¡A la caza de los datos!

Haskell Data Analysis Cookbook[1]

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

[1] N. Shukla. *Haskell data analysis cookbook: explore intuitive data analysis techniques and powerful machine learning methods using over 130 practical recipes*. Packt Publishing, 2014.

Contenidos

Leyendo un archivo de texto simple

Como vimos, podemos acceder de manera simple a un archivo de texto con `readFile "nombreArchivo"`:

```
main :: IO ()
main = do
   input <- readFile "input.txt"
   print input
```

Como vemos, hemos llamado a la función principal `main`. Es el punto de entrada a ejecutar si lanzamos Haskell sobre el fichero (*i.e.*, `Main.hs`) mediante:

```
runhaskell Main.hs
```

Podemos emplear `ghc` para generar un ejecutable:

```
Mi prompt>ghc Main.hs
[1 of 1] Compiling Main             ( Main.hs, Main.o )
Linking Main.exe ...

Mi prompt>Main.exe "input.txt"
[6,6,10,7,6,7]
```

Lógicamente, también podemos llamar a la función `main` desde `ghci`.

Ejemplos cubiertos

Los siguientes ejemplos serán analizados a lo largo de estas diapositivas[2]

- Accumulating text data from a file path
- Catching I/O code faults
- Keeping and representing data from a CSV file

[2] Los ejemplos presentados en estas diapositivas se han tomado del libro indicado en la portada, para facilitar la comprensión por parte del alumnado. Todo el mérito debe otorgarse a la fuente original.

Introducción

Hoy día hay datos por todas partes, pero antes de poder tratarlos y construir modelos necesitamos adquirirlos y organizarlos. A menudo las funtes de las que partamos pueden tener distinta naturaleza.

A lo largo de este conjunto de diapositivas se toman datos de diversas fuentes de datos.

1. Usando archivos locales en diferentes formatos de archivo.
2. Descargando de Internet desde Haskell.
3. Empleando bases de datos en Haskell.

Accumulating text data from a file path

Procesando archivo local

Este ejemplo procesa un archivo de texto:

```
main :: IO ()
main = do
   input <- readFile "input.txt"
   print (countWords input)
countWords :: String -> [Int]
countWords input = map (\line -> length (words line)) (lines input)
```

En este código leemos el texto (`readFile`) y lo sacamos a un String `input`, que pasamos a la función `countWords`, cuyo resultado imprimimos por pantalla.

En `countWords` extraemos de la cadena `input` su lista de líneas (`lines`), separadas en la cadena original por caracteres `\n`. Por cada línea, aplicamos una función que devuelve la lista de palabras de una cadena (`words`), de la que extraemos su longitud. Así:

```
$ runhaskell Main.hs
[6,6,10,7,6,7]
```

Catching I/O code faults

Tratando posibles errores

El ejemplo anterior funciona si el fichero buscado existe. Pero si no, nos salta un error no controlado. Se subsana procesando el error con `catch`:

```
import Control.Exception (catch, SomeException)
import System.Environment (getArgs)

main :: IO ()
main = do
   args <- getArgs
   let fileName = case args of
     (a:_) -> a
     _ -> "input.txt"
   input <- catch (readFile fileName)
                  (\err -> print (err::SomeException) >> return "")
   if (length input > 0) then (ejemplo1 input) else return ()

ejemplo1 = print.countWords

countWords :: String -> [Int]
countWords input = map (\line -> length (words line)) (lines input)
```

Veamos algunos aspectos de interés...

Catching I/O code faults

Tratando posibles errores - Recibiendo args

- Recibimos los argumentos pasados por línea de comandos con `getArgs`:

```
*Main> :t getArgs
getArgs :: IO [String]
```

Como vemos, nos devuelve algo del tipo `IO [String]`.

- Extraemos del contexto de IO (con `<-`) la lista de argumentos (en este caso, cadenas con cada argumento por separado).
- Haciendo un buen uso de patrones en `case of`, distinguimos si hemos recibido el nombre del archivo o no:

```
let fileName = case args of
    (a:_) -> a
    _ -> "input.txt"
```

  - Así, si la lista contiene parámetros (patrón `(a:_)`, a modo de nuestro habitual `(x:xs)`), interpretamos que el primer argumento es el nombre del fichero.
  - Si no encaja con ese patrón, y por tanto la lista de argumentos es vacía, por defecto asignamos el nombre `input.txt`

Catching I/O code faults

Tratando posibles errores - Capturando la excepción

- Capturamos con `catch` las posibles excepciones (`SomeException`) que aparezcan durante la lectura de fichero (`readFile fileName`).

```
*Main> :t catch
catch :: GHC.Exception.Exception e => IO a -> (e -> IO a) -> IO a
```

Esperamos algo de tipo `IO a`, y algo de tipo `e -> IO a`, con `e` un tipo de excepción.

```
catch (readFile fileName)
      (\err -> (print (err::SomeException) >> return ""))
```

En nuestro caso, `readFile fileName` devuelve `IO String`, luego el 2º argumento del catch debe ser una función que reciba excepción y devuelva `IO String`. Así lo hacemos, con una función lambda que recibe `err` y emplea la `return ""` para devolver el tipo deseado.

