MODELOS FORMALES DE COMPUTACIÓN

Introducción a Haskell

---

1. Cómo usar GHC

GHCi es el entorno interactivo del GHC (*Glasgow Haskell Compiler System*), en el que las expresiones Haskell se evalúan interactivamente y los programas son interpretados. El entorno GHCi viene con el compilador GHC, es de dominio público y se distribuye bajo licencia GNU. GHC suele estar disponible como paquete para ser instalado en la mayoría de las distribuciones Linux. También está disponible para Windows y Mac.

En los PC's de los laboratorios del DSIC, GHCi opera sobre Windows. Pueden obtenerse ésta y otras versiones de GHCi vía WWW en la siguiente dirección de internet:

```
http://www.haskell.org/ghc/
```

1.1. Para empezar

El sistema está accesible en el menú de `Inicio` → `GHCi` → `ghci`. Al entrar en el entorno, el intérprete queda a la espera de que introduzcas un comando o una expresión a evaluar. La mayoría de comandos en Haskell empiezan con un ':', seguido de uno o más caracteres. Es importante recordar dos comandos de uso frecuente:

`? :q` Salir de GHC
`? :?` Mostrar la lista de todos los comandos disponibles:

```
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:cd <dir>           - change directorio to <dir>
:load <modulename>  - loads the specified module
:reload             - reloads the current module
:type <expression>  - print type of the expression
```

```
:info <function>     - view function in on-line documentation
:edit [<row:col> ]   - edit the currently loaded module
:jump                - jump editor to last error location
:quit                - exit the interpreter
:help                - shows available commands
:clear               - clear the screen
```

Para interrumpir un proceso en curso, basta utilizar el comando habitual de interrupción `^C`.

A continuación se describen algunas de las características del entorno GHCi. Para una descripción detallada, consultar la documentación on-line.

1.2. **Usando** Haskell

Para evaluar una simple expresión, teclear por ejemplo en la línea de entrada de comandos:

```
2 + 3 * 8
```

seguida de la tecla `RET`. Las funciones aritméticas como `+` y `*` son predefinidas. GHCi evalúa la expresión e imprime su valor en la zona central de la ventana, mostrando de nuevo el *prompt* `Prelude` al finalizar:

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

Ahora podemos preguntar el tipo de esta expresión, tecleando

`:t 2 + 3 * 8`. El intérprete responde:

```
2 + 3 * 8 :: Num a => a
```

indicando que la expresión, cuando se evalúe, dará un valor de tipo numérico, ya que puede interpretarse como un natural, un número entero, etc.

1.3. Edición y Carga de Programas

No existe un editor de texto asociado a GHCi pero muchos editores de texto conocidos soportan el formato Haskell y colorean y disponen de funcionalidades específicas para programas Haskell .

En el nivel más alto, un programa en Haskell es un conjunto de módulos.

Cada módulo contiene una colección de declaraciones, que incluye el nombre del módulo, el tipo de las funciones contenidas en éste y la definición de dichas funciones.

Abre el editor y escribe el siguiente programa:

```
module Hello where

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

Salva el fichero con el nombre `Hello.hs`. En GHCi, el nombre del fichero debe tener la extensión `.hs`. Además, el nombre de cada módulo del programa debe coincidir con el nombre del fichero donde se ha escrito.

Volviendo a la ventana del intérprete GHCi, cargamos el programa almacenado en el fichero `Hello.hs` introduciendo `:load` en la línea de comandos, junto con el nombre del fichero y ruta de acceso.

Nota: Para ahorrarnos escribir la ruta de acceso cada vez, se recomienda al iniciar la sesión en GHCi cambiar el directorio de referencia con el comando `:cd` seguido del *path* del directorio de trabajo.

La carga del fichero produce el siguiente mensaje de error:

