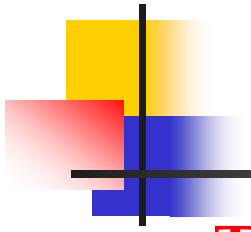


Tema 10. Programando con acciones: I/O.

- **Trasparencia referencial:**
  - "El valor de una expresión está totalmente determinado por el valor de sus subexpresiones"
  - "Las variables en Programación Funcional son variables matemáticas"
  - "No hay asignaciones ni efectos laterales"
  - Característica de la Prog. Declarativa (Funcional, Lógica)
- **Ventajas:**
  - Mayor legibilidad y fiabilidad de los programas
  - Permite razonamiento matemático (ecuacional)

Trasparencia versus Opacidad referencial

- Opacidad referencial de lenguajes imperativos (ó mixtos):
  - Efectos laterales
  - Funciones dependientes de su historia

Ejemplo:

```
function f (x: Int) return Int
    begin
      read y
      if  par y then return x else return x+1
    end
```

- *¿Valor de una expresión?* ¿ f(3) ?
- *¿Razonamiento?* ¿ f(3) + f(3) = 2* f(3) ?

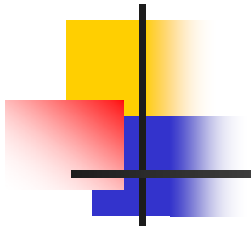
Problemas de la Entrada/Salida en P.F.

- ¿Cómo añadir al modelo funcional una acción de entrada / salida?

  Supongamos inputnum :: Int una expresión que devuelve un valor (numérico) leído del "input" (el que corresponda)

Sea el input actual: 5 8 2 6 .....
Sea la expresión: inputdif = inputnum – inputnum

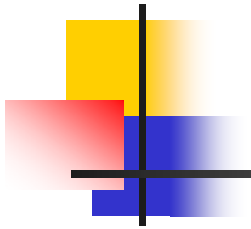
  - *¿Valor de* inputdif *? ¿Orden de evaluación?*
  - *¿Razonamiento matemático?*
  - *¿Trasparencia referencial?*



Soluciones a la Entrada/Salida en P.F.

- **Solución "basada en streams"**
  - el input (I) y el output (O) son :: String
  - un programa de I/O es una función f :: String –> String
  - utilizada en Miranda y primeras versiones de Haskell

- **Solución mediante la "mónada IO"**
  - solución más robusta y moderna
  - un programa de I/O es una acción f :: IO α
  - utilizada en la definición de Haskell98



Programación monádica

- **Usos de las mónadas**
  - Entrada/salida (mediante la mónada IO)
  - Otras clases de interacción “con el mundo exterior”
  - Ejecución de acciones secuenciales
  - Efectos laterales
- **Con las ventajas**
  - Visto como un “entorno imperativo” sobre Haskell, *sin “comprometer” el modelo funcional* subyacente
  - Con leyes precisas para *razonar* sobre las mónadas

¿Qué es una mónada?

- Desde un punto de vista matemático:

una mónada es una terna (m, return, (>>=)) consistente en:

- un "constructor de tipos" m
- una operación return :: α –> m α
- una operación (>>=):: m α –> (α –> m β) –> m β

cumpliendo 3 leyes:

- neutro a izquierda: return a >>= f = f a
- neutro a derecha: : p >>= return = p
- asociatividad:

(p >>= f) >>= g = p >>= s **where** s x = f x >>= g

La clase Monad en Haskell

```
class Monad m where
    return :: α –> m α
    (>>=) :: m α –> (α –> m β) –> m β
    (>>) :: m α –> m β –> m β
    fail :: String –> m α

    -- Minimal complete definition: (>>=), return
    p >> q  =  p >>= \ _ –> q
    fail s     =  error s
```

Operación (>>=) de la clase Monad

```
(>>=) :: Monad m  => m α –> (α –> m β) –> m β
```

p :: m α — p es un programa que realiza alguna acción (relativa a m) y devuelve un valor de tipo α

q :: α –> m β — q es una función que aplicada a un valor x::α devuelve el programa q x :: m β

