

Prácticas de laboratorio de LTP (Parte II : Programación Funcional)

Práctica 4: Introducción a Haskell



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Jose Luis Pérez
jlperez@dsic.upv.es

Introducción : Lenguaje Haskell

- ✓ Haskell es un lenguaje de programación **puramente funcional**. A diferencia de los lenguajes de programación imperativos tradicionales (C, C++, Java, C#, etc.)
- ✓ En los lenguajes funcionales se hace hincapié en el **que**, mientras que en los lenguajes imperativos se hace hincapié en el **como**.
- ✓ En los lenguajes puramente funcionales, como Haskell:
 - Las **funciones no tienen efectos colaterales**, por lo que la ejecución de las mismas no puede alterar elementos globales.
 - De hecho, ante un mismo argumento de entrada, **devuelven siempre el mismo resultado**.
- ✓ Además, Haskell es un lenguaje **fuertemente tipado**, que resuelve la asignación de tipos estáticamente.

1. Objetivo de la práctica

El objetivo de esta practica es introducir el lenguaje Haskell y presentar las facilidades básicas del entorno **GHCi**, que es la versión interactiva del Glasgow Haskell Compiler (**GHC**). Utilizaremos **GHCi** en la primera practica y combinaremos su uso con **GHC** en las dos practicas siguientes.

La práctica tiene dos objetivos principales:

- 1) Utilización del entorno GHCi para crear y ejecutar programas
- 2) Conocer los fundamentos de programación del lenguaje Haskell

Nota: En Poliformat se dispone de un enlace a un libro de Haskell en castellano. Y también el fichero **codigoEnPdf_P4** que podéis utilizar para copiar y pegar los ejemplos que se presentan durante la sesión.

2. Aprendiendo a utilizar GHCi

Utilizaremos en esta primera sesión la versión interactiva del **GHC** invocando **ghci** :

```
jlperez@EVIPL-022-OK:~$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude>
```

La mayoría de comandos en **GHCi** empiezan con “:”, seguido de uno o mas caracteres. Es importante recordar dos comandos de uso frecuente:

```
prelude> :q  Salir de GHCi
prelude> :?  Mostrar la lista de todos los comandos disponibles
```

Y estos algunos de los comandos mas habituales. Cualquiera de ellos se puede abreviar por ejemplo **:lo** o **:l** para **:load**.

```
:load <modulename> - loads the specified module
:reload - reloads the current module
:type <expression> - print type of the expression
:info <function> - display information about the given names
:quit - exit the interpreter
:help - shows available commands
```

2.1. Evaluando expresiones con GHCi

Podemos teclear expresiones Haskell directamente en la línea de entrada de comandos basta con escribirlo y a continuación pulsar intro. GHCi evaluará la expresión y mostrará el resultado.

```
Prelude> 2+3*8  
26  
Prelude>
```

Otras funciones aritméticas se escriben en notación *prefija*, como **div** o **mod** que calculan, respectivamente, el cociente y el resto de la división entera:

```
Prelude> div 15 2  
7  
Prelude> mod 12 5  
2
```

Sin embargo, las funciones *infijas* se pueden utilizar en notación *prefija* usando paréntesis:

```
Prelude> (+) 2 3  
5
```

Por otra parte, las funciones *prefijas* se pueden utilizar en notación *infija* usando el acento hacia la izquierda:

```
Prelude> 15 `div` 2  
7  
Prelude> 12 `mod` 5  
2
```

2.1. Evaluando expresiones con GHCi

Es posible preguntar por el tipo de una expresión (comando **:type** que se puede abreviar por **:t**).

```
Prelude> True && False
False
Prelude> :t True && False
True && False :: Bool
```

En el segundo caso, **GHCi** nos indica que la expresión se evalúa a un tipo lógico (**Bool**). Observa que el símbolo **::** sirve para indicar que la expresión a su izquierda es del tipo que aparece a su derecha.