```
Hello.hs:3:33:
   lexical error in string/character literal at character 'e'
Failed, modules loaded:none
```

Observa que se muestra la posición exacta del error (fila 3, columna 33). Corrige el error, reemplazando las comillas simples por dobles comillas, es decir, "hello". Intenta cargar de nuevo el programa con el comando `:r`. La compilación no produce errores pero es una buena práctica escribir explícitamente en cada módulo el perfil de las funciones definidas en el mismo. Si quieres, puedes copiar ahora en tu programa el tipo inferido automáticamente por el intérprete para la función `hello`, lo que evitará futuros mensajes de aviso referentes a esto.

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

Nota: las listas de elementos se definen en Haskell usando los corchetes, e.g. `[Char]` para una lista de caracteres, `[Int]` para una lista de números enteros, etc. Además el tipo de datos String es manejado en Haskell como una lista de caracteres, de forma que la expresión `"hello"` se representa realmente como la lista `['h','e','l','l','o']`.

Después de cargar el programa y comprobar que no tiene errores, el intérprete podrá evaluar expresiones que contienen llamadas a las funciones definidas en él. Por ejemplo, evalúa la expresión `hello 10`.

Debes recordar que cada orden `:load` inicializa la base de datos del

intérprete, por lo que sólo se pueden evaluar expresiones que contengan funciones predefinidas o definidas en el módulo actual (en este caso, la función `hello`).

1.4. Tipos

Haskell es un lenguaje fuertemente tipado. La comprobación de tipos se realiza en tiempo de compilación. GHC no sólo es capaz de detectar errores de tipo sino que también puede sugerir posibles formas de resolver los conflictos. Para ilustrar este hecho, en la ventana del editor ConTEXT, escribe el siguiente programa:

```
module Errortipos where

main = f (0,'a')

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

y sálvalo con el nombre `Errortipos.hs`. Una vez de vuelta en el intérprete, carga el fichero. Como ves el intérprete nos da el siguiente mensaje de error

```
Errortipos.hs:3:13:
    Couldn't match expected type 'Int' against inferred type 'Char'
    In the expression: 'a'
    In the first argument of 'f', namely '(0, 'a')'
    In the expression: f (0, 'a')
Failed, modules loaded: none.
```

que nos indica que la expresión `(0, 'a')` es de tipo `(Int , Char)` cuando se esperaba una expresión de tipo `(Char , Int)` de acuerdo al tipo definido en el programa para la función `f`. Una posible solución al problema es cambiar el orden de los elementos `0` y `'a'`.

1.5. Definición de Funciones

La definición de una función `f` consta en general de la declaración de su perfil o tipo (que ya hemos visto y comentado que es opcional ya que GHC infiere los tipos automáticamente, aunque es recomendable incluir la declaración en el programa) y un número de ecuaciones. Por ejemplo, el esquema general puede ser de la forma:

```
f :: Type1 -> ... -> Typen -> Typef
f (patron1) ... (patronn) = expresion
```

Cada una de las expresiones $\texttt{patron}_i$ representa un argumento de la función y se conoce como 'patrón'. Un patrón sólo puede contener constructores o variables, no puede contener funciones definidas. Los nombres de las variables, de los constructores de datos y de las funciones se escriben en minúsculas.

Es posible omitir los paréntesis en las definiciones de función, siempre que no haya ambigüedad y se ajuste a la definición del tipo de la función. Por ejemplo, se puede introducir la siguiente definición para la función que calcula la longitud de una lista:

```
module Long where

long [] = 0
long (x:t) = 1 + long t
```

Ahora, una vez cargado este programa en el intérprete podemos pedir que se evalúe la expresión `long ([1,2,3])` devolviéndonos el valor `3`. En este caso podemos también escribir la misma expresión sin los paréntesis, `long [1,2,3]`, ya que no hay ambiguedad por tener `long` un único argumento. De esta forma obtendremos la misma respuesta.

Si se quiere definir una función usando una ecuación condicional, la notación a usar es ésta:

$$\begin{array}{l} \texttt{f } x_1\ x_2\ \ldots x_n \\ \qquad\qquad \mid \mathit{condicion}_1 \ = \ \mathit{exp}_1 \\ \qquad\qquad \mid \mathit{condicion}_2 \ = \ \mathit{exp}_2 \\ \qquad\qquad \vdots \\ \qquad\qquad \mid \mathit{condicion}_m \ = \ \mathit{exp}_m \end{array}$$

Por ejemplo, para la función potencia, podemos definir la siguiente función:

```
module Power1 where

power1  :: Int -> Int -> Int
power1 _ 0 = 1
power1 n t = n * power1 n (t - 1)
```

aunque una versión más eficiente es:

```
module Power2 where

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

donde `even` y `div` son funciones predefinidas y denota cualquier valor (del tipo correspondiente).

Otras expresiones que podemos usar en la definición de lenguajes son las siguientes.

where. Usamos la expresión (ecuación) `where` cuando queremos definir

una expresión local a una función.

