Programas interactivos

Entrada y salida por consola

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

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 programa Haskell no tiene efectos laterales.

Ejemplo de programa interactivo

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

```
ghci> 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 (de E/S) 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:

```
ghci> putChar 'b'
bghci> it
()
```

Nota: el comando `it` devuelve el resultado devuelto por la instrucción anteriormente introducida en el intérprete.

Secuenciación

- Una sucesión de acciones puede combinarse en una acción compuesta mediante expresiones do.
- Ejemplo: el procedimiento `ejSecuenciacion` lee dos caracteres y devuelve el par formado por ellos. Veamos en una sesión:

```
ghci> ejSecuenciacion
b f
('b','f')
```

- Definición de la misma:

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

Primitivas derivadas

Lectura

- Lectura de cadenas del teclado:

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

Primitivas derivadas

Escritura

- Escritura de cadenas en la pantalla:

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

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

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

Secuencia de acciones

- Ejecución de una lista de acciones (equivalente a poner cada elemento como una línea dentro de un `do`).
- Veamos una ejecución:

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

- Definición:

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

Secuencia de acciones

Ejemplos - I

- Ejemplo: programa pide una cadena y dice el número de caracteres que tiene.
- Veamos una ejecución:

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

- Código del programa:

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

Secuencia de acciones

Ejemplos - II

- Ejemplo: función que recibe una lista de cadenas y va imprimiendo una en cada línea.
- Veamos una ejecución:

```
ghci> pruebaF ["hola","vamos a probar","no me fío mucho"]
hola
vamos a probar
no me fío mucho
ghci>
```

- Código de la función:

```
pruebaF :: [String] -> IO ()
pruebaF xs = sequence_ (map putStrLn xs)
```

Compilación de programas

- Como en muchos lenguajes, la ejecución comienza por la función que se llama `main`:

```
main :: IO ()
main = putStrLn "hola mundo"
```

- Desde la terminal/cmd, puedes **evaluar** tu programa (si tu fichero se llama hola.hs):

```
$ runhaskell hola.hs
```

- Desde la terminal/cmd, puedes **compilar** tu programa (si tu fichero se llama hola.hs):

```
$ ghc hola.hs -o hola
$ ./hola
```

Compilación de programas

- Si has compilado el programa, ten en cuenta que en la salida se hace un buffering que se libera cuando se encuentra un salto de línea.
- `putStr` puede tener comportamiento distinto al esperado. Fuente.

```
main = do putStr "Who are you? "
          name <- getLine
          putStrLn ("Hello, " ++ name)
```

- Se puede arreglar de dos formas con el módulo `System.IO`:
  - Forzando la salida del buffer después de `putStr`

```
import System.IO
main = do putStr "Who are you? "
          hFlush stdout
          name <- getLine
          putStrLn ("Hello, " ++ name)
```

  - Desactivando el buffering en la salida

```
main = do
       hSetBuffering stdout NoBuffering
       putStr "Who are you? "
       name <- getLine
       putStrLn ("Hello, " ++ name)
```

Bibliografía


H. Daumé III. *Yet Another Haskell Tutorial*, 2006.
Chapter 5: Basic Input/Output


G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
Chapter 9: Interactive programs


B. O'Sullivan, D. Stewart y J. Goerzen. *Real World Haskell*. O'Reilly, 2008.
Chapter 7: I/O


B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. Razonando con Haskell. Thompson, 2004..
Capítulo 7: Entrada y salida


S. Thompson. Haskell: The Craft of Functional Programming, Second Edition. Addison-Wesley, 1999.
Chapter 9: Generalization: patterns of computation
Chapter 18: Programming with actions