiva que lea con `getChar` una cadena de caracteres y escriba en pantalla su inversa con `putChar`.

4. La notación `do`

Aunque el operador `>>=` es muy potente, su uso es bastante engorroso, pues es necesario introducir varias λ-expresiones de forma explícita. Para mejorar la legibilidad, Haskell introduce la notación `do`: se trata de azúcar sintáctico para una secuencia de acciones. La anterior acción compuesta se escribiría en notación `do`:

```
doTres = do
           x <- getChar
           y <- getChar
           z <- getChar
           return (x, y, z)
```

La sintaxis de `do` es muy simple; dentro de un `do` aparece una secuencia de:

- **generadores** de la forma:

  patrón `<-` *expresión de tipo* `IO a`

- **acciones** `return` de la forma:

`return` *expresión de cualquier tipo*

- **definiciones locales** de la forma:

`let` *patrón* = *expresión de cualquier tipo*

Hay varios detalles a tener en cuenta:

1. Tras el `do` se aplica la regla de **sangrado**, luego todas las acciones deben tener el mismo margen izquierdo.

2. Podemos evitar la regla de sangrado mediante llaves y puntos y coma; por ejemplo:

```
doTresLlaves = do
                 {
                     x <- getChar
                       ; y <- getChar
                    ; z <- getChar
                  ; return (x, y, z)
                 }
```

   Aunque lo habitual es respetar el sangrado incluso cuando se usan las llaves y los puntos y coma.

3. Todas las acciones deben ser de tipo `IO`, pero no es necesario que todas tengan el mismo tipo `a`. Por ejemplo:

```
leeDos :: IO (String, Integer)
leeDos = do
   nombre <- getLine      -- IO String
   edad <- readLn         -- IO Integer
   return (nombre, edad)  -- IO (String, Integer)
```

4. Las acciones se ejecutan en el orden en que están escritas, de la primera a la última.

5. Las variables introducidas por un generador están visibles desde ese punto hasta el final del `do`, o hasta que sean ocultadas por un generador posterior (lo que se considera mal estilo).

6. Un generador de la forma `_ <-` *expresión* puede reescribirse simplemente como *expresión*, ya que no nos interesa el valor devuelto. Este truco se emplea frecuentemente con las acciones de salida (`putChar`, `putStr`, etc.) que simplemente devuelven `()`. Por ejemplo:

```
dosDeTres = do
    putStr "introduce tres caracteres: "
    x <- getChar
    _ <- getChar
    y <- getChar
    return (x, y)
```

   es equivalente a:

```
dosDeTres' = do
    putStr "introduce tres caracteres: "
    x <- getChar
    getChar       -- eliminamos el generador
    y <- getChar
    return (x, y)
```

7. La acción `return` no tiene por qué ser la última acción y, de hecho, puede haber varias acciones `return`. Esto se debe a que `return` **no interrumpe el flujo de control**; el objetivo de `return` es añadir cómputos puros de un tipo arbitrario a una secuencia de acciones. Recuerda que el tipo de `return` es `a -> IO a`, es decir, convierte un valor de tipo `a` en una acción de tipo `IO a`.

```
misterio = do
    x <- getX
    print x

getX =  do
    return "hola"
    return (5+2*3)
    return ("hola mundo" !! 3)
    return (foldr (+) 0 [1..10])
    return "adiós"
```

8. Una acción de la forma `x <- return y` puede reescribirse como `let x = y` (lo que vuelve a demostrar que `return` no altera el flujo de control).

9. La secuencia `do` tiene que terminar en una acción de entrada/salida; nunca en un generador o una definición local. Típicamente se trata de un `return`, aunque puede ser cualquier otra acción de entrada/salida. El tipo del `do` es el de su última acción.