Podemos utilizar los operadores de comparación para evaluar expresiones lógicas:

```
Prelude> 5 == 5
True
Prelude> 5 /= 5
False
```

Podemos crear una variable **a** en el entorno interactivo y después referenciarla dentro de una expresión.

```
Prelude> let a = 7
Prelude> succ a
8
```

La aplicación de funciones siempre tiene máxima prioridad. Se deben utilizar paréntesis cuando haya posible ambigüedad.

```
Prelude> succ (a * 5)
41
Prelude> succ a * 5
45
```

2.2. Edición y carga de programas

Definir un módulo:

```
module Signum where
```

La primera letra del nombre del módulo debe ser MAYÚSCULA.

El nombre del fichero debe ser el mismo que el nombre del módulo.

Signum.hs

Definir una función dentro del módulo:

```
module Signum where
  -- Definición de la función signum' (signo):
  signum' :: Int -> Int
  signum' x = if x < 0 then -1 else
              if x == 0 then 0 else 1
```

signum' es una función que toma un entero como argumento de entrada y devuelve otro entero

La función **signum'** recibe su valor para un parámetro de entrada **x**

Ambas definiciones deben tener la misma indentación. En este caso dos espacios

2.2. Edición y carga de programas

Carga de un módulo

```
Prelude> :l Signum
[1 of 1] Compiling Signum          ( Signum.hs, interpreted )
Ok, one module loaded.
```

A continuación indicamos algunas invocaciones sencillas a la función **signum'**.

```
*Signum> signum' 0
0
*Signum> signum' (-10)
-1
*Signum> 
```

Observa ahora que en el prompt pone ***Signum>**. solo se pueden evaluar expresiones que contengan funciones predefinidas, o definidas en el módulo **Signum**, o en algún módulo importado por el módulo actual, aunque **Prelude** se importa siempre por defecto.

Añadir otra función al módulo:

Signum'' es una nueva versión de la función **Signum'**. Debe tener la misma indentación (igual que todas las funciones definidas en este módulo)

En este caso se utiliza el carácter **|** para expresar la selección múltiple, en vez de **if**

```
module Signum where

signum' :: Int -> Int
signum' x = if x < 0 then -1 else
             if x == 0 then 0  else 1

signum'' :: Int -> Int
signum'' x
  | x < 0    = -1
  | x == 0   = 0
  | otherwise = 1
```


2.3. Mensajes de error y alertas

GHCi informa de posibles errores sintácticos y de tipos durante la carga de un fichero. Por ejemplo, si se abre el editor y se escribe el programa:

```
module Hello where
  hello n = concat (replicate n 'hello')
```

Se salva en fichero **Hello.hs**, y se carga a continuación en **GHCi** utilizando el comando **:load**, se produce el siguiente mensaje de error:

```
Prelude> :l Hello.hs
[1 of 1] Compiling Hello
```

```
Hello.hs:2:33: error:
```

- Syntax error on 'hello'
Perhaps you intended to use TemplateHaskell or TemplateHaskellQuotes
- In the Template Haskell quotation 'hello'

```
2 | hello n = concat (replicate n 'hello')
```

```
Failed, no modules loaded.
```

Concatena **n** replicas de la cadena de texto **"hello"**. Pero hay un error sintáctico, ya que las cadenas de texto usan **"**

Si corregimos el error reemplazando las comillas simples por dobles, es decir, **"hello"**. Podemos cargar de nuevo el programa utilizando ahora el comando **:r**, abreviatura del comando **:reload** que carga de nuevo el ultimo fichero.

