Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2020-21
Parte II: Programación Funcional

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Práctica 4

Índice

1. Objetivo de la Práctica

El objetivo de esta práctica es introducir el lenguaje Haskell y presentar las facilidades básicas del entorno GHCi, que es la versión interactiva del "The Glasgow Haskell Compiler"(GHC).

Utilizaremos GHCi en la primera práctica y combinaremos su uso con GHC en las dos prácticas siguientes.

2. Aprendiendo a utilizar GHCi

El sistema está instalado en los laboratorios de prácticas y la aplicación está accesible desde cualquier directorio. El intérprete responde escribiendo el comando `ghci` en la terminal Linux:

```
$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

Al entrar en el entorno, el intérprete queda a la espera de que se introduzca un comando o una expresión a evaluar. Por lo general, la autocompleción de expresiones y comandos, mediante tabulador, es posible. La mayoría de comandos en GHCi empiezan con un ':', seguido de uno o más caracteres. Es importante recordar dos comandos de uso frecuente:

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

Algunos de los comandos más habituales se muestran a continuación. Cualquiera de ellos se puede abreviar con el inicio del mismo (basta el primer carácter), p.ej. `: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
```

Para interrumpir el proceso en curso, se puede utilizar, como es habitual, `^C`.

Normalmente tendremos abiertos simultáneamente el GHCi y un editor para crear y modificar nuestros programas que cargaremos y recargaremos en

GHCi. Se recomienda utilizar un editor que reconozca la sintaxis de Haskell (coloreado de comentarios, de palabras reservadas, etc.). En el entorno Linux son varios los editores que asocian a Haskell los ficheros con extensión `.hs`, entre ellos destacamos EMACS (tal y como está configurado en los laboratorios), GEDIT o GEANY.[1]

2.1. Evaluando expresiones con GHCi

Podemos teclear expresiones Haskell directamente en la línea de entrada de comandos. Por ejemplo, para evaluar

```
2+3*8
```

basta con escribirlo y a continuación pulsar intro. GHCi evaluará la expresión y mostrará el resultado. A continuación mostrará de nuevo el *prompt* que, por defecto, salvo que hayamos importado otros módulos, será `Prelude>`.[2]

El resultado en tu terminal debería ser similar a éste:

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

Las funciones aritméticas como `+` y `*` están predefinidas y se escriben en notación *infija*. 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
```

[1] Aunque las prácticas se realizan en Linux, comentar que para Windows está disponible CONTEXT así como las versiones de GEANY y de EMACS.

[2] `Prelude` hace referencia al módulo por defecto que ha cargado inicialmente el sistema y cuyas definiciones se pueden utilizar en la sesión.

Es posible preguntar por el tipo de una expresión (comando `:type` que se puede abreviar por `:t`). Compara, por ejemplo, el resultado de estas dos expresiones:

```
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.

2.2. Edición y carga de programas

Como ya sabes, un programa en Haskell es un conjunto de definiciones de tipos de datos, de funciones, etc.

Prueba a escribir el siguiente programa en un fichero llamado `Signum.hs` . Para GHC y GHCi, el nombre del fichero debe tener la extensión `.hs`. Aunque el nombre de un módulo no tiene por qué coincidir con el nombre del fichero donde se ha escrito, es muy recomendable hacerlo así

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

A continuación carga carga el programa en GHCi de la manera siguiente:

```
Prelude> :load Signum.hs
[1 of 1] Compiling Signum          ( Signum.hs, interpreted )
Ok, modules loaded: Signum.
```

En este caso, el programa `Signum.hs` estaba en la misma carpeta donde estamos ejecutando GHCi.

Como ya se ha dicho, GHCi tiene una función de completado automático usando la tecla tabulador, como en la mayoría de intérpretes de comandos de los sistemas operativos. Por ejemplo, escribiendo `:l S` y pulsando el tabulador, GHCi completa el comando con el nombre completo del fichero. Lo mismo ocurre cuando se evalúan expresiones, donde GHCi puede completar el nombre de una función con un nombre más largo de lo común.

