Lógica Computacional, 2019-2
Nota de laboratorio 1: Introducción a Haskell

Manuel Soto Romero

6 de febrero de 2019

Facultad de Ciencias, UNAM

1. Conociendo Haskell

Haskell es el lenguaje de programación que se usa durante la primera parte de estas notas de laboratorio, a continuación se listan algunas de sus principales características:

- Es un lenguaje de programación funcional, esto quiere decir que las funciones pueden ser pasadas como parámetro a otras funciones, son devueltas como resultado e incluso son almacenadas en estructuras de datos, es decir, son tratadas como cualquier valor en el lenguaje.
- Es un lenguaje de programación funcional *puro,* esto es, que las funciones cumplen con el principio de *transparencia referencial* que dice que *si una función es llamada con una misma entrada dos o más veces, siempre se obtiene la misma salida*.
- El principio de transparencia referencial, da lugar a que no haya concepto de estado en el lenguaje, esto es, *no se tienen efectos secundarios* externos que alteren la definición de una función o el valor asignado a una variable, pues las operaciones de asignación nunca están presentes.
- No existen primitivas para iterar como lo son `while` o `for` de otros lenguajes de programación, en su lugar, se usa recursión.
- Usa un mecanismo de evaluación perezoso para la reducción de expresiones, esto quiere decir que Haskell no evalúa una expresión, hasta que sea estrictamente necesario.

1.1. Interacción con Haskell

Para interactuar con Haskell, es recomendable hacerlo a través del intérprete de GHC[1] pues su uso es bastante sencillo e intuitivo para aquellos usuarios que se están iniciando en el uso del lenguaje, sin embargo, también es posible compilar los programas escritos en Haskell a través de GHC. Para usar el intérprete, se hace uso de la línea de comandos y un editor de texto.

Para ejecutar Haskell, debe ejecutarse el comando `ghci` desde la línea de comandos, esto cargará automáticamente un programa que actúa como si fuera una calculadora: se escribe una expresión, se presiona la tecla `[Intro]` y se imprime una respuesta, este programa es conocido como REPL[2] o intérprete.

[1] Glasgow Haskell Compiler

[2] Del inglés Read-Eval-Print Loop.

```
$ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/ :? for help
Prelude>
```

Los archivos de definiciones de Haskell tienen la extensión `.hs`, para cargar estos archivos se debe abrir el intérprete y especificar el archivo con las definiciones correspondientes a través del comando `:load` o su versión simplificada `:l`.

```
Prelude> :l archivo.hs
[1 of 1] Compiling Main ( archivo.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Para recargar un archivo de definiciones, en caso de alguna modificación, se usa el comando `:reload` o su versión simplificada `:r`.

```
Prelude> :r
Ok, modules loaded: Main.
```

Para salir del intérprete se usa el comando `:quit` o su versión simplificada `:q`.

```
Prelude> :q
Leaving GHCi.
```

Para definir estos archivos `.hs` puede usarse cualquier editor de textos como Emacs, Vi, Sublime Text, Atom, Gedit, etcétera.

2. Tipos de datos primitivos

Para comenzar, se presentan los tipos de datos primitivos, se usa el intérprete para visualizar cómo es que se evalúan estas expresiones.

2.1. Tipos básicos

Booleanos (`Bool`)

Para representar booleanos, se tienen el tipo de dato `Bool` con las constantes lógicas `True` y `False` para representar verdadero y falso respectivamente.

```
Prelude> True
True
Prelude> False
False
```

Números enteros (`Int`, `Integer`)

El tipo `Int` representa números enteros acotados, en arquitecturas de 32 bits, el valor máximo es `2147483647` y el valor mínimo es `-2147483648`.

```
Prelude> 1729
1729
Prelude> (-12)
-12
```

Observación. Los números enteros negativos, en Haskell, deben ir delimitados por paréntesis.

El tipo `Integer` representa números enteros no acotados, es decir, de longitud variable.

```
Prelude> 23870814395708943570813992 3489
23870814395708943570813992 3489
```

Números reales (`Float`, `Double`)

El tipo `Float` representa números reales de precisión simple, mientras que el tipo `Double` representa números reales de precisión doble.