2.3. Mensajes de error y alertas

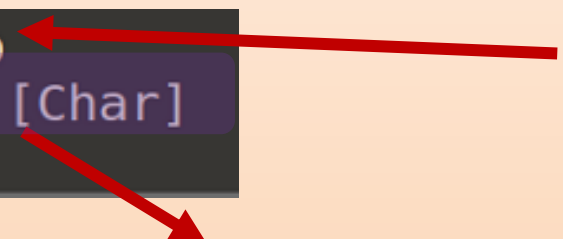
Y ahora podemos ejecutar la función **hello**:

```
*Hello> :r
Ok, one module loaded.
*Hello> hello 5
"hello hello hello hello hello "
*Hello>
```

```
module Hello where
  hello n = concat (replicate n 'hello')
```

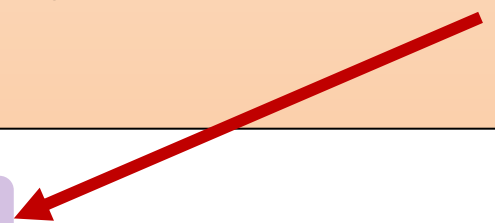
El perfil de las funciones indica su tipo. Si no se escribe, el compilador intenta deducirlo o inferirlo. Para consultar el tipo inferido automáticamente por el interprete para la función **hello** se utiliza el comando **t** (type) :

```
*Hello> :t hello
hello :: Int -> [Char]
*Hello>
```



GHCi ha inferido que el tipo de entrada debe ser **Int** y el de salida una cadena de caracteres **[Char]**, y es por ello que no se han producido errores de compilación, pero una buena práctica es escribir explícitamente el perfil de cada función.

```
module Hello where
  hello :: Int -> [Char]
  hello n = concat (replicate n "hello ")
```



2.3. Tipos

Haskell es un lenguaje fuertemente tipado. La comprobación de tipos se realiza en tiempo de compilación. Abrase el editor y escríbase el siguiente programa en un fichero con nombre **Typeerrors.hs**:

```
module Typeerrors where
  convert :: (Char, Int) -> String
  convert (c,i) = [c] ++ show i

  main = convert (0, 'a')
```

en Haskell se pueden definir tuplas de elementos de cualquier longitud simplemente usando los paréntesis y la coma. En este caso el tipo de datos **(Char, Int)** y el patrón **(c,i)**.

La función **show** sirve para convertir a cadena (tipo **String** o también **[char]**)

[c] convierte el elemento de tipo **Char** en una cadena (con un solo Char) y **++** concatena cadenas

```
*Hello> :l Typeerrors
[1 of 1] Compiling Typeerrors          ( Typeerrors.hs, interpreted )
```

```
Typeerrors.hs:5:21: error:
```

- Couldn't match expected type 'Int' with actual type 'Char'
- In the expression: 'a'
- In the first argument of 'convert', namely '(0, 'a')'
- In the expression: convert (0, 'a')

```
5 | main = convert (0, 'a')
```

```
Failed, no modules loaded.
```

```
Prelude> 
```

El mensaje de error indica que la expresión **(0, 'a')** es de tipo **(Int, Char)** cuando se esperaba una expresión de tipo **(Char, Int)**

2.3. Tipos

El problema se soluciona cambiando el orden de los argumentos de la llamada a la función **convert**.

```
convert :: (Char, Int) -> String
convert (c,i) = [c] ++ show i

main = convert ('a',0)
```

Nótese que los elementos de tipo **Char** se expresan con comillas simples, mientras que los de tipo **String** o **[Char]** con comillas dobles

Es posible definir funciones directamente en **GHCi** precediendo la definición con **let**, cuyo uso en general en Haskell se describe en el material de lectura previa:

```
Prelude> let convert' (c,i) = [c] ++ show i
Prelude> convert' ('c',3)
"c3"
Prelude>
```

Otra versión de la función **convert** podría disponer de dos argumentos de entrada simples, en vez de una tupla:

```
convert' :: Char -> Int -> String
convert' c i = [c] ++ show i

main' = convert' 'a' 0
```

Dos argumentos de entrada, **Char** e **Int**

Los argumentos no se ajustan al patrón de la tupla.