El operador (>>=) sirve para combinar p con q :

p >>= q — programa (de tipo m β) obtenido al ejecutar primero el programa p, que devolverá un valor x, y ejecutar después el programa q x

Operación return de la clase Monad

```
return :: Monad m  => α –>  m α
```

return es una función que aplicada a un valor x::α

devuelve el programa return x :: m α , donde:

return x es un programa que no realiza acción alguna y devuelve el valor x::α

"La idea de return es elevar el tipo α al tipo m α, para una mónada m dada"

Operación (>>) de la clase Monad

```
(>>) :: Monad m  => m α –>  m β –> m β
```

p :: m α — p es un programa que realiza alguna acción (relativa a m) y devuelve un valor de tipo α

q :: m β — q es un programa que realiza alguna acción (relativa a m) y devuelve un valor de tipo β

El operador (>>) sirve para combinar dos acciones:

p >> q — programa (de tipo m β) obtenido al ejecutar primero el programa p, ignorar el valor devuelto, y ejecutar después el programa q

Instancias de la clase Monad

- Para definir una instancia de Monad se especificará:
  - Un “constructor de tipos” para m
  - Una operación para (>>=) y otra para return

  que cumplan las 3 leyes dadas

Ejemplo: -- *instancia predefinida*

```
instance Monad [] where
    (x:xs) >>= f    = f x ++ (xs >>= f)
    []     >>= f    = []
    return x        = [x]
    fail s          = []
```

Maybe: Instancia de Monad

- tipo Maybe:

```
data  Maybe α   =   Nothing | Just α
                    deriving (Eq, Ord, Read, Show)
```

- Maybe como instancia predefinida de Monad:

```
instance Monad Maybe where
        Just x  >>= k          = k x
        Nothing >>= k          = Nothing
        return                 = Just
        fail s                 = Nothing
```

IO: instancia de Monad

- tipo IO (entrada/salida)

  newtype IO α -- *builtin datatype of IO actions*

- IO como instancia predefinida de Monad:

```
instance Monad IO where
    (>>=)   =   primbindIO
    return  =   primretIO
    fail s  =   ioError (userError s)
```

(donde primbindIO y primretIO son primitivas del sistema)

El tipo IO α

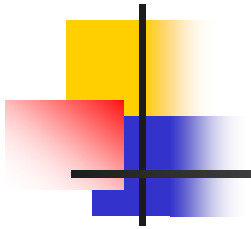
- Una expresión de tipo IO α realiza una *acción de entrada/salida* y produce un *resultado de tipo α*

- Principales operaciones (predefinidas) de IO α

- putChar :: Char –> IO ( ) -- *imprime un carácter*
- putStr :: String –> IO ( ) -- *imprime un string*
- putStrLn :: String –> IO ( ) -- *imprime un string y salta de línea*

donde ( ) es el único elemento del tipo trivial ( )

- getChar :: IO Char -- *lee un carácter*
- getLine :: IO String -- *lee un string*



La notación “do”: Ejemplo 1

- Es una notación alternativa de Haskell para escribir combinaciones de (>>=) con “cláusulas where” anidadas

Ejemplo 1:

```
-- Programa que lee n caracteres del teclado
leer :: Int –> IO String
leer 0 = return []
leer n =  getChar >>= q
                  where q c = leer (n-1) >>= r
                                    where r cs = return (c:cs)
```

La notación “do”: Ejemplo 1’

- El ejemplo anterior sería equivalente al siguiente programa con notación “do”:

Ejemplo 1’:

```
-- Programa que lee n caracteres del teclado
leer’ :: Int –> IO String
leer’ 0 = return []
leer’ n  =  do
                c  <– getChar
                cs <– leer’ (n-1)
                return (c:cs)
```

La notación “do”: Ejemplo 2

Ejemplo 2:

```
-- Para leer una línea del teclado (hasta el return)
leerLinea :: IO String                    -- predefinido: getLine
leerLinea  =  getChar  >>= q
              where
               q c =   if c == ‘\n’
                       then return “”
                       else  leerLinea >>= r
                             where r cs = return (c:cs)
```

La notación “do”: Ejemplo 2’