Catching I/O code faults

Tratando posibles errores - Distinguiendo casos

- Ya tenemos controlada la posible excepción, la sacamos del contexto IO:

```
input <- catch (...) (...)
```

- Si no saltó la excepción, `input` dispondrá del contenido del fichero en una cadena, pero si devolvimos la cadena vacía, es que saltó la excepción. Eso nos permite distinguir los casos a tratar:

```
if (length input > 0) then (ejemplo1 input) else return ()
```

  En este caso, al ser la última instrucción del `main` y estar definido éste como `IO ()`, ambas partes del `if` (`then` y `else`) deben devolver `IO ()`.

- En efecto, tenemos que:
  - La función `ejemplo1` es de tipo `String -> IO ()`, luego `(ejemplo1 input)` es de tipo `IO ()`.
  - En el contexto en que está empleado, `()` es de tipo `()`, luego al usar `return` nos queda la expresión `return ()` de tipo `IO ()`.

Catching I/O code faults

Tratando posibles errores - Procesando I

Por terminar de analizar el ejemplo:

- Lo anterior sería bastante genérico. Ya la llamada a `ejemplo1 input` o a cualquier otra función determinará cómo procesamos el fichero, qué hacemos con él.
- En el ejemplo:

```
ejemplo1 = print.countWords
```

Vemos que la función `ejemplo1` es equivalente a la composición de `countWords` con `print`. Es decir:

- Pasa el `input` a `countWords`, que devuelve una lista con el número de palabras de cada línea del fichero.
- La lista obtenida es pasada a `print`,

```
print :: Show a => a -> IO ()
```

que toma dato de cualquier tipo *mostrable* (restricción `Show`) y lo imprime por pantalla, devolviendo así `IO ()`.

Catching I/O code faults

¿Se nos ocurre una solución mejor?

- El tratamiento de excepciones permite capturar tratar casos poco frecuentes y/o inesperados
- No obstante, si podemos anticiparnos a ellas, es mejor hacerlo.
- Así, en nuestro ejemplo, donde capturamos la excepción al leer el fichero:

```
input <- catch (readFile fileName)
               (\err ->
                  (print (err::SomeException) >>
                   return ""))
```

Podemos en su lugar comprobar si el fichero existe, y solo si es así trataremos de leerlo:

```
exists <- doesFileExist fileName
input <- if exists then readFile fileName else return ""
```

Nota: `doesFileExist` pertenece al módulo `System.Directory`, luego debemos importarlo.

Archivos CSV

Los archivos CSV suelen contener datos de un determinado `dataframe`, un conjunto de datos conteniendo:

- Una primera fila de cabecera, con los nombres de los campos.
- Las restantes filas conteniendo los registros, con un valor para cada campo.

Generalmente, los datos vienen separados, dentro de la misma línea, por comas (`,`) o por punto y coma (`;`), según la región (en España se emplea `;`, puesto que usamos `,` para decimales.)

Keeping and representing data from a CSV file

Manejo de CSV en Haskell

- En Haskell disponemos de la biblioteca `csv` para procesar estos archivos. La instalamos con `cabal install csv`
- La importamos en nuestro fichero con: `import Text.CSV`[3]
- Veremos que con la función `parseCSV` podemos extraer a una *lista de registros* el contenido de un fichero `CSV` dado. Dicha función no devuelve exactamente la lista de registros, sino algo que puede ser de tipo `Text.Parsec.Error.ParseError`, si hay error en el análisis del fichero, o de tipo `CSV` si no lo hay.
- En el segundo caso, se devuelve una lista de registros (algo del tipo `CSV`, sinónimo de `[Record]`).
- Un `Record` consta (es sinónimo) de lista de fields (`[Field]`).

[3] Ver http://hackage.haskell.org/package/csv-0.1.2/docs/Text-CSV.html

Procesamiento básico de CSV

El menor procesamiento sobre este tipo de fuente puede conllevar:

- Leer el contenido del fichero (`readFile`)
- *Parsear* el fichero (`parseCSV`)
- Mostrar los registros del CSV (`printCSV`)

```
import Text.CSV

main :: IO ()
main = do
  let fileName = "input.csv"
  -- Leemos:
  input <- readFile fileName
  -- Parseamos:
  let csv = parseCSV fileName input
      filas = case csv of
        (Right lineas) -> lineas
        _ -> []
  filasValidas = filter (\x -> length x == 2) filas
  -- Mostramos:
  putStrLn $ printCSV filas
```

El tipo Either

- Es un tipo *parametrizado*
- Está definido como `data Either a b = Left a | Right b`
- Es similar a `Maybe`, pero:
  - En lugar de distinguir los patrones `Nothing` y `(Just x)`
  - Debemos ver qué hacer según si viene `(Left x)` o `(Right y)`
  - Como vemos, los tipos `a` y `b` pueden ser distintos