Ahora hay dos argumentos sin paréntesis, y sin comas

Precedencias en los operadores funcionales

Hay que tener en cuenta la precedencia de los operadores, teniendo en cuenta que en una expresión funcional se aplica la asociatividad a izquierdas...

```
Prelude> :r
[1 of 1] Compiling Typeerrors ( Typeerrors.hs )
Ok, one module loaded.
*Typeerrors> convert ('a',succ 0)
"a1"
*Typeerrors> convert' 'a' succ 0
<interactive>:8:1: error:
• Couldn't match expected type 'Integer -> [Char]'
  with actual type '[Char]'
• The function 'convert' is applied to three arguments,
  but its type 'Char -> Int -> [Char]' has only two
  In the expression: convert' 'a' succ 0
  In an equation for 'it': it = convert' 'a' succ 0
• Relevant bindings include it :: t (bound at <interactive>:8:1)
```

La invocación no da problemas. Primero se evalúa **succ**, y luego **convert**

La invocación es errónea ya que Haskell intentará aplicar primero la función de la izquierda (**convert'**), tomando **succ** como segundo argumento (que no es el esperado, **Int**)

En es caso es necesario utilizar los paréntesis para que el tipo del segundo argumento y el número de los mismos sea el esperado:

```
*Typeerrors> convert' 'a' (succ 0)
"a1"
*Typeerrors> 
```

Curricación y aplicación parcial

Otra diferencia importante entre **convert** con **convert'** es que esta última está curricada, mientras que la primera no. Esto permite la aplicación parcial de la función **convert'**...

```
convert :: (Char, Int) -> String
```

```
convert' :: Char -> (Int -> String)
```

h

La función **h** es el resultado de la aplicación parcial de **convert'**. Vemos que el operador **->** es asociativo a derechas.

```
*Typeerrors> let h = convert' 'a'  
*Typeerrors> h 1  
"a1"  
*Typeerrors> h 3  
"a3"  
*Typeerrors> :t h  
h :: Int -> String  
*Typeerrors>
```

La función **h** solo tiene un argumento de entrada, y añade el número a "a"

Este tipo de aplicación parcial se puede realizar sobre las operaciones aritméticas:

```
*Typeerrors> let f = (+) 5  
*Typeerrors> let f2 = (*2)
```

```
*Typeerrors> f 5  
10  
*Typeerrors> f2 4  
8  
*Typeerrors>
```

Ajuste de patrones

Un **ajuste de patrones** (*pattern matching*) consiste en una especificación de pautas que deben ser seguidas por los datos, los cuales pueden ser deconstruidos permitiéndonos acceder a sus componentes.

Permite separar el cuerpo que define el comportamiento de una función en varias partes, de forma que el código quede mucho más elegante, limpio y fácil de leer. Veamos un ejemplo, la función **sayme**:

```
sayme :: Int -> String
sayme 1 = "¡Uno!"
sayme 2 = "¡Dos!"
sayme 3 = "¡Tres!"
sayme 4 = "¡Cuatro!"
sayme 5 = "¡Cinco!"
sayme x = "No entre uno 1 y 5"
```

Los patrones son
verificados de arriba a
abajo

El ajuste de patrones nos puede evitar en muchos casos la utilización de sentencias condicionales..

```
sayme' x
| x==1 = "¡Uno!"
| x==2 = "¡Dos!"
| x==3 = "¡Tres!"
| x==4 = "¡Cuatro!"
| x==5 = "¡Cinco!"
| otherwise = "No entre uno 1 y 5"
```

3. Ejercicios: ejercicios resueltos (I)

1. Escribir una función **nextchar** que tome como argumento un carácter y devuelva el carácter siguiente (según la codificación que se utilice).

```
module Nextchar where
  import Data.Char
  nextchar :: Char -> Char
  nextchar c = chr ((ord c) + 1)
```

Se debe importar **Data.Char** para poder utilizar las funciones **chr** y **ord**

2. Escribir una función **fact** para calcular el factorial de un número entero no negativo

```
module Fact where
  fact :: int -> int
  fact 1 = 1
  fact n = n*fact(n-1)
```