```
Prelude> 25.132742
25.132742
Prelude> 25.13285723183498
25.13285723183498
```

Caracteres (`Char`)

Los caracteres se delimitan, como en la mayoría de lenguajes de programación, por símbolos de comilla simple.

```
> 'a'
'a'
> 'E'
'E'
```

Cadenas (`String`)

Las cadenas son agrupaciones de caracteres y de delimitan por comillas dobles.

```
> ``Hola mundo''
``Hola mundo''
```

Existen, en Haskell, otros tipos básicos, su representación y usos se discutirán más adelante.

2.2. Funciones predefinidas

Para usar funciones sobre los tipos básicos, debe especificarse el nombre de la función a aplicar e indicar los argumentos de la función separando cada uno por espacios. A continuación se presentan algunos ejemplos.

Funciones lógicas

Se tienen funciones básicas de la lógica como son la negación, conjunción, disyunción y algunas otras de comparación.

```
Prelude> not True
False
Prelude> True && False
False
Prelude> False || True
True
Prelude> 1 == 1
True
Prelude> 1 /= 1
False
```

Como puede apreciarse, la función `not` representa la negación, la función `&&` la conjunción y finalmente, la función `||` la disyunción de expresiones lógicas.

Observación. Es importante destacar que todas las funciones de Haskell, son en un principio prefijas[3], sin embargo existen algunas funciones que pueden usarse infijamente[4] siempre y cuando, éstas reciban dos parámetros. La disyunción y conjunción son ejemplos de estas funciones.

Funciones aritméticas

Al igual que las funciones lógicas, la mayoría de funciones aritméticas funcionan infijamente. Para usar una función prefija que recibe dos parámetros de manera infija se puede delimitar el nombre de la función por el símbolo ‘ como es el caso de la función `div`.

```
Prelude> 1 + 2 + 3
6
Prelude> 2/3 - 1/3
0.3333333333333333
Prelude> 2 * 3 * 6
36
Prelude> 10 / 2 / 2
2.5
Prelude> div 5 2
2
Prelude> 5 ‘div‘ 2
2
Prelude> 2^3
8
Prelude> 2**3
8.0
Prelude> sqrt 81
9
Prelude> mod 10 2
0
Prelude> max 1 2
2
Prelude> min 1 2
1
```

Manejo de cadenas

[3]Que se coloca primero la función y luego los parámetros.
[4]Que la función va entre los parámetros

```
Prelude> length "Manzana"
7
Prelude> "Manzana" !! 3
'z'
Prelude> "Man" ++ "zana" "Manzana"
```

3. Definición de funciones

Adicional a las funciones predefinidas sobre los tipos de datos básicos, es posible definir funciones propias. El diseño de funciones es un proceso que debe seguir una serie de pasos ordenados y bien especificados para garantizar su correcto funcionamiento, a continuación se presenta un método de definición de funciones y se presentan algunos ejemplos junto a su solución.

3.1. Método de definición de funciones

Cuando se define una función que cumple cierto objetivo, se recomienda seguir los siguientes pasos en el orden en que se indica:

- Entender lo que la función tiene que hacer.
- Escribir su contrato, o sea, su tipo (cuántos parámetros, de qué tipo, qué regresa).
- Escribir la descripción de la función a través de los comentarios.
- Finalmente, implementar el cuerpo de la función.

Para especificar el contrato de una función se usa una notación similar a la del dominio y contradominio de las funciones matemáticas. Se usa la siguiente sintaxis:

```
<nombreFuncion> :: <TipoParámetro1> -> ... -> <TipoParámetroN> -> <TipoSalida>
```

Se especifica el nombre de la función, se especifica el tipo de cada parámetro separado por una flecha (->) y se indica el tipo de la salida de la función al final. Por convención, para nombrar funciones, se usa notación de camello[5], es decir, la primera palabra que identifica la función se escribe en minúsculas y el resto inicia con una letra mayúscula.

Para especificar la descripción de la función, se usan comentarios. En Haskell existen dos tipos de comentarios:

De una línea

[5] Del inglés camel case.

