Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2019-20
Parte II: Programación Funcional

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

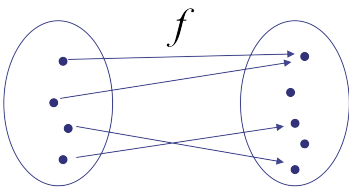
Práctica 4 – Material de lectura previa

Índice

1. Qué es el 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 que se hace hincapié en el *cómo* (o sea, que son lenguajes en los que las soluciones de los problemas se expresan como secuencias de tareas que deben ejecutarse para resolverlos), en los funcionales se hace más hincapié en el *qué*, esto es, en la descripción de los elementos del problema así como de las relaciones entre dichos elementos.

En un lenguaje de programación funcional, se definen funciones basadas en el concepto de función matemática. Recuérdese que las funciones matemáticas se definen entre dos conjuntos de elementos, denominados respectivamente *dominio* y *codominio*.



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 argumento de entrada, devuelven siempre el mismo resultado. Por otra parte, el resultado devuelto por una función solo depende de sus argumentos de entrada (lo que se denomina *transparencia referencial*). Dos funciones son iguales si y solo si devuelven lo mismo ante los mismos argumentos.
- El dominio o el codominio de una función puede contener otra función, o lo que es lo mismo, los argumentos o resultado de una función pueden ser, a su vez, otra función. Idénticamente, una expresión puede evaluar a una función. Por ejemplo:

```
Prelude> let f = (3.5 *)   -- f es "multiplicar por 3.5"
Prelude> f 5               -- se aplica f al argumento 5
17.5
Prelude>
```

- El orden en la definición de las funciones no es relevante (aunque la misma función puede ser especificada, algunas veces, en una serie de casos cuyo orden sí que puede ser relevante).

Además, Haskell es un lenguaje fuertemente tipado, que resuelve la asignación de tipos estáticamente, lo que implica que todas las expresiones y

funciones tienen un tipo asignado durante el proceso de compilación o interpretación. Toda función tiene su *signatura*, perfil, o definición de dominio y codominio. Por ejemplo:

```
example :: String -> Int
```

determina que la función `example` está definida de las cadenas (`String`) en los enteros (`Int`). Haskell, adicionalmente, puede inferir los tipos de las expresiones o las signaturas de las funciones, a partir de cómo están construidas y de los tipos explícitos o implícitos de las subexpresiones que puedan aparecer en las mismas.

Una característica relevante de Haskell es que tiene *evaluación perezosa*[1], lo que quiere decir que retrasará o postergará los cómputos mientras que pueda hacerlo. Por ejemplo, la ejecución de las dos primeras líneas de código Haskell que sigue tiene como resultado lo que se muestra en la tercera:

```
Prelude> let m7 = [x | x <-[1..], x `mod` 7 == 0]
Prelude> take 10 m7
[7,14,21,28,35,42,49,56,63,70]
Prelude>
```

La expresión `[1..]` es un rango que representa en Haskell la lista de todos los enteros a partir del 1. La primera línea utiliza `let` para asignar en el entorno interactivo (el intérprete GHCi) y asocia al identificador `m7` una *lista intensional* que define la lista infinita de todos los múltiplos de 7. Los rangos, las listas y las listas intensionales se estudiarán con más detalle en próximas prácticas.

La expresión `take n ls`, devuelve los primeros `n` elementos de cierta lista `ls`. Como puede verse, el cálculo de la primera línea se ha demorado, en función de lo expresado en la segunda, de forma que tan solo se han generado realmente los elementos iniciales necesarios de una lista infinita.

2. Tipos de datos simples

Existe una colección de tipos de datos, funciones y operadores que pueden usarse en cualquier programa. Son los elementos predefinidos del lenguaje, que se encuentran, en el caso de Haskell, en el módulo `Prelude` del sistema. Todas las funciones predefinidas (a excepción de las aritméticas) son, en un principio, prefijas aunque, tal y como se verá, es posible también usarlas en notación infija. También es posible omitir los paréntesis de los argumentos cuando no haya confusión (por ejemplo, una función como `f (a) (b)` puede también escribirse como `f a b`).