```
f x1 x2 ...xn = exp
     where
        definicionFuncion1
        ...
        definicionFuncionm
```

Ejemplos de uso:

```
f x y = g (a+1)  (a+2)
      where
        a = (x + y) / 2
        g x y = x * y

f x y = g (a+1)  (a+2)
      where
        a = (x + y) / 2 ; g x y = x * y
```

let. Usamos la expresión `let` con el mismo fin. La sintaxis general es la siguiente:

```
f x1 x2 ...xn =
     let   definicionFuncion1
           ...
           definicionFuncionm
     in exp
```

Ejemplo de uso:

```
f = let a=3+2
    in a*a*a
```

1.6. Estrategia de Reducción

La estrategia de reducción en Haskell es *lazy* (perezosa). Esta estrategia reduce una expresión (parcialmente) sólo si realmente es necesario para calcular el resultado. Es decir, se reducen los argumentos sólo lo suficiente para poder aplicar algún paso de reducción en el símbolo de función más externo.

Gracias a esto, es posible trabajar con estructuras de datos infinitas.

En el `prelude` están definidas algunas funciones que devuelven listas infinitas, como la función `repeat`:

```
repeat :: a -> [a]
repeat x = xs where xs = x:xs
```

La llamada `repeat 3` devuelve la lista infinita
`[3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,....`

Una lista infinita que es generada por `repeat`, puede ser usada como resultado parcial por una función que tiene un resultado finito. La siguiente

función, por ejemplo, toma un número finito de elementos de una lista:

```
toma :: Int -> [a] -> [a]
toma 0 _ = []
toma n (x:t) = x : toma (n - 1) t
```

Por ejemplo, la llamada `toma 3 (repeat 4)` devuelve la lista `[4,4,4]`.

2. Tipos de datos simples

Existe una colección de tipos de datos, funciones y operadores que pueden usarse en cualquier programa. Éstos son los elementos predefinidos del lenguaje, que se encuentran en el módulo `Prelude`. Para importar un módulo del Prelude, por ejemplo `Char` definido más abajo, hay que escribir