10. El `do` puede anidarse: dentro de un `do` puede aparecer otro. Además, si una secuencia `do` tiene una sola acción, se puede eliminar el `do`.

```
favorito = do
   putStr "¿Cuál es tu lenguaje favorito?"
   xs <- getLine
   let ys = map toLower xs
   if ys == "haskell"
      then                              -- do eliminado
             putStr "¿cuál si no?"
      else do
             putStrLn "oh, vamos..."
             putStrLn ("Haskell es muy superior a " ++ xs)
```

11. Una definición con `do` puede ser recursiva.

```
favorito' = do
   putStr "¿Cuál es tu lenguaje favorito?"
   xs <- getLine
   let ys = map toLower xs
   if ys == "haskell"
      then
             putStr "¿cuál si no?"
      else do
             putStrLn "oh, vamos..."
             putStrLn ("Haskell es muy superior a " ++ xs)
             favorito'  -- recursión
```

Ejercicio 5. Repite los ejercicios anteriores usando notación `do`.

Ejercicio 6. Define una función polimórfica `pregunta :: Read a =>String ->IO a` que escriba el mensaje que recibe como parámetro por pantalla y devuelva la respuesta introducida por el usuario. Por ejemplo:

```
preguntas :: IO (String, Integer)
preguntas = do
   nombre <- pregunta "¿cómo te llamas?"
   edad <- pregunta "¿cuántos años tienes?"
   return (nombre, edad)
```

Ejercicio 7. Define una función `leeSecuencia :: Read a =>(a->Bool) ->IO [a]` que vaya leyendo por teclado una secuencia de valores de tipo `a` hasta leer uno que verifique el predicado que recibe como argumento. Los valores leídos deben devolverse en una lista, sin incluir el último. Por ejemplo:

```
Main> leeSecuencia (== -1)
6
0
-5
1024
8
-1
[6,0,-5,1024,8]
it :: [Integer]
```

5. Acceso a los parámetros del sistema

A veces necesitamos escribir programas que tengan acceso a los parámetros que se pasan en la línea de órdenes del sistema. Por ejemplo, podemos querer pasar nombres de ficheros a través de la línea de órdenes. Para acceder a estos parámetros, podemos utilizar la función:

```
getArgs :: IO [String]
```

exportada por el módulo `System.Environment`.

```
import System.Environment

muestraArgs :: IO ()
muestraArgs = do
    xs <- getArgs
    case xs of
        [] -> putStrLn "No se han pasado argumentos"
        _  -> putStrLn ("Se han pasado los argumentos " ++ show xs)
```

Ejercicio 8. Escribe un programa que imprima el número de parámetros del sistema que ha recibido y los parámetros convertidos a minúsculas. Por ejemplo:

```
C:> cuenta Esto es UN EjemPlO Simple
He recibido 5 parámetros:
esto
es
un
ejemplo
simple
```

6. Acciones y orden superior

El tipo `IO` es un tipo de primera clase; es decir, puede emplearse como cualquier otro tipo Haskell. Por ejemplo:

- Una acción puede almacenarse en una variable:

```
variable :: IO ()
variable = do
    let x = putStrLn "acción almacenada"
    x
    x
    x
    (id . id . id) x
    case [1..10] of
      (2:_) -> putStrLn "Mal"
      _     -> putStrLn "Bien"
```

- Una acción puede pasarse como parámetro a una función (por ejemplo, el operador `>>=`) o devolverse como resultado de una función (por ejemplo, `getChar`). La siguiente función ejecuta una acción dos veces:

```
dosVeces :: IO () -> IO ()
dosVeces io = do
    io
    io
```

- Podemos almacenar acciones en una estructura de datos; por ejemplo, en una lista. La siguiente función ejecuta una lista de acciones:

```
secuencia_ :: [IO a] -> IO ()
secuencia_ [] = return ()
secuencia_ (x:xs) = do
    x
    secuencia_ xs

ejemploSec :: IO ()
ejemploSec = secuencia_ [ putStrLn xs | xs <- ["ampliación ","de ","programación"] ]
```

  La función `secuencia_` está predefinida en `Prelude` con el nombre `sequence_`.

