Introducción a la Programación en Haskell

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20

Objetivos

Objetivos

El objetivo de este tema es presentar:

- la estructura básica de un programa
- el entorno de trabajo con Haskell
- la forma de organizar los fuentes de nuestro programa
- los elementos básicos del lenguaje

Introducción

¿ Haskell ?

Haskell Brooks Curry (1900.09.12 – 1982.09.01) fue un matemático y lógico estadounidense.
El trabajo principal de Curry fue en lógica matemática, especialmente en la teoría de sistemas y procesos formales y lógica combinatoria, el fundamento para los lenguajes de programación funcionales.

Si no me equivoco, 1 es igual a 2.
– Paradoja de Curry –

Haskell es un lenguaje que permite trabajar de modo interactivo/interpretado o compilado.

Existen varios compiladores de Haskell. Los más conocidos son:

- HUGS (*Haskell User's Gofer System*)
- **GHC** (*Glasgow Haskell Compiler*)

Para este curso hemos elegido GHC.

Entorno de trabajo

El Glasgow Haskell Compiler (GHC) es un compilador con las siguientes características:

- open–source, disponible para windows, linux, osx, freebsd, ...
- soporta el estándar Haskell 2010
- genera código nativo (compila)
- entorno interactivo REPL (interpreta)

Es parte de la denominada Haskell Platform (compilador + bibliotecas), y es prácticamente el compilador 'oficial' de Haskell.

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

Hello World

Hello World

Hello World en Haskell:

hello.hs

```haskell
-- Hello World

module Main where

main :: IO()
main = putStrLn "Hello World!!"
```

Hello World

Hello World mínimo en Haskell (podemos omitir algunas declaraciones):

hello.hs

```haskell
-- Hello World

main = putStrLn "Hello World!!"
```

Hello World

Para compilar nuestro fichero fuente usamos ghc:

```
~/hs/hello> ghc hello.hs @\keys{\return}@
[1 of 1] Compiling Main             ( hello.hs, hello.o )
Linking hello ...

~/hs/hello> ls @\keys{\return}@
hello  hello.hi  hello.hs  hello.o

~/hs/hello> ./hello @\keys{\return}@
Hello World!!
```

Hello World

GHC utiliza los siguientes ficheros:

`hello.hs`	⇝	código fuente (*haskell source*, ASCII, UTF8)
`hello.o`	⇝	fichero objeto (binario)
`hello.hi`	⇝	fichero de interfaz (*haskell interface*, binario)
`hello`	⇝	programa ejecutable (nativo)

Los ficheros de interfaz se utilizarán para la importación de módulos en compilación separada.

Un programa más completo tendría este aspecto:

fact.hs

```haskell
main = do
  putStrLn "Gimme a number:"
  arg <- getLine
  print (fact (read arg))
  putStrLn "Thanks!"

fact 0 = 1
fact n = n * fact (n-1)
```

Compilación separada

Compilación separada

La estructura básica de un programa Haskell se basa en **módulos**:

Factorial.hs

```haskell
module Factorial where

fact 0 = 1
fact n = n * fact (n-1)
```

Compilación separada

Un módulo:

- se identifica con un nombre: `module Factorial`
  Nota: debe empezar con mayúscula.
- se implementa en un fichero cuyo nombre debe coincidir con el del módulo: `Factorial.hs`
- puede importarse desde otros módulos: `import Factorial`
- pueden estar organizados en una jerarquía:
  `module Util.Factorial`
  `Util/Factorial.hs`

Compilación separada

La estructura básica de un programa Haskell se basa en los **módulos**:

fact-m.hs

```haskell
import Factorial

main = do
  putStrLn "Gimme a number:"
  arg <- getLine
  print (fact (read arg))
  putStrLn "Thanks!"
```

Compilación separada

El compilador detecta lo que es necesario recompilar:

```
~/hs> ls
fact-m.hs  Factorial.hs

~/hs> ghc fact-m.hs
[1 of 2] Compiling Factorial        ( Factorial.hs, Factorial.o )
[2 of 2] Compiling Main             ( fact-m.hs, fact-m.o )
Linking fact-m ...

~/hs> ls
fact-m.hi  fact-m.o   Factorial.hs
fact-m*    fact-m.hs  Factorial.hi  Factorial.o

~/hs> touch fact-m.hs; ghc --make fact-m
[2 of 2] Compiling Main             ( fact-m.hs, fact-m.o )
Linking fact-m ...

~/hs> touch fact-m.hs; ghc fact-m.hs
[2 of 2] Compiling Main             ( fact-m.hs, fact-m.o )
Linking fact-m ...
```

Entorno interactivo

Entorno interactivo

El entorno interactivo de GHC se lanza mediante el comando **`ghci`**.