```
import Data.Char
```

Todas las funciones pre-definidas (a excepción de las aritméticas) son, en un principio, prefijas, aunque también es posible usarlas en notación infija.

1. **El tipo** `Bool`

   Los valores de este tipo representan expresiones lógicas cuyo resultado puede ser verdadero o falso. Sólo hay dos valores constantes para este tipo: `True` y `False` (escritos de esta forma), que representan los dos resultados posibles.

   Funciones y Operadores

   Los siguientes operadores y funciones predefinidos operan con valores booleanos:

   `(&&) :: Bool -> Bool -> Bool` es la conjunción lógica

   `(||) :: Bool -> Bool -> Bool` es la disyunción lógica

   `not :: Bool -> Bool` es la negación lógica

   `(==) :: Bool -> Bool -> Bool` devuelve `True` si el primer argumento es igual al segundo, y `False` si son distintos.

   `(/=) :: Bool -> Bool -> Bool` devuelve `True` si el primer argumento no es igual al segundo argumento, y `False` si son iguales.

2. **El tipo** `Int`

   Los valores de este tipo son números enteros de precisión limitada. Los valores constantes de este tipo se escriben con la notación habitual.

   Funciones y Operadores

   Algunas de las funciones y operadores definidos para este tipo son:

   `(+), (-), (*) :: Int -> Int -> Int` son la suma, resta y producto de enteros respectivamente

`(^) :: Int -> Int -> Int` es el operador potencia. El exponente deberá ser un natural

`div, mod :: Int -> Int -> Int` son el cociente y resto de dividir dos enteros respectivamente

`abs :: Int -> Int` es el valor absoluto

`signum :: Int -> Int` devuelve `1`, `-1` ó `0` repectivamente, según sea el signo del argumento entero

`(==) :: Int -> Int -> Bool` devuelve `True` si el primer argumento es igual al segundo, y `False` si son distintos.

`(/=) :: Int -> Int -> Bool` devuelve `True` si el primer argumento no es igual al segundo argumento, y `False` si son iguales

`even, odd :: Int -> Bool` comprueban la paridad (par o impar) de un número entero

3. **El tipo** `Float`

   Los valores de este tipo representan números reales. Hay dos modos de escribir valores reales:

   notación habitual: por ejemplo `1.35`, `-1.0`, ó `1.`

   notación científica: por ejemplo `1.5e3` (que denota el valor 1.5×10^3, ó `1.5e-7`

   Funciones y Operadores

   Algunas de las funciones y operadores definidos para este tipo son los siguientes. Nótese que en algunos entornos de Haskell, como en Helium, no se permite la sobrecarga de operadores y la notación para los operadores es distinta para los enterios y para los reales. Esto no ocurre en otros compiladores/intérpretes de Haskell como GHC.

   `(+), (-), (*), (/) :: Float -> Float -> Float` que son la suma, resta, producto y división de reales respectivamente

   `(==) :: Float -> Float -> Bool` devuelve `True` si el primer argumento es igual al segundo y `False` si no lo es

   `(/=) :: Float -> Float -> Bool` devuelve `True` si el primer argumento es distinto al segundo, y `False` si son iguales

   `sqrt :: Float -> Float` devuelve la raíz cuadrada de un real

   `(^) :: Float -> Int -> Float` devuelve la potencia de base real y exponente entero

   `(**) :: Float -> Float -> Float` devuelve la potencia con base y exponente reales

   `truncate :: Float -> Int` devuelve la parte entera de un real

   `signumFloat :: Float -> Int` devuelve `1`, `-1` ó `0` según el signo del número real

4. **El tipo** `Char`

   Un valor de tipo `Char` representa un caracter (letra, dígito,...). Un valor constante de tipo caracter se escribe entre comillas simples (por ejemplo `'a'`,`'9'`,...).

   Funciones y Operaciones

   Algunas de las funciones definidas para este tipo son:

   `ord :: Char -> Int` devuelve el código ASCII correspondiente al caracter argumento.

   `chr :: Int -> Char` es la función inversa a `ord`.

   `isUpper, isLower, isDigit, isAlpha :: Char -> Bool` comprueban si el carácter argumento es una letra mayúscula, minúscula, un dígito o una letra, respectivamente.

   `toUpper, toLower :: Char -> Char` convierten la letra que toman como argumento en mayúscula o minúscula, respectivamente.

   `eqChar :: Char -> Char -> Bool` devuelve `True` si los dos argumentos son iguales y `False` en caso contrario.

3. Ejercicios a resolver sobre tipos básicos

1. Escribir una función `siguienteLetra` que tome como argumento una letra del alfabeto y devuelva la letra que le sigue. Suponer que la letra 'A' sigue a la 'Z' (tanto en mayúsculas como en minúsculas).

2. Escribir una función recursiva que devuelva el sumatorio desde un valor entero hasta otro.

3. Escribir una función recursiva que devuelva el producto desde un valor entero hasta otro.

4. Define una función binaria `maximo` que devuelve el mayor de sus dos argumentos.

5. Define una función `fact` para calcular el factorial de un número.

6. Usando la función anterior da una definición de la función `sumaFacts` tal que calcule la suma de los factoriales hasta un número n, es decir, `sumFacts` n = `fact 0 + fact 1 + . . . +` `fact` n.

4. El tipo `lista`

En programación funcional es posible emplear tipos estructurados cuyos valores están compuestos por objetos de otros tipos. Por ejemplo, el tipo lista `[a]` puede servir para aglutinar en una única estructura objetos del *mismo* tipo (denotado, en este caso, por la variable de tipo `a`, que puede

instanciarse a cualquier tipo). En Haskell , las listas pueden especificarse encerrando sus elementos entre corchetes y separándolos con comas:

```
[1,2,3] :: [Int]
['a','b','c','d'] :: [Char]
[cos,log,(1.0 +.)] :: [Float -> Float] -- Lista de funciones de
                -- reales en reales: coseno, logaritmo en base 2,
                -- incrementar una unidad un n\'umero real.
```

Sin embargo, las listas