[1] Lazy evaluation.

1. **El tipo `Bool`**

   Los valores de este tipo representan expresiones lógicas cuyo resultado puede ser verdadero o falso. Solo 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[2]. Los valores constantes de este tipo se escriben con la notación habitual $0, 1, -1, 2, -2, \ldots$

   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, respectivamente, según sea el signo del argumento entero.

[2]En Haskell existe también el tipo `Integer`, con precisión no limitada, con las mismas operaciones que el `Int`. Ambos tipos son compatibles entre sí.

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

3. **El tipo `Float`**

   Los valores de este tipo representan números reales con precisión limitada a 7 o 8 dígitos decimales[3]. 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)

   Funciones y Operadores

   Algunas de las funciones y operadores definidos para este tipo son los siguientes.

   - `(+)`, `(-)`, `(*)`, `(/) :: Float -> Float -> Float`. 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.

[3] En Haskell existe también el tipo `Double` de precisión doble. Sin embargo, ambos tipos, `Float` y `Double`, no son compatibles entre sí, por lo que pueden ser necesarias operaciones explícitas de transformación entre ellos.

4. **El tipo `Char`**

   Un valor de tipo `Char` representa un carácter (letra, dígito...). Un valor constante de tipo carácter se escribe entre comillas simples (por ejemplo `'a'`, `'9'`, ...). Para utilizar este tipo de datos, hay que incluir `import Data.Char`, bien en el módulo que se esté definiendo, bien en la sesión GHCi en la que se desee actuar (comando `:module +`).

   Funciones y Operaciones

   Algunas de las funciones definidas para este tipo son:

   - `ord :: Char -> Int`. Devuelve el código ASCII/Unicode correspondiente al carácter 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.
   - `(==) :: Char -> Char -> Bool`. Devuelve `True` si el primer argumento es igual al segundo, y `False` si son distintos.
   - `(/=) :: Char -> Char -> Bool`. Devuelve `True` si el primer argumento no es igual al segundo argumento, y `False` si son iguales.

3. Módulos

Un programa en Haskell es un conjunto de definiciones de tipos de datos, de funciones, etc. Todo esto se organiza como una colección de *módulos*. Sintácticamente hablando, un módulo empieza con la palabra `module` seguida del nombre del módulo (que ha de comenzar por **mayúscula**) y de la palabra `where` que sirve para abrir un bloque, como en el siguiente ejemplo:

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

Una de las formas de crear comentarios en Haskell es utilizar los caracteres `--` que empiezan un comentario hasta el final de la línea, como se ha

mostrado en el ejemplo anterior. También podemos crear comentarios (que, en este caso, pueden abarcar varias líneas) utilizando el símbolo `{-` para abrir el comentario y `-}` para cerrarlo. Observa también que en Haskell no puede haber un `if` sin su correspondiente `else`.

En Haskell se utiliza la indentación para marcar los bloques al igual que en otros lenguajes como Python y a diferencia de otros que, como Java, utilizan las llaves (`{` y `}`) para este propósito.

Para ejemplos sencillos podemos simplemente poner el código Haskell sin definir un módulo (en ese caso, es como haber definido un módulo de nombre `Main`).

4. Definición de Funciones

La definición de una función `f` consta en general de la declaración de su perfil o tipo (aunque es opcional, puesto que GHCi infiere los tipos automáticamente, es recomendable incluirlos en la declaración) y un cierto número de ecuaciones.