En ese entorno podemos cargar un programa, compilarlo, depurarlo, ejecutarlo y ejecutar cualquiera de las funciones que se definen en él.

```
~/hs> ghci
GHCi, version 9.2.1: https://www.haskell.org/ghc/  :? for help
ghci> :load hello.hs
[1 of 1] Compiling Main             ( hello.hs, interpreted )
Ok, one module loaded.
ghci> main
Hello World!!
ghci> :quit
Leaving GHCi.
```

Entorno interactivo

Podemos también cargar un módulo independiente:

```
~/hs> ghci
GHCi, version 9.2.1: https://www.haskell.org/ghc/  :? for help
ghci> :load Factorial
[1 of 1] Compiling Factorial        ( Factorial.hs, interpreted )
Ok, one module loaded.
ghci> fact 23
25852016738884976640000
ghci>
```

Caso especial: el módulo `Prelude`.

Contiene la biblioteca estándar de Haskell, se carga automáticamente y siempre está disponible tanto en interactivo como en el compilador.

Sintaxis básica

En Haskell se utiliza el concepto de **definición** para reemplazar al de asignación en otros lenguajes:

la **asociación** de un nombre simbólico con una expresión.

```
ghci> a = 'c'
ghci> a
'c'
ghci> a = "hola"
ghci> a
"hola"
ghci> a = 1
ghci> a
1
```

```
ghci> :set +t
ghci> a = 'c'
a :: Char
ghci> a = "hola"
a :: String
ghci> a = 1
a :: Num p => p
ghci> :unset +t
```

En Haskell son los **valores** los que tienen un tipo, no las variables (ya que no existen...).

```
ghci> :set +t
ghci> a = 1
a :: Num p => p
ghci> a = 'c'
a :: Char
ghci> a = "hola"
a :: String
ghci> a = True
a :: Bool
ghci> :unset +t
```

```
ghci> :type 1
1 :: Num p => p
ghci> :type 'a'
'a' :: Char
ghci> :type "hola"
"hola" :: String
ghci> :type True
True :: Bool
ghci> h = "hola"
ghci> :type h
h :: String
```

Los **tipos básicos** definidos en Haskell son:

`Bool`	`True,False`	`&& || not ...`
`Int`	entero nativo	`+ - * ^ div mod ...`
`Integer`	entero infinito	`+ - * ^ div mod ...`
`Float`	simple	`+ - * / ...`
`Double`	doble	`+ - * / ...`
`Char`	`''`	`[Data.Char] ord chr isUpper ...`
`String`		`length ...`

Todos esos tipos disponen de operadores básicos predefinidos:

- Igualdad: == /=
- Comparación: < > <= >=

Los operadores pueden tratarse como funciones y viceversa:

```
ghci> 2 + 3
5
ghci> (+) 2 3
5
ghci> div 9 2
4
ghci> 9 `div` 2
4
```

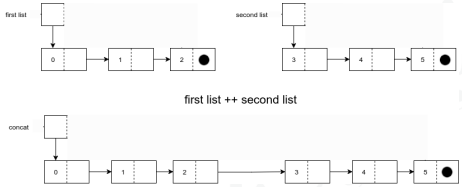
La estructura de datos básica en Haskell es la **lista homogénea**:

```
1  [Int]              : [1,2,3,4]
2  [Char] = String    : ['H','o','l','a'] = "Hola"
3  [(Num->Num->Num)] : [(+),(*)]
```

Las operaciones básicas con listas son

Construcción	:	`1:[2,3]`
Concatenación	:	`[2,3] ++ [4,5]`
Longitud	:	`length`
Acceso	:	`head tail last init take drop (!!)`

Las listas se implementan mediante listas de enlace simple:



Los datos contenidos en los nodos son compartidos.
La implementación y el coste de las operaciones sobre listas se deduce de esa decisión (*p.e.*, head vs `last`).

Otra estructura de datos básica de Haskell son las **tuplas**:

```
1  (Int,Int)           : (1,2)
2  (Char,String)       : ('b',"-...")
3  (String,Float,Int) : ("Juan",1.75,52)
```

Veremos más tarde cómo acceder a sus componentes...

La instrucción elemental en Haskell es la **llamada a función**:

```
ghci> head "Hola"
'H'
ghci> tail "Hola"
"ola"
ghci> head [1,2,3]
1
ghci> tail [1,2,3]
[2,3]
```

Los parámetros se separan mediante espacios.

```
ghci> add x y = x+y
ghci> add 3 4
7
```

Mucho cuidado con esto:

```
ghci> add (3,4)

<interactive>:12:1: error:
```

¿ Por qué ?

Los parámetros se separan mediante espacios.