```
-- Comentario de una línea
```

De varias líneas

```
{- Este es un comentario
   de varias líneas -}
```

Finalmente, para definir una función en su forma más simple, se usa la siguiente sintaxis:

```
<nombreFuncion> <parametro1> ... <parametroN> = <salida>
```

Se debe especificar un nombre para la función, una secuencia de parámetros de entrada separados por espacios (puede no haber parámetros), un símbolo de igual (=) y un cuerpo.

Para definir funciones en un archivo `.hs` se debe crear un módulo cuyo nombre sea el mismo del archivo. Por ejemplo si el archivo se llama `Ejemplo1.hs`, la primera línea del archivo debe ser:

```
module Ejemplo1 where
```

3.2. Ejemplos de definición de funciones

Ejemplo 1. Definir la función `ultimaCifra` tal que (`ultimaCifra x`) es la última cifra del número `x`. Por ejemplo:

```
ultimaCifra 325 = 5
```

Solución. Función `ultimaCifra`:

```
1 -- Función que obtiene la última cifra de un número entero.
2 ultimaCifra :: Integer -> Integer
3 ultimaCifra x = mod x 10
```

Listado de código 1: Última cifra de un número

□

Ejemplo 2. Definir la función `maxTres` tal que (`maxTres x y z`) es el máximo de `x`, `y` y `z`. Por ejemplo:

```
maxTres 6 2 4 = 6
maxTres 6 7 4 = 7
maxTres 6 7 9 = 9
```

Solución. Función `maxTres`:

```haskell
-- Función que obtiene el máximo de tres números enteros.
maxTres :: Integer -> Integer -> Integer -> Integer
maxTres x y z = max x (max y z)
```

Listado de código 2: Máximo de tres números

□

Ejemplo 3. Definir la función `xor1` que calcule la disyunción excluyente a partir de la tabla de verdad. Usar 4 ecuaciones, una por cada línea de la tabla.

Observación. El operador de disyunción excluyente devuelve `True` siempre que una de las dos fórmulas sea verdadera y la otra sea falsa.

Solución. Función `xor1`:

```haskell
-- Disyunción excluyente a partir de tablas de verdad.
xor1 :: Bool -> Bool -> Bool
xor1 False False = False
xor1 False True  = True
xor1 True False  = True
xor1 True True   = False
```

Listado de código 3: Disyunción excluyente mediante tablas de verdad

□

Ejemplo 4. Definir la función `areaCirculo` tal que `(area-circulo d)` calcula el área de un círculo de diámetro `d`. Por ejemplo:

```haskell
areaCirculo 10 = 78.53
areaCirculo 4  = 12.56
areaCirculo 16 = 201.06
```

Solución. Función `area-circulo`:

```haskell
-- Función que calcula el área de un círculo dado su diámetro.
areaCirculo :: Float -> Float
areaCirculo d = pi * (d / 2) * (d / 2)
```

Listado de código 4: Área de círculo

□

4. Asignaciones locales

En el Ejemplo 4, se define una función que calcula el área de un círculo a partir de su diámetro. La línea 3 del Listado código 4, que implementa esta función, muestra un cálculo repetido: la división del parámetro `d` entre 2. Este tipo de cálculo resulta ser inefiniciente, por lo que Haskell provee dos formas: `let` y `where` que permiten definir variables con alcance local.

4.1. Asignaciones locales con `let`

La primitiva `let` tiene la siguiente sintaxis:

```
let <variable> = <valor> in
   <cuerpo>
```

Esta primitiva permite asignar un nombre a expresiones a través de variables con alcance local. De esta forma, se obtiene una expresión más eficiente, en cuanto a la evaluación se refiere, debido a que el valor asociado a la variable o asignación local, sólo se calcula una vez.

Ejemplo 5. Modificar la función `areaCirculo` para que haga uso de la primitiva `let` y evite cálculos repetitivos.

Solución. Función `areaCirculo2` usando `let`:

```
1 -- Función que calcula el área de un círculo dado su diámetro.
2 areaCirculo2 :: Float -> Float
3 areaCirculo2 d =
4    let r = (d / 2) in
5       pi * r * r
```

Listado de código 5: Área de círculo usando la primitiva let

□

4.2. Asignaciones locales con `where`

A diferencia de `let`, `where` no es una expresión por sí misma pues debe aparecer siempre dentro de la especificación de una función. La sintaxis de una función que usa `where` es la siguiente:

```
<nombreFuncion> <parametro1> ... = <resultado>
   where <variable> = <valor>
```

Ejemplo 6. Modificar la función `areaCirculo` para que haga uso de `where` y evite cálculos repetitivos.

Solución. Función `areaCirculo3` usando `where`:

```
1 -- Función que calcula el área de un círculo dado su diámetro.
2 areaCirculo3 :: Float -> Float
3 areaCirculo3 d = pi * r * r
4    where r = (d / 2)
```

Listado de código 6: Área de círculo usando la primitiva let

□

5. Condicionales

De la misma forma que en matemáticas, en Haskell pueden definirse funciones por partes de acuerdo a ciertas condiciones. Para establecer estas condiciones existen principalmente dos primitivas: `if` y guardias. La primera primitiva es usada por lo general cuando se tiene una única condición mientras que la segunda se usa cuando se tienen dos o más condiciones.

5.1. Condicional `if`

La sintaxis del condicional `if` es la siguiente:

```
if <condición> then <then-expr> else <else-expr>
```

El primer valor `<condición>` debe ser una expresión booleana, el segundo valor `<then-expr>` se evaluará siempre que la condición sea verdadera, y el tercer valor `<else-expr>` se ejecutará cuando no lo sea.

Ejemplo 7. Definir la función `valorAbsoluto` tal que `(valorAbsoluto x)` es el valor absoluto de un número entero. Por ejemplo:

```
valorAbsoluto 1729 = 1729
valorAbsoluto -265 = 265
```

Solución. Función `valorAbsoluto`:

```
1 -- Función que calcula el valor absoluto de un número entero.
2 valorAbsoluto: Integer -> Integer
3 valorAbsoluto x = if x < 0 then x * (-1) else x
```

Listado de código 7: Valor absoluto de un número

La condición se encuentra en la línea 3 `(x < 0)`, si el valor de entrada es menor que cero, entonces se multiplicará el mismo por `-1` y en caso contrario se regresa el valor tal cual.

□

5.2. Guardias

Al igual que `where`, las guardias sólo pueden usarse con funciones. La sintaxis de una función que hace uso de guardias, es la siguiente:

```
<nombreFuncion> <parametro1> ...
    | <condición1> = <expresión1>
    ...
    | otherwise = <expresión2>
```

Se tienen una serie de expresiones de la forma `<condición> = <expresion>` representando los posibles casos y el valor a devolver en caso de que se cumpla la condición. Opcionalmente se tiene un caso `otherwise` que se evalúa siempre que ninguna condición anterior haya sido verdadera.

Ejemplo 8. Definir la función `nombreMes` tal que `(nombreMes n)` es el nombre del mes representado por el número entero `n`. Por ejemplo:

```
nombreMes 8  = "Agosto"
nombreMes 10 = "Octubre"
nombreMes 11 = "Noviembre"
```

Solución. Función `nombreMes`:

```
1  -- Función que obtiene el nombre del mes representado por el número
2  -- recibido como parámetro.
3  nombreMes :: Integer -> String
4  nombreMes n
5      | n == 1  = "Enero"
6      | n == 2  = "Febrero"
7      | n == 3  = "Marzo"
8      | n == 4  = "Abril"
9      | n == 5  = "Mayo"
10     | n == 6  = "Junio"
11     | n == 7  = "Julio"
12     | n == 8  = "Agosto"
13     | n == 9  = "Septiembre"
14     | n == 10 = "Octubre"
15     | n == 11 = "Noviembre"
16     | n == 12 = "Diciembre"
17     | otherwise = error "Mes inválido"
```

Listado de código 8: Función que obtiene el nombre de un mes dado el número que lo representa

Las líneas 5 a 16 representan todas las posibles condiciones, mientras que la línea 17 representa el caso `otherwise` en caso de que ninguna de las condiciones se cumpla. Si ninguna de las condiciones se cumple, se lanza un error indicando que se trata de un mes inválido.

Para lanzar errores, se usa la primitiva `error`, esta primitiva recibe una cadena que describe el error.

□