Finalizamos con el esqueleto un programa para aplicar la misma operación a cada fichero que se pasa como parámetro. Observa cómo separamos la interacción del cómputo:

```
module AplicaFich where

import System.Environment

-- interacción (código impuro)

main :: IO ()
main = do
    args <- getArgs
    sequence_ (map aplicaFich args)

aplicaFich :: FilePath -> IO ()
aplicaFich n = do
    xs <- readFile n
    writeFile (n ++ ".out") (opera xs)

-- cómputo (código puro)

opera :: String -> String
opera = ...
```

Ejercicio 9. Define una función `repite :: Int ->IO a ->IO ()` que ejecute `n` veces la acción de entrada/salida dada. Tienes que dar dos versiones, una recursiva y otra que utilice `sequence_`.

Ejercicio 10. Define la función `secuencia_` mediante *a)* el operador `>>=`, *b)* un `foldr` y *c)* un `foldl`.

Ejercicio 11. Define la función `when :: Bool ->IO () ->IO ()` que ejecuta una acción si y solo si se satisface la guarda. Por ejemplo:

```
ejemploWhen = do
   putStr "dame un número: "
   x <- readLn
   when (x < 0)  (putStrLn "Negativo")
   when (x == 0) (putStrLn "Cero")
   when (x > 0)  (putStrLn "Positivo")
```

7. Excepciones

Las operaciones de entrada/salida pueden no poder llevarse a cabo porque se produzca un error: el fichero no existe, no tenemos privilegios para acceder al fichero, el dispositivo está lleno, etc. Tales situaciones son señaladas mediante excepciones. Haskell define el tipo `IOError` para las excepciones que se pueden elevar al fracasar la ejecución de una acción `IO`. Para facilitar el manejo de estas excepciones, contamos con la función predefinida `cacth`:

```
catch :: IO a -> (IOError -> IO a) -> IO a
        acción        manejador      acción
```

que combina una acción `IO` con su correspondiente manejador. Lo habitual es utilizar `catch` como operador infijo. Por ejemplo:

```
leeEnteroOK1 :: IO Integer
leeEnteroOK1 =    readLn
              `catch`
                 \ excepcion -> do
                          print excepcion
                          return 0 -- valor por defecto
```

Es frecuente que el manejador sea recursivo, lo que permite repetir la acción hasta obtener un resultado correcto:

```
leeEnteroOK2 :: IO Integer
leeEnteroOK2 =    readLn
              `catch`
                 \ _ -> do
                          putStrLn "El valor introducido no es un entero."
                          putStrLn "Introduce otro, por favor."
                          leeEnteroOK2
```

También podemos propagar una excepción mediante `ioError :: IOError ->IO a`:

```
leeEnteroOK3 :: IO Integer
leeEnteroOK3 =    readLn
              `catch`
                 \ e -> ioError e -- se puede eta-reducir
```

O bien elevar nuestra propia excepción mediante `userError :: String ->IOError`:

```
leeEnteroOK4 :: IO Integer
leeEnteroOK4 =    readLn
              `catch`
                 \ _ ->  ioError (userError "Excepción: entero no válido")
```

Ejercicio 12. Define una función `preguntaOK` que se comporte como la función `pregunta` pero solicitando al usuario que vuelva a responder la pregunta hasta que introduzca un dato correcto.

Ejercicio 13. Define una función `intenta :: IO a ->IO (Either IOError a)` que intente ejecutar la acción que reciba como argumento y devuelva como resultado, bien una excepción, o bien el resultado de la acción. Esta función está predefinida en `Prelude` como `try`.

8. Ejercicios

Ejercicio 14. Escribe un programa Haskell que muestre un histograma con la frecuencia absoluta de las palabras que aparecen en un fichero de texto. El histograma debe estar ordenado alfabéticamente.

Utiliza el TAD bolsa para almacenar las palabras del texto y su número de apariciones.

Para extraer las palabras del texto, debes implementar un módulo `Tokeniser` que exporte la función

```
tokens :: (a -> Bool) -> a -> [a]
```

que separa una lista en sus *tokens* constituyentes. Un *token* es una secuencia de elementos consecutivos que pueden formar parte de un *token*. Los elementos que no pueden formar parte de un *token* se consideran separadores.

