

Instituto Tecnológico de Mexicali/TecNM

Programación Lógica y Funcional

Noviembre 4

2016

Mexicali B.C., México

Tutorial Haskell

Elaborado por:
Agustín Sandez Gómez
(12490468)

Profesor:
Jorge Antonio Atempa
Camacho

Índice

	Pág.
1 - Introducción	2
2 - Operadores	5
3 - Funciones básicas succ, min, xax	8
4 - Crear funciones	9
5 - Estructura IF	11
6 - Listas	12
8 - Funciones de Listas 1 de 2	15
9 - Funciones de Listas 2 de 2	17
10 - Rangos	20
11 - Funciones de Listas Infinitas	21
12 - Listas Intencionales	22
13 - Listas Intencionales Dobles	23
14 - Tuplas vs. Listas	24
15 - Funciones Duplas	26
16 - Lista de Duplas ZIP	26
17 - Comando :t (Tipos de Datos)	27
18 - Conversores Show y Read	28
Conclusión	29
Ficha Bibliográfica	29

Tutorial Haskell

1-Introducción

¿Qué es Haskell?

Haskell es un lenguaje de programación estandarizado multi-propósito puramente funcional con semánticas no estrictas y de fuerte tipificación estática. Su nombre se debe al lógico estadounidense Haskell Curry.

Haskell es un lenguaje de programación funcional, hoy en día se conoce muy poco de él, por lo que aún no es muy popular, pero se espera que con el transcurso de algunos años tenga un sitio importante dentro de la programación.

Conforme se avance en este tutorial se irá viendo su sintaxis

Lo primero será descargar la plataforma de desarrollo desde la página oficial, entrando a la sección de descargas se deberá ir hasta donde aparece **Haskell Platform**, aparecerán 3 opciones para los sistemas operativos más populares en este caso se trabajara en la plataforma de Windows 10.