En caso de duda se pueden usar paréntesis para resolver la ambigüedad:

```
ghci> print add 3 4

<interactive>:13:1: error:
...
ghci> :type print
print :: Show a => a -> IO ()
ghci> print (add 3 4)
7
```

La llamada a función es la operación de mayor prioridad, incluso mayor que los operadores aritméticos.
En caso de duda se pueden usar paréntesis para resolver la ambigüedad:

```
ghci> print 3 + 4
<interactive>:13:1: error:
```

Interpretación basada en las prioridades:

```
ghci> (print 3) + 4   // Error
```

Solución:

```
ghci> print (3 + 4)
```

Las funciones son valores de primera clase, por lo que podemos:

- construir valores de tipo función mediante las **expresiones lambda**

```
(\x -> x+x)
```

- usar valores de tipo función

```
ghci> (\x -> x+x) 4
8
it :: Num a => a
ghci> (\x -> x+x) 1.5
3.0
it :: Fractional a => a
```

Las funciones son valores de primera clase, por lo que podemos:

- definir símbolos asociados a valores de tipo función (*ie*, definir funciones)

```
f = (\x -> x+x)
f :: Num a => a -> a
```

- usar funciones

```
ghci> f 4
8
it :: Num a => a
ghci> f 1.5
3.0
it :: Fractional a => a
```

Las funciones son valores de primera clase, por lo que podemos:

- definir funciones mediante una notación simplificada

```
f x = x+x
f :: Num a => a -> a
```

- usar funciones

```
ghci> f 4
8
it :: Num a => a
ghci> f 1.5
3.0
it :: Fractional a => a
```

Las funciones pueden declararse, pero no es estrictamente necesario. Muchas veces puede ser incluso contraproducente, ya que Haskell utiliza el sistema de **inferencia de tipos** de Hindley–Milner, y las funciones son polimórficas por defecto.

```haskell
add :: Int -> Int -> Int
add x y = x+y

print (add 3 4)      -- OK
7
print (add 3.0 4.0) -- Error
```

```haskell
-- add :: Int -> Int -> Int
add x y = x+y

print (add 3 4)      -- OK
7
print (add 3.0 4.0) -- OK
7.0
```

Aun así, puede haber casos en los que nos interese especificar (declarar) una función, aunque sea polimórfica:

```haskell
-- 'a' y 'b'
-- son tipos desconocidos
primero :: a -> b -> a
primero x y = x

segundo :: a -> b -> b
segundo x y = y
```

```haskell
-- 'a' es un tipo desconocido

get :: Int -> [a] -> a
get pos lst = lst !! pos
```

Permiten definir relaciones entre los tipos de los datos implicados en la función (parámetros y resultado).

Las funciones en Haskell están definidas en **forma de Curry**, es decir, soportan la curryficación o evaluación parcial:

```haskell
get :: Int -> [a] -> a
get pos lst = lst !! pos



get 0 [1,2,3]
1
get 1 [1,2,3]
2
```

```haskell
prm :: [a] -> a
prm = get 0

seg :: [a] -> a
seg = get 1

prm [1,2,3]
1
seg [1,2,3]
2
```

Las funciones en Haskell están definidas en **forma de Curry**, es decir, soportan la curryficación o evaluación parcial.
Eso significa que cualquier función con varios argumentos puede interpretarse según qué símbolo -> elijamos como 'central':

```
1  get    :: Int -> [a] -> a
2
3  get    :: Int -> [a] -> a
4
5
6  filter :: (a -> Bool) -> [a] -> [a]
7
8  filter :: (a -> Bool) -> [a] -> [a]
```

Las variables (y en particular los parámetros de una función) en Haskell son variables en el sentido matemático, son **definiciones**.

Las funciones también son **definiciones**.

La evaluación de expresiones se realiza mediante reglas de sustitución:

```
ghci> z = 3
ghci> doble x = x+x
ghci> doble z
-- por la definicion de doble
z+z
-- por la definicion de z
3+3
-- por la definicion de (+)
6
```

Las funciones también son **definiciones**.

La evaluación de expresiones se realiza mediante reglas de sustitución.

Eso permite la implementación de la evaluación perezosa:

```
ghci> primero x y = x
ghci> a = 1
ghci> b = error "nooooooo"
ghci> primero a b
-- definicion de primero
a
-- defincion de a
1
```

```
ghci> primero x y = x
ghci> a = 1
ghci> b = error "nooooooo"
ghci> primero b a
-- definicion de primero
b
-- defincion de b
error "nooooooo"
*** Exception: nooooooo
CallStack (from HasCallStack):
  error, called at ....
```

¿ Qué salida obtenemos en este caso ?

```
1  ghci> inc x = x+1
2  ghci> z = 2
3  ghci> z
4  2
5  ghci> z = inc z
6  ghci> z
7  ????
```

```
ghci> z
-- por la definicion de z
inc z
-- por la definicion de inc
z + 1
-- por la definicion de z
inc z + 1
-- por la definicion de inc
z + 1 + 1
-- por la definicion de z
inc z + 1 + 1
-- por la definicion de inc
z + 1 + 1 + 1
-- por la definicion de z
inc z + 1 + 1 + 1
....
-- hasta el infinito y más allá
```

En realidad, muchas de las operaciones que Haskell ofrece como predefinidas se implementan en el **`Prelude`**, escritas a su vez en el propio lenguaje, y mediante operaciones todavía más básicas.

Ejercicio para dentro de un par de clases:

Usando solo el operador de construcción '`:`' y el **ajuste de patrones**:
¿ Cómo implementarías operaciones como **`head`, `tail`, `last`**, etc... ?

Problema

Ejemplo:

head

```
head :: [a] -> a

head (x:_) = x
```

Y el resto... ?

Introducción a la Programación en Haskell

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20