

Tema 13: Programas interactivos

Informática (2019–20)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 13: Programas interactivos

1. Programas interactivos
2. El tipo de las acciones de entrada/salida
3. Acciones básicas
4. Secuenciación
5. Primitivas derivadas
6. Ejemplos de programas interactivos
 - Juego de adivinación interactivo
 - Calculadora aritmética
 - El juego de la vida

Programas interactivos

- ▶ Los programas por lote no interactúan con los usuarios durante su ejecución.
- ▶ Los programas interactivos durante su ejecución pueden leer datos del teclado y escribir resultados en la pantalla.
- ▶ Problema:
 - ▶ Los programas interactivos tienen efectos laterales.
 - ▶ Los programas Haskell no tienen efectos laterales.

Ejemplo de programa interactivo

- ▶ Especificación: El programa pide una cadena y dice el número de caracteres que tiene.

- ▶ Ejemplo de sesión:

```
*Main> longitudCadena  
Escribe una cadena: "Hoy es lunes"  
La cadena tiene 14 caracteres
```

- ▶ Programa:

```
longitudCadena :: IO ()  
longitudCadena = do putStr "Escribe una cadena: "  
                    xs <- getLine  
                    putStr "La cadena tiene "  
                    putStr (show (length xs))  
                    putStrLn " caracteres"
```

El tipo de las acciones de entrada/salida

- ▶ En Haskell se pueden escribir programas interactivos usando tipos que distingan las expresiones puras de las **acciones** impuras que tienen efectos laterales.
- ▶ `IO a` es el tipo de las acciones que devuelven un valor del tipo `a`.
- ▶ Ejemplos:
 - ▶ `IO Char` es el tipo de las acciones que devuelven un carácter.
 - ▶ `IO ()` es el tipo de las acciones que no devuelven ningún valor.

Acciones básicas

- ▶ `getChar :: IO Char`

La acción `getChar` lee un carácter del teclado, lo muestra en la pantalla y lo devuelve como valor.

- ▶ `putChar :: c -> IO ()`

La acción `putChar c` escribe el carácter `c` en la pantalla y no devuelve ningún valor.

- ▶ `return a -> IO a`

La acción `return c` devuelve el valor `c` sin ninguna interacción.

- ▶ Ejemplo:

```
*Main> putChar 'b'  
b*Main> it  
( )
```

Secuenciación

- ▶ Una sucesión de acciones puede combinarse en una acción compuesta mediante expresiones `do`.
- ▶ Ejemplo:

```
ejSecuenciacion :: IO (Char,Char)
ejSecuenciacion = do x <- getChar
                    getChar
                    y <- getChar
                    return (x,y)
```

Lee dos caracteres y devuelve el par formado por ellos. Por ejemplo,

```
*Main> ejSecuenciacion
b f
('b', 'f')
```

Primitivas derivadas de lectura

- ▶ Lectura de cadenas del teclado:

```
_____ Prelude _____  
getLine :: IO String  
getLine = do x <- getChar  
             if x == '\n' then return []  
             else do xs <- getLine  
                   return (x:xs)
```

Primitivas derivadas de escritura

- ▶ Escritura de cadenas en la pantalla:

```
_____ Prelude _____  
putStr :: String -> IO ()  
putStr []      = return ()  
putStr (x:xs) = do putChar x  
                  putStr xs
```

- ▶ Escritura de cadenas en la pantalla y salto de línea:

```
_____ Prelude _____  
putStrLn :: String -> IO ()  
putStrLn xs = do putStr xs  
                putChar '\n'
```

Primitivas derivadas: lista de acciones

- ▶ Ejecución de una lista de acciones:

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

Por ejemplo,

```
*Main> sequence_ [putStrLn "uno", putStrLn "dos"]
uno
dos
*Main> it
()
```

Ejemplo de programa con primitivas derivadas

- ▶ Especificación: El programa pide una cadena y dice el número de caracteres que tiene.

- ▶ Ejemplo de sesión:

```
*Main> longitudCadena
Escribe una cadena: "Hoy es lunes"
La cadena tiene 14 caracteres
```

- ▶ Programa:

```
longitudCadena :: IO ()
longitudCadena = do putStr "Escribe una cadena: "
                    xs <- getLine
                    putStr "La cadena tiene "
                    putStr (show (length xs))
                    putStrLn " caracteres"
```

Tema 13: Programas interactivos

1. Programas interactivos
2. El tipo de las acciones de entrada/salida
3. Acciones básicas
4. Secuenciación
5. Primitivas derivadas
6. Ejemplos de programas interactivos
 - Juego de adivinación interactivo
 - Calculadora aritmética
 - Financiera básica