El enlace para esta página: <https://www.haskell.org/downloads>

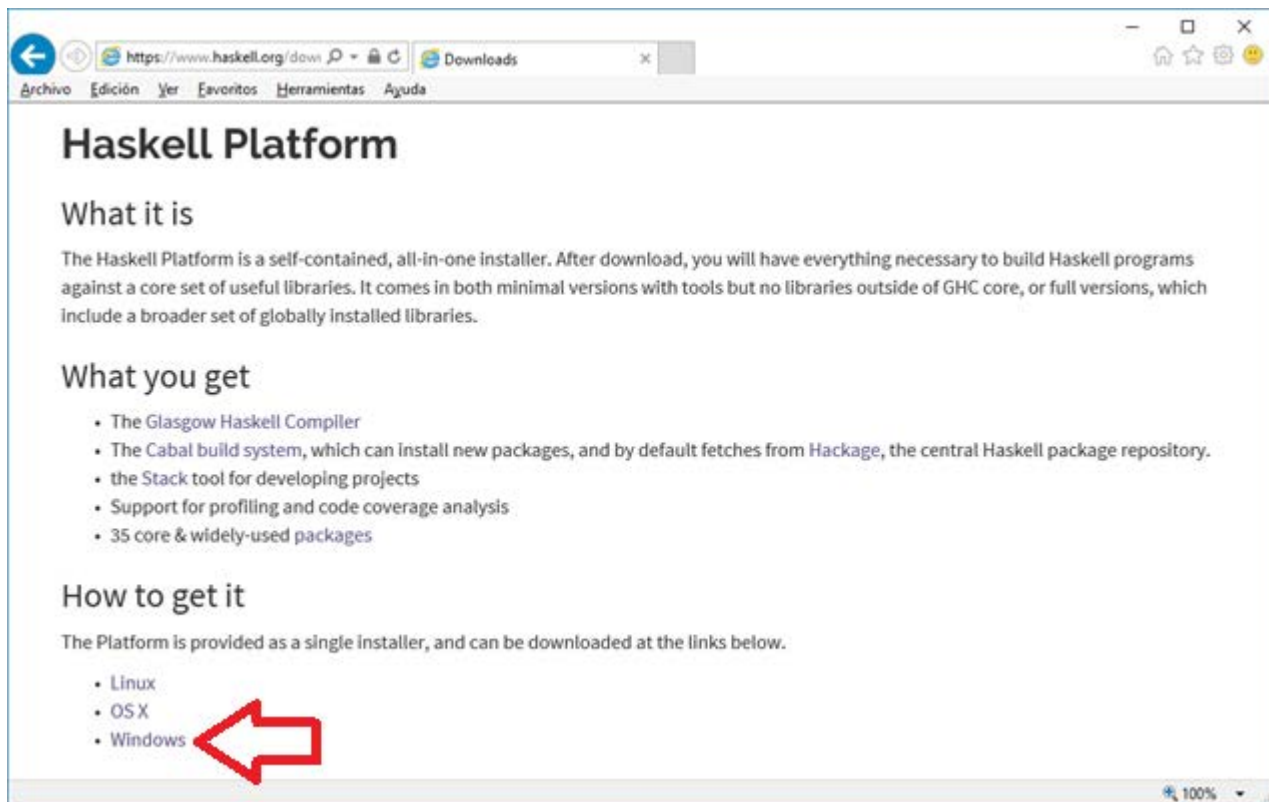


Fig. 1 Página descargas Haskell

Tutorial Haskell

Una vez seleccionada la opción de Windows aparecerá las diferentes versiones de descarga ya sea de 32 o 64 bits además de contemplar una versión reducida o completa.