El primer argumento es un predicado (`a ->Bool`) que es cierto si un dato de tipo `a` puede formar parte de un *token*. El segundo argumento es la lista cuyos *tokens* queremos separar.

Por ejemplo, supongamos que un *token* está formado exclusivamente por caracteres alfabéticos; en tal caso, tenemos:

```
Main> tokens isAlpha "Esto es, simplemente, un ejemplo."
["Esto","es","simplemente","un","ejemplo"]
```

Aquí tienes otros ejemplos de uso de `tokens`:

```
Main> tokens isAlphaNum "x = x+1; if (x>0) {y = x++;} else {y = x--;}"
["x","x","1","if","x","0","y","x","else","y","x"]

Main> tokens isDigit "uno 1, dos 2, tres, 3"
["1","2","3"]
```

Observa que la función `tokens` es polimórfica, la listas no tienen por qué se de caracteres:

```
Main> tokens even  [1,2,3,4,4,5,5,9,11,13,2,2,2,15,16]
[[2],[4,4],[2,2,2],[16]]
```

Ejercicio 15. Escribe un programa Haskell que permita cifrar y descifrar un fichero de texto. El programa debe aceptar como parámetros el nombre del fichero a cifrar o descifrar, el nombre del fichero destino, la clave de cifrado/descifrado y una opción que indique si queremos cifrar o descifrar. Puedes utilizar cualquier método de cifrado simple que conozcas, excepto el método César, que está resuelto en clase. Si no conoces ningún método de cifrado simple, consulta referencias.

Ejercicio 16. Escribe programas Haskell que implementen las órdenes UNIX `wc`, `head`, `tail` y `more`. No es necesario implementar todas las opciones de estas órdenes. Si no conoces estas órdenes o no dispones de una instalación de UNIX para probarlas, puedes consultar su funcionamiento aquí http://www.linuxmanpages.com/.

Ejercicio 17. Escribe un programa Haskell que implemente un juego de adivinanza. El programa recibe como parámetros dos números positivos, `inf` y `sup` que indican un rango. Después genera un número aleatorio entre `inf` y `sup`. Si no se reciben parámetros, se toma por defecto `inf = 1` y `sup = 100`. El programa va solicitando por teclado conjeturas al jugador, y le va indicando si el número a adivinar es mayor, menor o igual a su conjetura, en cuyo caso el juego acaba.

Para generar un número aleatorio puedes utilizar el módulo `System.Random`, que exporta las funciones:

```
randomIO :: (Random a) => IO a
randomRIO :: (Random a) => (a, a) -> IO a
```

Todos los tipos básicos de Haskell (`Bool`, `Char`, `Int`, etc.) son instancia de la clase `Random`. La primera función devuelve un valor aleatorio del tipo `a`, mientras que la segunda devuelve un valor aleatorio dentro del rango indicado por la tupla.

Ejercicio 18. Escribe un programa Haskell para jugar al Nim. En el juego del Nim hay p pilas de fichas, cada una con un número de fichas entre 1 y n. Dos jugadores juegan por turnos: en cada turno se tiene que escoger una pila y retirar de ella al menos una ficha. No es posible pasar el turno. Gana el jugador que realiza el último movimiento; es decir, que retira las últimas fichas.

El programa debe aceptar como parámetros el número de pilas p y el número máximo de fichas n que puede haber en una pila. Con estos parámetros, el programa debe plantear un juego de Nim aleatorio. El estado del juego se representará con una lista de naturales, donde el elemento i-ésimo representa el número de fichas que quedan en la pila i-ésima. Por ejemplo, un juego de Nim con 4 pilas podría representarse por `[6, 1, 8, 3]`, y se visualizará en pantalla de la siguiente manera:

```
[1] : ******
[2] : *
[3] : ********
[4] : ***
```

Cada jugador debe introducir en su turno dos números: la pila y el número de fichas que desea retirar.

Tienes que implementar al menos una versión que permita jugar a dos personas. Si te atreves, también puedes implementar una versión para jugar contra la máquina. Puedes encontrar más información sobre el Nim y sus variantes aquí http://en.wikipedia.org/wiki/Nim.