En Haskell se identifican los denominados *símbolos constructores* y *símbolos definidos*. Los símbolos definidos son aquellos que disponen de alguna ecuación que defina su evaluación, p.ej. la función `signum'` vista antes. Los símbolos constructores son aquellos que no están definidos por ninguna ecuación. El esquema general para definir una función puede ser de la forma:

```
f :: Type_1 -> ... -> Type_n -> Type_f
f (patron_1) ... (patron_n) = expresion
```

Cada una de las expresiones $\texttt{patron}_i$ representa un argumento de la función y se conoce como *patrón*. Un patrón solo puede contener constructores o variables, no puede contener funciones definidas. Los nombres de las variables y de las funciones se escriben en minúsculas. Nótese la diferencia entre escribir $\texttt{add}(1, 2, 3)$, que es la sintaxis habitual en la mayoría de lenguajes de programación, y `add 1 2 3`, que es la sintaxis funcional.

Es posible omitir los paréntesis de los patrones 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 Length where
  length' [] = 0
  length' (x:t) = 1 + length' t
```

Nota: una lista se puede ver como una secuencia de expresiones conectadas por el símbolo `:` y terminadas con `[]` (que representa la lista vacía), p.ej. la cadena de caracteres `"hello"`, que en realidad es la lista de caracteres `['h','e','l','l','o']`, se puede escribir en Haskell también como `'h':'e':'l':'l':'o':[]`. Las listas de elementos se estudian en detalle en la próxima práctica.

Ahora, una vez cargado este programa en el intérprete, se puede evaluar la expresión `length' ([1,2,3])`, retornando la misma el valor `3`. En este caso es posible también escribir la misma expresión sin los paréntesis: `length' [1,2,3]`, ya que no hay ambiguedad por tener `length'` un único argumento. De esta forma se obtiene la misma respuesta.

Además, la función `length'` admite otros tipos de listas de elementos, p.ej. la expresión `length'"hola"` devuelve `4` y, en cambio, la expresión `length'["hola"]` devuelve `1`.

Se pueden definir funciones usando *ecuaciones condicionales*, o *con guarda*, su forma general es:

$$
\begin{array}{l}
\texttt{f } \texttt{x}_1 \texttt{ x}_2 \texttt{ ... x}_n \\
\qquad\qquad\qquad \texttt{| } condicion_1 \texttt{ = } exp_1 \\
\qquad\qquad\qquad \texttt{| } condicion_2 \texttt{ = } exp_2 \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad \texttt{| } condicion_m \texttt{ = } exp_m
\end{array}
$$

Por ejemplo, para la función potencia, se puede definir la siguiente función:

```
module Power1 where
  power1 :: Int -> Int -> Int
  power1 _ 0 = 1
  power1 n t = n * power1 n (t - 1)
```

donde el símbolo "`_`" representa una variable (como `n` o `t`) pero cuyo nombre es irrelevante. Una versión más eficiente usando ecuaciones condicionales 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 la expresión `otherwise` siempre se evalúa a `True`.

Nota: El orden de aparición de las ecuaciones en el programa es importante ya que Haskell buscará la primera ecuación aplicable de arriba a abajo en el programa y, una vez la haya encontrado, no seguirá probando el resto.

Lo mismo ocurre con las ecuaciones condicionales, donde las condiciones se evalúan de arriba a abajo y, una vez una condición se evalúa a `True`, no se comprueba el resto de condiciones. Por ejemplo, para la función `power2`, si el segundo argumento vale `0`, aplicará la primera ecuación y no intentará aplicar la segunda. En cambio si el segundo argumento es distinto de `0`, la primera ecuación no es aplicable y comprobará la condición `even t` de la segunda ecuación, cuyos argumentos no requieren ningún símbolo constructor. Si dicha condición se evalúa a cierto, aplicará la parte derecha asociada al `even t` y, sino, aplicará la parte derecha del `otherwise`.

Otras formas o expresiones que se pueden utilizar en la definición de funciones en Haskell son las siguientes.

where. Se utiliza la forma `where` cuando se desea definir una *asociación de valores* local a una función[4].

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

Ejemplos de uso:

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

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. Las expresiones `let` tienen un fin similar a las `where` (permiten asociar valores a funciones en una expresión). Sin embargo, a diferencia de las formas `where`, las `let` son realmente expresiones evaluables[5]. Su sintaxis general es la siguiente:

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

[4]Esta *asociación de valores* tiene como ámbito la ecuación de la definición de una función donde aparezca.

[5]El ámbito de esta asociación es exclusivamente el de la expresión donde va incluida.

`in` *exp*

Ejemplos de uso:

```
f1 = let a = 3 + 2
    in a * a * a

f2 = 4 * (let a = 9 in a + 1) + 2
```

Adicionalmente, las expresiones `let` se pueden utilizar en GHCi. Mediante las mismas, se puede asociar valores a funciones que se pueden definir en una sesión. Por ejemplo:

```
Prelude> let x = (2 +)
Prelude> let y = x 4 + 3
Prelude> y
9
```

5. Currificación y aplicación parcial

En Haskell existen dos alternativas para definir funciones de dos o más argumentos. Por ejemplo, las siguientes son dos definiciones de una operación de adición de enteros, `add` y `cAdd`:

```
add :: (Int, Int) -> Int
add (x,y) = x + y

cAdd :: Int -> Int -> Int
cAdd x y = x + y
```

Existe un isomorfismo entre los dominios de estas dos versiones de la función[6]. La segunda versión se dice que está *currificada*, teniendo algunas ventajas respecto a la primera desde el punto de vista de la programación:

- es una forma de reducir el número de paréntesis de una expresión ya que para invocar a la función, se escribe `cAdd x y`, y no `add(x,y)`.
- facilita la aplicación parcial de una función, que, intuitivamente, consiste en no proporcionar todos los argumentos de una función.

En general, la aplicación parcial de una función supone el siguiente modo de funcionamiento. Sea una función `f` con dos argumentos de entrada de tipo `a` y `b` respectivamente y salida de tipo `c`:

```
f :: a -> b -> c
```

Supóngase cierta expresión `exp` de tipo `a` (que también puede escribirse como `exp::a`) y también puede verse como una función constante de tipo `a`.

[6] Un isomorfismo (u homomorfismo biyectivo) es una relación entre estructuras algebraicas que preserva la estructura. Los cómputos realizados en cualquiera de los dos dominios serán equivalentes.

Se puede escribir la expresión "`f exp`" que será de tipo

```
f exp :: b -> c
```

Es decir, "`f exp`" será una nueva función que tomará como argumento un elemento de tipo `b` y dará como resultado un elemento de tipo `c`.

Véase ahora el siguiente ejemplo con operadores aritméticos. Asumiendo el siguiente operador aritmético para la multiplicación (nótese que un operador definido entre paréntesis indica que dicho operador se puede utilizar en notación infija):

```
(*) :: Int -> Int -> Int
```

Ahora es posible definir varias funciones aritméticas, haciendo uso del orden superior.

```
squarepow :: Int -> Int
squarepow x = x * x

doubleHO :: (Int -> Int) -> Int -> Int
doubleHO f x = f (f x)

fourthpow :: Int -> Int
fourthpow = doubleHO squarepow
```

La función `doubleHO` es de *orden superior* ya que uno de sus argumentos es una función en vez de un dato, en concreto, el parámetro `f`. Es importante remarcar que la función `fourthpow` espera un argumento de tipo `Int` aunque su definición no incluya ningún parámetro formal (es decir no aparece definido como `fourthpow x`). Eso se debe al uso del la *aplicación parcial* de la función de orden superior `doubleHO` que recibe como argumento la función `squarepow`.

El operador " `->` " es asociativo por la derecha, es decir, "`a -> b -> c`" equivale a "`a -> (b -> c)`" y difiere de "`(a -> b) -> c`". El operador de aplicación funcional es asociativo por la izquierda, es decir, "`f a b`" equivale a "`(f a) b`" y difiere de "`f (a b)`".