Juego de adivinación interactivo

- ▶ Descripción: El programa le pide al jugador humano que piense un número entre 1 y 100 y trata de adivinar el número que ha pensado planteándole conjeturas a las que el jugador humano responde con mayor, menor o exacto según que el número pensado sea mayor, menor o igual que el número conjeturado por la máquina.
- ▶ Ejemplo de sesión:

```
Main> juego  
Piensa un numero entre el 1 y el 100.  
Es 50? [mayor/menor/exacto] mayor  
Es 75? [mayor/menor/exacto] menor  
Es 62? [mayor/menor/exacto] mayor  
Es 68? [mayor/menor/exacto] exacto  
Fin del juego
```

Juego interactivo

► Programa:

```
juego :: IO ()
juego =
  do putStrLn "Piensa un numero entre el 1 y el 100."
     adivina 1 100
     putStrLn "Fin del juego"

adivina :: Int -> Int -> IO ()
adivina a b =
  do putStr ("Es " ++ show conjetura ++ "? [mayor/menor/exacto] ")
     s <- getLine
     case s of
       "mayor" -> adivina (conjetura+1) b
       "menor" -> adivina a (conjetura-1)
       "exacto" -> return ()
       _        -> adivina a b
  where
    conjetura = (a+b) `div` 2
```

Segundo juego de adivinación interactivo

- ▶ Descripción: En el segundo juego la máquina genera un número aleatorio entre 1 y 100 y le pide al jugador humano que adivine el número que ha pensado planteándole conjeturas a las que la máquina responde con mayor, menor o exacto según que el número pensado sea mayor, menor o igual que el número conjeturado por el jugador humano.
- ▶ Ejemplo de sesión:

```
Main> juego2
Tienes que adivinar un numero entre 1 y 100
Escribe un numero: 50
    es bajo.
Escribe un numero: 75
    es alto.
Escribe un numero: 62
Exactamente
```

Segundo juego interactivo

► Programa:

```
import System.Random (randomRIO)

juego2 :: IO ()
juego2 = do n <- randomRIO (1::Int,100)
           putStrLn "Tienes que adivinar un numero entre 1 y 100"
           adivina' n

adivina' :: Int -> IO ()
adivina' n =
  do putStr "Escribe un numero: "
     c <- getLine
     let x = read c
     case (compare x n) of
       LT -> do putStrLn " es bajo."
              adivina' n
       GT -> do putStrLn " es alto."
              adivina' n
       EQ -> putStrLn " Exactamente"
```

Tema 13: Programas interactivos

1. Programas interactivos
2. El tipo de las acciones de entrada/salida
3. Acciones básicas
4. Secuenciación
5. Primitivas derivadas
6. Ejemplos de programas interactivos
 - Juego de adivinación interactivo
 - Calculadora aritmética
 - Ejemplo de...

Acciones auxiliares

► Importaciones

```
import I1M.Analizador
import System.IO
```

► Escritura de caracteres sin eco:

```
getCh :: IO Char
getCh = do hSetEcho stdin False
           c <- getChar
           hSetEcho stdin True
           return c
```

► Limpieza de la pantalla:

```
limpiaPantalla :: IO ()
limpiaPantalla = putStr "\ESC[2J"
```

Acciones auxiliares

- ▶ Escritura en una posición:

```
type Pos = (Int,Int)
```

```
irA :: Pos -> IO ()
```

```
irA (x,y) = putStr ("\ESC[" ++  
                    show y ++ ";" ++ show x ++  
                    "H")
```

```
escribeEn :: Pos -> String -> IO ()
```

```
escribeEn p xs = do irA p  
                    putStr xs
```

- ▶ En las funciones `limpiaPantalla` e `irA` se han usado **códigos de escape ANSI**

Calculadora

```
calculadora :: IO ()
calculadora = do limpiaPantalla
                escribeCalculadora
                limpiar

escribeCalculadora :: IO ()
escribeCalculadora =
  do limpiaPantalla
     sequence_ [escribeEn (1,y) xs
                | (y,xs) <- zip [1..13] imagenCalculadora]
     putStrLn ""
```

Calculadora

```

imagenCalculadora :: [String]
imagenCalculadora = ["+-----+",
                    "|           |",
                    "+---+---+---+---+",
                    "| q | c | d | = |",
                    "+---+---+---+---+",
                    "| 1 | 2 | 3 | + |",
                    "+---+---+---+---+",
                    "| 4 | 5 | 6 | - |",
                    "+---+---+---+---+",
                    "| 7 | 8 | 9 | * |",
                    "+---+---+---+---+",
                    "| 0 | ( | ) | / |",
                    "+---+---+---+---+"]

```

Los primeros cuatro botones permiten escribir las órdenes:

- ▶ q para salir ('quit'),
- ▶ c para limpiar la agenda ('clear'),
- ▶ d para borrar un carácter ('delete') y
- ▶ = para evaluar una expresión.