```
[1,'a',2]
['a',log,3]
[cos,2,(*)]
```

no son válidas (¿por qué?).

La lista vacía se denota como `[]`. Cuando no es vacía, podemos descomponerla usando una notación que separa el elemento inicial de la lista que contiene los elementos restantes. Por ejemplo:

```
1:[2,3]
'a':['b','c','d']
cos:[log]
```

Los tipos que incluyen variables de tipo en su definición (como el tipo lista) se denominan tipos genéricos o *polimórficos*. Podemos definir funciones sobre tipos polimórficos, que pueden emplearse sobre objetos de cualquier tipo que sea una instancia de los tipos polimórficos involucrados. Por ejemplo, la función (predefinida) `length` calcula la longitud de una lista:

```
> :t length
length :: [a] -> Int
>
```

La función `length` puede emplearse sobre listas de cualquier clase:

```
> length [1,2,3]
3
> length ['a','b','c','d']
4
> length [doble,cuadruple,fact]
3
> length [2..10]
9
```

4.1. Las funciones `map` y `filter`

Se trata de dos funciones predefinidas útiles para operar con listas. La función `map` aplica una función a cada elemento de una lista. Es una típica

función de las llamadas *de orden superior*, al aceptar, como primer argumento, no valores cualesquiera sino *funciones*. Por ejemplo,

```
> map cuadrado [9,3]
[81,9]
>
> map (<3) [1,2,3]
[True,True,False]
>
```

donde `cuadrado` es la función que calcula el cuadrado de un número entero. La definición de `map` es

```
map          :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

La función `filter` toma una función booleana `p` y una lista `xs` y devuelve la sublista de `xs` cuyos elementos satisfacen `p`. Por ejemplo

```
> filter even [1,2,4,5,32]
[2,4,32]
```

La definición de `filter` es

```
filter          :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) = if p x then x:filter p xs else filter p xs
```

4.2. Las listas intensionales

Haskell proporciona una notación alternativa para las listas, las llamadas *listas intensionales*. He aquí un ejemplo

```
> [x * x | x <- [1..5], odd x]
[1,9,25]
>
```

La expresión se lee: la lista de los cuadrados de los números impares en el rango de 1 a 5.

Formalmente, una lista intensional es de la forma `[e|Q]`, donde `e` es una expresión y `Q` es un *cualificador*. Un cualificador es una secuencia, posiblemente vacía, de *generadores* y *guardas* separados por comas. Un generador toma la forma `x <- xs`, donde `x` es una variable o tupla de variables, y `xs` es una expresión de tipo lista. Una guarda es una expresión booleana. El cualificador `Q` puede ser vacío, en cuyo caso escribimos simplemente `[e]`. He aquí algunos ejemplos:

```
> [(a,b) | a <- [1..3], b <- [1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
>
> [(a,b) | b <- [1..2], a <- [1..3]]
```

```
[(1,1),(2,1),(3,1),(1,2),(2,2),(3,2)]
>
```

Los últimos generadores pueden depender de variables introducidas por los precedentes, como en

```
> [(i,j) | i <- [1..4], j <- [i+1..4]]
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
>
```

Podemos intercalar libremente generadores y guardas:

```
[(i,j) | i <- [1..4], even i, j <- [i+1..4], odd j]
[(2,3)]
```

5. Ejercicios a resolver sobre tipos listas

1. Dada una lista de pares, define una función que devuelva la lista resultante de sumar cada par.
2. Define dos funciones que devuelvan el primer y último elemento de una lista.
3. Define una función para calcular la lista de divisores del número `n`. Utiliza listas intensionales.
4. Define una función para determinar si un entero pertenece a una lista de enteros.
5. Define una función que reemplace en una lista todas las ocurrencias de un elemento `n` por otro elemento `p`.
6. Define una función para contar cuántas veces aparece un elemento en una lista.
7. Define una función que calcule los múltiplos de 5 (intenta dos versiones: la primera usando listas intensionales y la segunda usando orden superior).
8. Define una función que extraiga de una cadena los caracteres escritos en mayúscula.