OPERACIONES DE ENTRADA Y SALIDA

- En los lenguajes imperativos, los programas están formados por *acciones* que examinan y modifican el estado actual del *sistema*. Algunas acciones típicas son la lectura y modificación de variables globales, la escritura en ficheros, la lectura de datos y el manejo de ventanas en entornos gráficos. El sistema de entrada - salida IO de Haskell permite utilizar acciones de este tipo, aunque están claramente separadas del núcleo puramente funcional del lenguaje.

La Monada IO

- El sistema de IO de Haskell está basado en un fundamento matemático que puede asustar a primera vista: las *mónadas*. Sin embargo, no es necesario conocer la teoría de mónadas subyacente para programar usando el sistema de IO , más bien, las mónadas son simplemente una estructura conceptual en las que las IO encaja.

La Monada IO

- Las acciones son definidas, pero no invocadas, al nivel de la máquina de evaluación que dirige la ejecución de un programa Haskell. La evaluación de la definición de una acción no hace que la acción sea realizada. Más bien, la ejecución de acciones es efectuada en un nivel distinto a la evaluación de expresiones que hemos considerado hasta este momento.

La Monada IO

- Las acciones son atómicas, como el caso de las definidas como primitivas del sistema, o se obtienen de la composición secuencial de otras acciones. La mónada de IO contiene primitivas que permiten construir acciones compuestas. Esta mónada actúa como un pegamento que une las acciones que forman parte de un programa.

Operaciones de IO básicas

- IO a es el tipo de una acción que puede realizar operaciones de entrada y salida, y que cuando es realizada genera un resultado del tipo a. Por ejemplo :
- getChar :: IO Char
- Esta función no devuelve un carácter, sino una acción, que cuando sea efectuada generará un carácter.

Operaciones de IO básicas

- Las acciones que solo hacen algo pero no devuelven ningún resultado tienen tipo IO().
- Ejemplo :
- putChar :: Char -> IO()
- es una función que toma un carácter y devuelve la acción que escribe este por pantalla.
- `Hugs> getChar`
- `r`
- `Hugs> putChar 'k'`
- `k`

Operaciones de IO básicas

- getLine :: IO String
- putStr :: String -> IO()
- Son funciones para leer cadenas del teclado y escribirlas por teclado
- `Hugs> getLine`
- `jsjfkj`

- `Hugs> putStr "gfjhg"`
- `gfjhg`

Operaciones de IO básicas

- La función :
- return :: a ->IO a
- Puede ser utilizada para convertir un valor en una acción. Así, return x es la acción que cuando se realice generará como resultado x, pero, no es posible convertir una acción en un valor puro, ya que se comprometería la transparencia referencial del lenguaje.

Nuevas acciones a partir de básicas utilizando Expresiones Do

- import Char
- miAccion::IO()
- miAccion = do
- putStr "Dame un texto:"
- xs<- getLine
- putStr "En mayusculas es:"
- putStr (map toUpper xs)

Consideraciones sobre las expresiones do

- Las acciones se efectúan secuencialmente en el orden especificado.
- El ámbito de una variable introducida por <- son las acciones posteriores hasta el final de la expresión do.
- El tipo de una expresión do es el tipo de su ultima acción.
- La regla del sangrado se aplica después de la palabra do, por lo que todas las acciones de su ámbito deben estar en el a misma columna.

Podemos definir lista de acciones

- acciones :: [IO()]
- acciones =[putStr "Dame tu nombre:",responder]
- where
- responder = do
- xs<-getLine
- putStr("hola"++" "++xs)
- Main> sequence_ acciones
- Dame tu nombre:"Andrea"
- hola "Andrea"

Utilizó la función predefinida

```
sequence_          :: [IO ()] -> IO ()
sequence_ []       =  return ()
sequence_ (a:as) =  do
                                  a
                                  sequence_  as
```

Excepciones

- Una excepción es una condición de error que puede ser corregida. En Haskell las excepciones que se introducen al realizar acciones de entrada y salida tienen como tipo IOError.
- Las excepciones pueden ser tratadas con la función:
- catch :: IO a -> (IOError -> IO a) ->IO a
- Que suele usarse de modo infijo. El primer argumento es la acción a realizar, mientras que el segundo es una función que, en caso de que se produzca una excepción, tomará ésta como argumento y será ejecutada.

Ejemplo de excepciones

- muestraArchivo :: IO()
- muestraArchivo = do
- putStr "Escribe el nombre del archivo: "
- nombre <- getLine
- contenido <- readFile nombre
- putStr contenido

- `Main> muestraArchivo`
- `Escribe el nombre del archivo: Practica2.1.hs`

Si el archivo no existe se produce una excepcion

- `Main> muestraArchivo`
- `Escribe el nombre del archivo: noexiste.hs`

- `Program error: noexiste.hs: IO.openFile: does not exist (file does not exist)`

Pide iterativamente un nuevo nombre de archivo

- verArchivo :: IO()
- verArchivo = muestraArchivo`catch` manejador
- where
- manejador err = do
- putStr ("Se produjo un error " ++ show err ++ "\n")
- verArchivo
- Main> verArchivo
- Escribe el nombre del archivo: Practica2.2
- Se produjo un error Practica2.2: IO.openFile: does not exist (file does not exist)
- Escribe el nombre del archivo:

READLN : LEE VALORES DEL TECLADO QUE PERTENESCAN A LA CLASE READ

```
leer_Entero :: IO Int
leer_Entero = readLn `catch` manejador
      where
         manejador err = do
                              putStr " Error. Prueba de nuevo "
                              leer_Entero
Main> leer_Entero
8

Main> leer_Entero
h
 Error. Prueba de nuevo w
 Error. Prueba de nuevo 9
```

Programa mejorado para leer un entero

- leerEntero :: IO()
- leerEntero = do
- putStr " Introduce un entero: "
- n <- leer_Entero
- putStr(" El entero introducido es : " ++ show n)

```
Main> leerEntero
 Introduce un entero: k
 Error. Prueba de nuevo 8
 El entero introducido es : 8
```

Print : escribe por pantalla datos que pertenezcan a la clase show

- print :: Show a => a -> IO()
- print = putStrLn . show

- `Main> print " hola mundo "`
- `" hola mundo "`

- Observe que se realiza un salto de línea

Print : Utilizó la función predefinida

- putStrLn ::String -> IO()
- putStrLn s = do
- putStr s
- putChar '\n'

Expresiones case

- Realizan una comprobaciones de patrones en cualquier punto de una expresión.

- case exp of
- patron1 -> resultado1
- patron2 -> resultado2.
  - .
  - .
  - .

  patronN -> resultado3

Otras funciones para el manejo de excepciones

- El programador puede provocar una excepción del ultimo tipo desde cualquier acción mediante la función :
- userError :: String ->IOError

- O propagar una excepción existente mediante la función :
- ioError :: IOError->IO a

Otras funciones para el manejo de excepciones

```
try :: IO a -> IO (Either IOError a)
try p = do
   x<-p
   return (Right x) `catch` (\ err -> return (Left err))
-------------------------------------------------------------
bracket_ :: IO a -> (a -> IO b) -> IO c ->IO c
bracket_ antes despues acc = do
                              x <- antes
                              rAcc <- try acc
                              despues x
                              case rAcc of
                                   Right r -> return r
                                  Left e -> ioError e
```