Los restantes botones permiten escribir las expresiones.

Calculadora

```
limpiar :: IO ()
```

```
limpiar = calc ""
```

```
calc :: String -> IO ()
```

```
calc xs = do escribeEnPantalla xs
```

```
    c <- getCh
```

```
    if elem c botones
```

```
        then procesa c xs
```

```
        else do calc xs
```

```
escribeEnPantalla xs =
```

```
    do escribeEn (3,2) " "
```

```
        escribeEn (3,2) (reverse (take 13 (reverse xs)))
```

Calculadora

```
botones :: String
botones = standard ++ extra
  where
    standard = "qcd=123+456-789*0()/"
    extra    = "QCD \ESC\BS\DEL\n"

procesa :: Char -> String -> IO ()
procesa c xs
  | elem c "qQ\ESC"    = salir
  | elem c "dD\BS\DEL" = borrar xs
  | elem c "=\n"       = evaluar xs
  | elem c "cC"        = limpiar
  | otherwise          = agregar c xs
```

Calculadora

```
salir :: IO ()
```

```
salir = irA (1,14)
```

```
borrar :: String -> IO ()
```

```
borrar "" = calc ""
```

```
borrar xs = calc (init xs)
```

```
evaluar :: String -> IO ()
```

```
evaluar xs = case analiza expr xs of  
    [(n,"")] -> calc (show n)  
    _         -> do calc xs
```

```
agregar :: Char -> String -> IO ()
```

```
agregar c xs = calc (xs ++ [c])
```


Tema 13: Programas interactivos

1. Programas interactivos
2. El tipo de las acciones de entrada/salida
3. Acciones básicas
4. Secuenciación
5. Primitivas derivadas

6. Ejemplos de programas interactivos

- Juego de adivinación interactivo

- Calculadora aritmética

- El juego de la vida

Descripción del juego de la vida

- ▶ El tablero del juego de la vida es una malla formada por cuadrados (“células”) que se pliega en todas las direcciones.
- ▶ Cada célula tiene 8 células vecinas, que son las que están próximas a ella, incluso en las diagonales.
- ▶ Las células tienen dos estados: están “vivas” o “muertas”.
- ▶ El estado del tablero evoluciona a lo largo de unidades de tiempo discretas.
- ▶ Las transiciones dependen del número de células vecinas vivas:
 - ▶ Una célula muerta con exactamente 3 células vecinas vivas “nace” (al turno siguiente estará viva).
 - ▶ Una célula viva con 2 ó 3 células vecinas vivas sigue viva, en otro caso muere.

Funciones anteriores

```
import Data.List (nub)
```

```
type Pos = (Int,Int)
```

```
irA :: Pos -> IO ()
```

```
irA (x,y) = putStr ("\ESC[" ++ show y ++ ";" ++ show x ++ "H")
```

```
escribeEn :: Pos -> String -> IO ()
```

```
escribeEn p xs = do irA p  
                    putStr xs
```

```
limpiaPantalla:: IO ()
```

```
limpiaPantalla= putStr "\ESC[2J"
```

El tablero del juego de la vida

► Tablero:

```
type Tablero = [Pos]
```

► Dimensiones:

```
ancho :: Int  
ancho = 5
```

```
alto :: Int  
alto = 5
```

El juego de la vida

- ▶ Ejemplo de tablero:

```
ejTablero :: Tablero
```

```
ejTablero = [(2,3), (3,4), (4,2), (4,3), (4,4)]
```

- ▶ Representación del tablero:

```
| 1234  
1  
2  0  
3 0 0  
4  00
```

El juego de la vida

- ▶ `(vida n t)` simula el juego de la vida a partir del tablero `t` con un tiempo entre generaciones proporcional a `n`. Por ejemplo,
`| vida 100000 ejTablero`

```
vida :: Int -> Tablero -> IO ()  
vida n t = do limpiaPantalla  
              escribeTablero t  
              espera n  
              vida n (siguienteGeneracion t)
```

- ▶ Escritura del tablero:

```
escribeTablero :: Tablero -> IO ()  
escribeTablero t = sequence_ [escribeEn p "0" | p <- t]
```

El juego de la vida

- ▶ Espera entre generaciones:

```
espera :: Int -> IO ()  
espera n = sequence_ [return () | _ <- [1..n]]
```

- ▶ `siguienteGeneracion t`) es el tablero de la siguiente generación al tablero `t`. Por ejemplo,

```
*Main> siguienteGeneracion ejTablero  
[(4,3),(3,4),(4,4),(3,2),(5,3)]
```

```
siguienteGeneracion :: Tablero -> Tablero  
siguienteGeneracion t = supervivientes t ++ nacimientos t
```