Entrando en detalle al ejemplo:

```
-- Analizamos esta cadena para ver si es un CSV válido
-- y en función de ello hacer una cosa u otra
let csv = parseCSV fileName input
-- csv es de tipo Either Text.Parsec.Error.ParseError CSV
-- Si no hay error, devuelve (Right x), con x de tipo CSV
-- Tratamos estos dos casos:
    filas = case csv of
      -- Devolvemos la lista de líneas del archivo si devuelve CSV
      (Right lineas) -> lineas
      -- En caso contrario (_, en este caso (Left y)), devolvemos []
      (Left err) -> []
```

Tratando el CSV

Exploremos un poco el contenido...

```
main = do
  let fileName = "input.csv"
  input <- readFile fileName
  let csv = parseCSV fileName input
      filas = case csv of
        (Right lineas) -> lineas
        (Left err) -> []
      filasValidas = filter (\x -> length x == 2) filas
      (cabecera,registros) = case filasValidas of
        [] -> ([],[])
        (cab:regs) -> (cab,regs)
  if null filasValidas then
    putStrLn "El fichero no es un CSV válido o carece de contenido"
  else do
    putStrLn $ "- Número de campos: " ++ show (length cabecera)
    putStrLn $ "- Lista de campos: " ++ "{" ++ (init.init.concat) ([ c ++ ", " | c <- cabecera
          ]) ++ "}"
    if null registros then
      putStrLn "Nada que mostrar"
    else do
      putStrLn $ "- Número de registros: " ++ show (length registros)
      putStrLn "- Registros:\n ---------"
      sequence_ [muestraReg reg cabecera | reg <- registros]
        where
          muestraReg reg cab =
            putStrLn $
              "\t{" ++
              (init.init.concat) ([cpo ++ ": " ++ val ++ ", " | (cpo,val) <- zip cab reg]) ++
              "}"
```

Consejo práctico

- Hemos empezado a complicar el procesamiento en el `main`
- Dependerá de si el `CSV` es válido o no
- Es más conveniente:
  - Delegar el tratamiento de errores a un manejador (pongamos `handleError`)
  - Delegar el procesamiento de los CSV válidos a otra función (pongamos `doWork`)
  - Llamar a una u otra según el resultado del parsing, para lo que podemos llamar a la función `either`:

```
either handleError doWork csv
```

Como vemos, le pasamos `handleError` (para tratar el error en esa función), `doWork` (para hacer el trabajo con el contenido del CSV, en caso de ser válido), y finalmente el propio dato `csv`, el dato de tipo `Either` *que podía devolver bien error o bien el csv válido.*

Manejo de CSV en Haskell

Mejorando la estructura

El siguiente ejemplo lee un archivo csv con personas y sus edades y encuentra la mayor:

```
import Text.CSV

main :: IO ()
main = do
  let fileName = "input.csv"
  input <- readFile fileName

  let csv = parseCSV fileName input
  either handleError doWork csv

  handleError csv = putStrLn "error parsing"
  doWork csv = (print.findOldest.tail) csv
```

Para un ejemplo concreto, mostraría lo que vemos a continuación:

```
$ runhaskell Main.hs
["Becca", "23"]
```

Manejo de CSV en Haskell

Primer ejemplo - Funciones adicionales

En el código anterior falta definir `findOldest`:

```
findOldest :: [Record] -> Record
findOldest [] = []
findOldest = foldl1 (\a x -> if age x > age a then x else a)

age [a,b] = toInt a

toInt :: String -> Int
toInt = read
```

Esta función recibe la lista de registros y mediante un plegado por la izquierda se va quedando en cada momento con el que tenga mayor edad entre el acumulador, `a`, y el elemento actual, `x`.

Como sabemos que el registro consta de dos campos, y el segundo de ellos representa la edad, con `age` obtenemos la edad de la persona contenida en el registro, y para comparar sus edades pasamos la cadena a entero (`toInt`).

Controlando errores de parsing

Tras leer el contenido del fichero con `readFile` y sacarlo del contexto de `IO` hacia `input` (de tipo `String`), pasamos a la función de *parsing*:

- El nombre del archivo (`fileName`, solo para referenciarlo al presentar errores).
- El contenido (`input`), cadena que se va a analizar y de la que tratar de obtener los registros.

```
let csv = parseCSV fileName input
either handleError doWork csv

handleError csv = putStrLn "error parsing"
doWork csv = (print.findOldest.tail) csv
```

Como vemos, la primera instrucción devuelve a `csv` el resultado del `parseCSV`. ¿De qué tipo nos queda `csv`?

- El tipo es `Either Text.Parsec.Error.ParseError CSV`

Bibliografía

Como adelantamos desde la portada de este conjunto de diapositivas, el contenido está basado en la fuente original:

 N. Shukla. *Haskell Data Analysis Cookbook*. Packt Publishing, 2014.