Observa que, una vez cargado el programa, el *prompt* cambia y ahora, en lugar de `Prelude>`, pone `*Signum>`. A continuación indicamos algunas invocaciones sencillas a la función `signum'`.

```
*Signum> signum' 0
```

```
0
*Signum> signum' 100
1
*Signum> signum' (-100)
-1
```

Cada ejecución de la orden `:load` inicializa la base de datos del intérprete, por lo que solo se pueden evaluar expresiones que contengan funciones predefinidas, o definidas en el módulo actual (en este caso, la función `signum'`), o en algún módulo importado por el módulo actual, aunque `Prelude` se importa siempre por defecto.

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, interpreted )

Hello.hs:2:41: parse error on input `)'
Failed, modules loaded: none.
```

Como se ve, el intérprete muestra la posición exacta del error (línea 2, columna 41). Puedes corregir el error reemplazando las comillas simples por dobles, es decir, `"hello"`. Carga de nuevo el programa utilizando ahora el comando `:r`, abreviatura del comando `:reload` que carga de nuevo el último fichero.

Aunque ahora la compilación ya no produce errores, es una buena práctica escribir explícitamente en cada módulo el perfil de las funciones definidas en el mismo. El perfil de las funciones indica su tipo. Si no se escribe, el compilador intenta deducirlo o inferirlo de la definición. Por ejemplo, se puede consultar el tipo inferido automáticamente por el intérprete para la función `hello` de la manera siguiente:

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

lo que indica que la función `hello` recibe un `Int` y devuelve una lista de `Char`. Añadir el perfil a la definición de la función es muy aconsejable y ayuda a detectar errores.

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']`. Las listas de elementos se estudian con detalle en la siguiente práctica.

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

```
*Hello> hello 10
"hello hello hello hello hello hello hello hello hello hello "
```

2.4. Tipos

Haskell es un lenguaje fuertemente tipado. La comprobación de tipos se realiza en tiempo de compilación. GHCi no solo es capaz de detectar errores de tipo sino que también puede sugerir posibles formas de resolver los conflictos. Ábrase 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')
```

Nota: en Haskell se pueden definir tuplas de elementos de cualquier longitud simplemente usando los paréntesis y la coma, p.ej. el tipo de datos `(Char, Int)` y el patrón `(c,i)` en la definición de la función anterior.

Nota: La función `show` sirve para convertir a cadena (tipo `String` o también `[char]`) multitud de tipos de datos (concretamente los de la *clase de tipos* `Show`, todo esto se estudiará en la práctica 6).

Cargando en el intérprete el fichero, se produce el mensaje de error:

```
Typeerrors.hs:6:20:
    Couldn't match expected type `Int' against inferred type `Char'
    In the expression: 'a'
    In the first argument of `convert', namely `(0, 'a')'
    In the expression: convert (0, 'a')
Failed, modules loaded: none.
```

que 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 `convert`. Evidentemente, una posible solución al problema es cambiar el orden de los elementos `0` y `'a'`.

Nota: 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> :t convert
convert :: Show a => (Char, a) -> [Char]
```

3. Ejercicios

3.1. Ejercicios resueltos

Se recomienda que probéis los ejercicios resueltos vosotros mismos.

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)
   ```

   Y su ejecución:

   ```
   *Nextchar> nextchar 'a'
   'b'
   ```

2. Definir una función `fact` para calcular el factorial de un número entero no negativo.

   ```
   module Factorial where
     fact :: Int -> Int
     fact 0 = 1
     fact n = n * fact (n - 1)
   ```

   Y su ejecución

   ```
   *Factorial> fact 3
   6
   ```

3.2. Ejercicios a resolver

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

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

2. Escribir una función recursiva que devuelva el sumatorio desde un valor entero hasta otro (incluyendo ambos). Ejemplo:

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

3. Definir una función binaria (con dos argumentos) `max'` que devuelva el mayor de sus dos argumentos. 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 sí que lo son.

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

5. Escribir una función `daysAmonth` que calcule el número 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 sí que lo fue.

```
> 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`,... Ejemplo:

```
> remainder 20 7
6
```

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. Ejemplo:

```
> sumFacts 5
153
```