El *pattern matching* se suele utilizar en los algoritmos recursivos para separar el caso base del caso general

Y la recursividad es muy utilizada en la programación declarativa y concretamente en la programación funcional

3. Ejercicios: ejercicios resueltos (y II)

3. Veamos otro ejemplo de función recursiva. En este caso la función **power1** tiene 2 argumentos y calcula la potencia de un número entero:

```
module Power where
  power1 :: Int -> Int -> Int
  power1 _ 0 = 1
  power1 x n = x * power1 x (n-1)
```

El carácter `_` representa una variable cuyo nombre es irrelevante en el caso base de la recursión, ya que no se utiliza en la parte derecha de la igualdad

El uso del paréntesis es necesario

4. La función **power2** devuelve el mismo resultado que **power1** pero de una manera más eficiente:

```
power2 :: Int -> Int -> Int
power2 _ 0 = 1
power2 x n
  | even n = power2 (x * x) (div n 2)
  | otherwise = x * power2 (x * x) (div n 2)
```

En este caso se hace uso de *pattern matching* y también de un selector múltiple, para determinar si la potencia es par (función **even**)

3. Ejercicios: ejercicios a resolver (I)

Se recomienda crear un módulo **Practica4** donde incluir todas las funciones relacionadas con los ejercicios que se proponen seguidamente:

1. Escribir una función **numCbetw2** que devuelva cuantos caracteres hay entre dos caracteres dados (sin incluirlos). Por ejemplo:

No es necesario utilizar recursividad para resolverlo, se debe utilizar la función **ord**

```
> numCbetw2 'a' 'c'  
1  
> numCbetw2 'e' 'a'  
3  
> numCbetw2 'a' 'b'  
0  
> numCbetw2 'x' 'x'  
0
```

2. Escribir una función recursiva, **addRange**, que devuelva el sumatorio desde un valor entero hasta otro (incluyendo ambos). Por ejemplo:

En este caso no se puede utilizar el ajuste de patrones para especificar el caso base, hay que utilizar **if**

```
> addRange 5 5  
5  
> addRange 5 10  
45  
> addRange 10 5  
45
```

3. Ejercicios: ejercicios a resolver (II)

3. Definir una función binaria (con dos argumentos) **max'** que devuelva el mayor de sus dos argumentos. Por ejemplo:

```
> max' 5 50  
50  
> max' 10 1  
10
```

4. Escribir una función **leapyear** que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4. Sin embargo, no lo son los múltiplos de 100, a excepción de los múltiplos de 400 que si que lo son.

```
> leapyear 1992  
True  
> leapyear 1900  
False
```

Se puede utilizar el resto de la división entera (función **mod**) para comprobar si un número es múltiplo de otro. Recuerda que este método se invoca de forma *prefija* lo que quiere decir que se invoca: **mod 4 2** o **4 `mod` 2** (recuerda que para convertir una función prefija en infija debes utilizar la tilde invertida)

3. Ejercicios: ejercicios a resolver (y III)

5. Escribir una función **daysAmonth** que calcule el numero de días de un mes, dados los valores numéricos del mes y año. Considerar los años bisiestos para febrero. Por ejemplo, 1800 no fue bisiesto mientras que el año 2000 si que lo fue.

Se debe utilizar la selección múltiple y la función **leapyear**

```
> daysAmonth 2 1800
28
> daysAmonth 2 2000
29
> daysAmonth 10 2015
31
```

6. Escribir una función **remainder** que devuelva el resto de la división de dos enteros no negativos, divisor distinto de 0, usando sustracciones, es decir: sin utilizar las funciones **div**, **mod**, **rem**,.

```
> remainder 20 7
6
```

De nuevo se debe de utilizar la recursividad

7. Usando la función factorial definida previamente, escribir una definición de la función **sumFacts** tal que calcule la suma de los factoriales hasta **n**, es decir, **sumFacts n = fact 1 + ... + fact n**.

```
> sumFacts 5
153
```