El juego de la vida

- ▶ `(supervivientes t)` es la listas de posiciones de `t` que sobreviven; i.e. posiciones con 2 ó 3 vecinos vivos. Por ejemplo,
`| supervivientes ejTablero ~> [(4,3),(3,4),(4,4)]`

```
supervivientes :: Tablero -> [Pos]
supervivientes t = [p | p <- t,
                    elem (nVecinosVivos t p) [2,3]]
```

- ▶ `(nVecinosVivos t c)` es el número de vecinos vivos de la célula `c` en el tablero `t`. Por ejemplo,
`| nVecinosVivos ejTablero (3,3) ~> 5`
`| nVecinosVivos ejTablero (3,4) ~> 3`

```
nVecinosVivos :: Tablero -> Pos -> Int
nVecinosVivos t = length . filter (tieneVida t) . vecinos
```

El juego de la vida

(vecinos p) es la lista de los vecinos de la célula en la posición p . Por ejemplo,

vecinos (2,3) \rightsquigarrow [(1,2), (2,2), (3,2), (1,3), (3,3), (1,4), (2,4), (3,4)]

vecinos (1,2) \rightsquigarrow [(5,1), (1,1), (2,1), (5,2), (2,2), (5,3), (1,3), (2,3)]

vecinos (5,2) \rightsquigarrow [(4,1), (5,1), (1,1), (4,2), (1,2), (4,3), (5,3), (1,3)]

vecinos (2,1) \rightsquigarrow [(1,5), (2,5), (3,5), (1,1), (3,1), (1,2), (2,2), (3,2)]

vecinos (2,5) \rightsquigarrow [(1,4), (2,4), (3,4), (1,5), (3,5), (1,1), (2,1), (3,1)]

vecinos (1,1) \rightsquigarrow [(5,5), (1,5), (2,5), (5,1), (2,1), (5,2), (1,2), (2,2)]

vecinos (5,5) \rightsquigarrow [(4,4), (5,4), (1,4), (4,5), (1,5), (4,1), (5,1), (1,1)]

```
vecinos :: Pos -> [Pos]
```

```
vecinos (x,y) = map modular [(x-1,y-1), (x,y-1), (x+1,y-1),
                             (x-1,y),           (x+1,y),
                             (x-1,y+1), (x,y+1), (x+1,y+1)]
```

El juego de la vida

- `modular p` es la posición correspondiente a `p` en el tablero considerando los plegados. Por ejemplo,

```
modular (6,3)  ~> (1,3)
```

```
modular (0,3)  ~> (5,3)
```

```
modular (3,6)  ~> (3,1)
```

```
modular (3,0)  ~> (3,5)
```

```
modular :: Pos -> Pos
```

```
modular (x,y) = (1 + (x-1) 'mod' ancho,  
                1 + (y-1) 'mod' alto)
```

El juego de la vida

- ▶ `(tieneVida t p)` se verifica si la posición `p` del tablero `t` tiene vida. Por ejemplo,

```
tieneVida ejTablero (1,1)  ⇔ False
tieneVida ejTablero (2,3)  ⇔ True
```

```
tieneVida :: Tablero -> Pos -> Bool
tieneVida t p = elem p t
```

- ▶ `(noTieneVida t p)` se verifica si la posición `p` del tablero `t` no tiene vida. Por ejemplo,

```
noTieneVida ejTablero (1,1)  ⇔ True
noTieneVida ejTablero (2,3)  ⇔ False
```

```
noTieneVida :: Tablero -> Pos -> Bool
noTieneVida t p = not (tieneVida t p)
```

El juego de la vida

- `(nacimientos t)` es la lista de los nacimientos de tablero `t`; i.e. las posiciones sin vida con 3 vecinos vivos. Por ejemplo,

```
|nacimientos ejTablero  ~> [(3,2),(5,3)]
```

```
nacimientos' :: Tablero -> [Pos]
nacimientos' t = [(x,y) | x <- [1..ancho],
                        y <- [1..alto],
                        noTieneVida t (x,y),
                        nVecinosVivos t (x,y) == 3]
```

El juego de la vida

- Definición más eficiente de nacimientos

```
nacimientos :: Tablero -> [Pos]
nacimientos t = [p | p <- nub (concatMap vecinos t),
                  noTieneVida t p,
                  nVecinosVivos t p == 3]
```

Bibliografía

1. H. Daumé III. *Yet Another Haskell Tutorial*. 2006.
 - ▶ Cap. 5: Basic Input/Output.
2. G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.
 - ▶ Cap. 9: Interactive programs.
3. B. O'Sullivan, J. Goerzen y D. Stewart. *Real World Haskell*. O'Reilly, 2009.
 - ▶ Cap. 7: I/O.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
 - ▶ Cap. 7: Entrada y salida.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
 - ▶ Cap. 18: Programming with actions.