- Equivalente al ejemplo 2 pero con notación “do”:

```
Ejemplo 2’:        -- Para leer una línea del teclado (hasta el return)
leerLinea’ :: IO String
leerLinea’  =  do
                  c  <– getChar
                  if c == ‘\n’
                  then return “”
                  else   do
                         cs <– leerLinea’
                         return (c:cs)
```

Traducción de la notación “do”

- La traducción de una expresión “do” a operaciones con (>>=) y “cláusulas where” se rige por dos reglas:

1.

```
do  r
```

(-- r una acción :: M b) se traduce a

```
r
```

2.

```
do  x <– p
    C
    r
```

-- p :: M α

-- C secuencia de acciones

-- r :: M β

se traduce a

```
p >>= f
where  f  x  =  do  C
                    r
```

Ejemplo de traducción del “do” (1)

- El caso ( ) <– p se abrevia a p

Ejemplo:

```
do   putChar ‘a’
     putStr “miga”
```

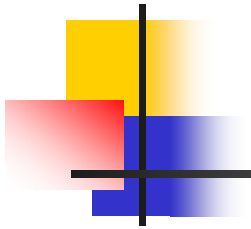
-- *como* ( ) <– putChar ‘a’

equivale a

```
putChar ‘a’ >>  putStr “miga”
```

e igualmente a

```
putChar ‘a’ >>= f   where  f  _  = putStr “miga”
```



Ejemplo de traducción del “do” (2)

-- *Programa que lee un carácter y lo imprime en mayúscula*

```
prog1’ =  do  x <– getChar
              putChar (toUpper x)
              putChar ‘\n’
```

equivale al programa sin “do”:

```
prog1 = getChar >>= f
        where
        f x = putChar (toUpper x) >> putChar ‘\n’
```

Funciones “print” y “read”

-- *Para imprimir valores de tipos “mostrables”:*

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

```
? print 5
  5
```

-- *Para leer valores de tipos “leibles”:*

- read :: Read α => String –> α

Ejemplo:

```
leerEnt :: IO Int
leerEnt =  do   e <– getLine
                return (read e)
```

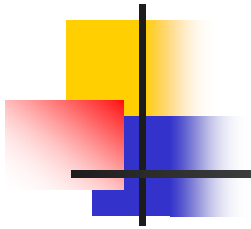
Ejemplos de programas IO

-- *Programa que lee una lista de enteros (escrita como lista):*

```
leerLisEnt :: IO [Int]
leerLisEnt = do   lin <– getLine
                  return (read lin)
```

-- *Programa que lee una lista de enteros, aplica la función parámetro e imprime la lista resultante:*

```
mapLisEnt :: Show α => (Int –> α) –> IO( )
mapLisEnt f  = do  lis <– leerLisEnt
                   print (map f  lis)
```



Más ejemplos de programas IO

```
-- Programa que lee enteros (uno en cada línea) y los devuelve
   en una lista:                    leerLisEnt2:: IO [Int]
   leerLisEnt2 = do  lis <– getLine
                     if lis == ""  then return []
                     else  do   resto <– leerLisEnt2
                                return ( (read lis) : resto )

-- Programa que lee enteros (uno en cada línea) e imprime la
   suma de todos ellos:             leeYsuma :: IO ( )
   leeYsuma = do   lisEnt <– leerLisEnt2
                   print (sum lisEnt)
```

Ficheros de entrada/salida

-- *Para leer el contenido de un fichero (como string):*

- readFile :: FilePath –> IO String

-- *Para escribir un string en un fichero:*

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

-- *Para añadir un string al final de un fichero:*

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

donde **type** FilePath = String -- *es el "path"del fichero (usar / en vez de \ )*

Ejemplos con ficheros

```
prog2 = do
        s <– readFile “C:/...../entrada.txt”
        writeFile “C:/……/salida.txt” (map toUpper s)
        putStrLn “salida.txt es entrada.txt en mayúsculas”

prog3
=  do   s <– readFile “C:/...../entrada.txt”
        appendFile “C:/……/salida.txt”
                ( “\n\n\t\t  -- En mayúsculas queda: --\n\n” ++
                  (map toUpper s))
        putStrLn “añadido a salida.txt”
```