El enlace para esta página: <https://www.haskell.org/platform/windows.html>

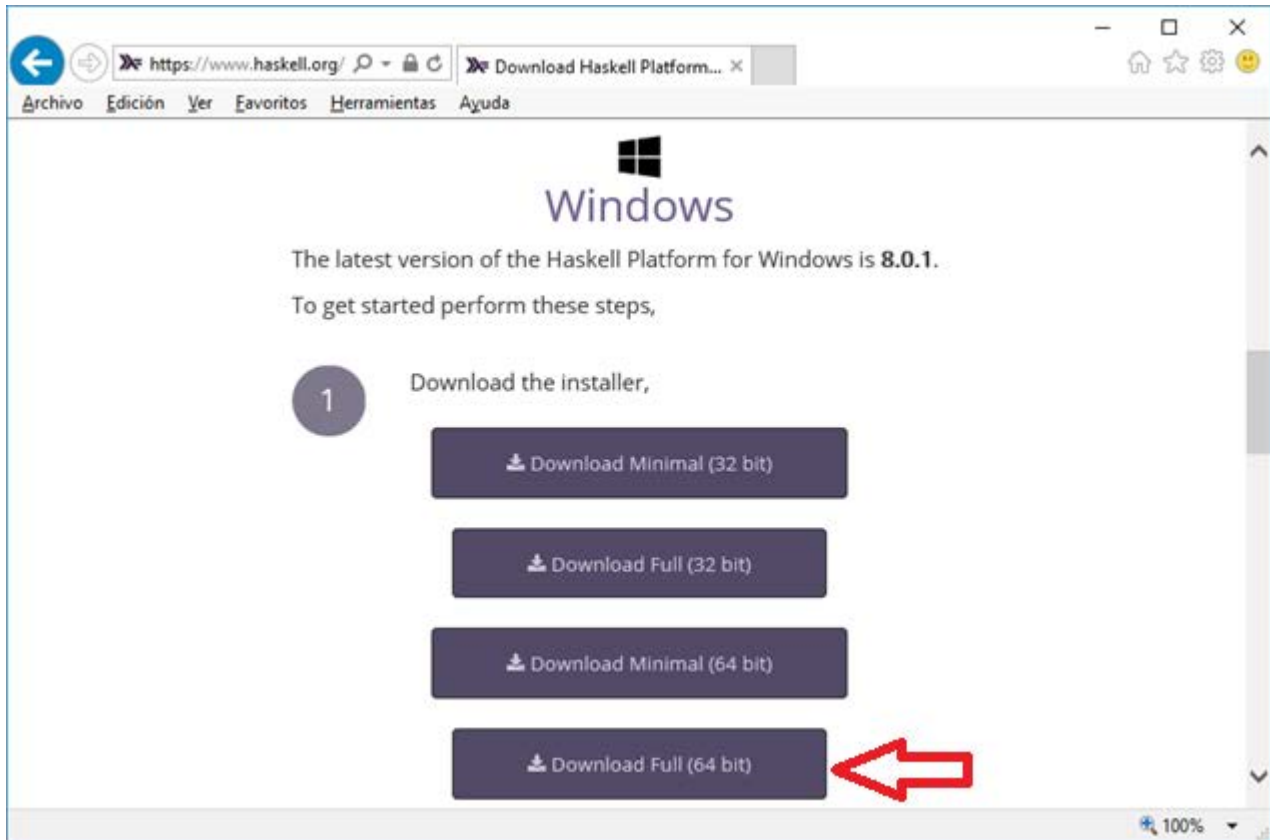



Fig. 2 Diferentes Versiones Haskell para Windows

Para este tutorial se utilizara el siguiente equipo de cómputo y software:

- **Laptop:** Workstation Dell precisión M6400
- **Procesador:** Intel Core 2 Extreme @ 3600mhz
- **RAM:** 8000 Mb DDR3 @ 1600 MHz
- **SSD:** 128 Gb @ 500 MB/s lectura
- **Video:** Nvidia Quadro FX 3700m
- **S.O:** Windows 10 Pro 64 bits Versión 1607
- **Haskell:** Versión completa 8.0.1 @ 64 bits
- **Aplicación Adicional:** Sublime (se utilizara para la edición de archivos de Haskell, se recomienda descargarlo de su página oficial)

Enlace para descarga de **Sublime**: <https://www.sublimetext.com/>

Tutorial Haskell

Una vez que se haya realizado satisfactoriamente la instalación de Haskell se podrá acceder a él a través de Banderita Windows  → Todos los programas → Haskell Platform 8.0.1 → WinGHCi, como se ve en la Fig.3

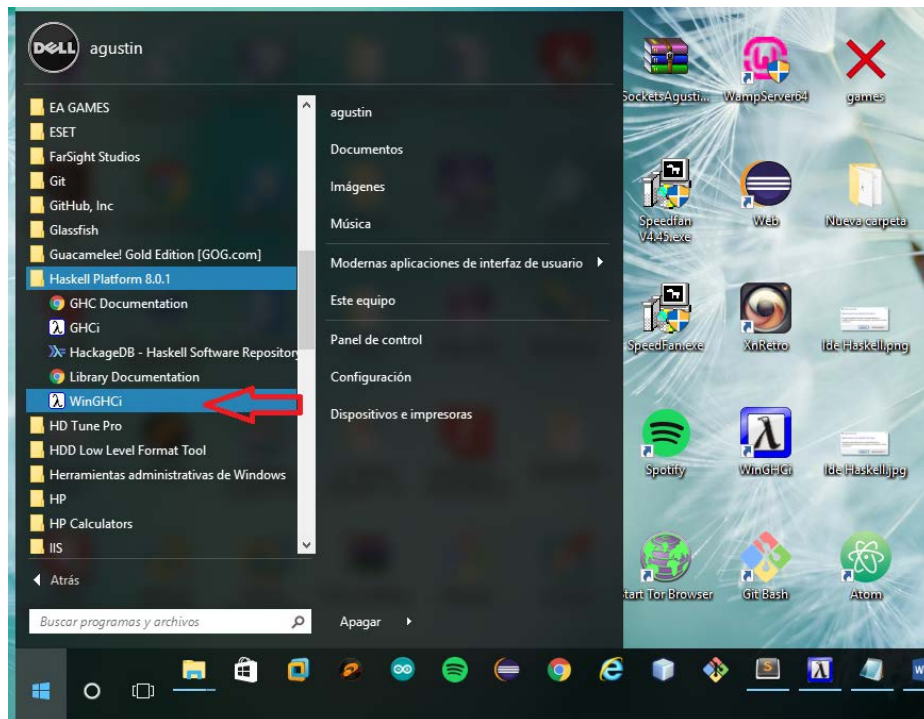


Fig. 3 Ubicación de icono Haskell

Una vez que se abra la aplicación Haskell se vera de la siguiente forma, Fig.4

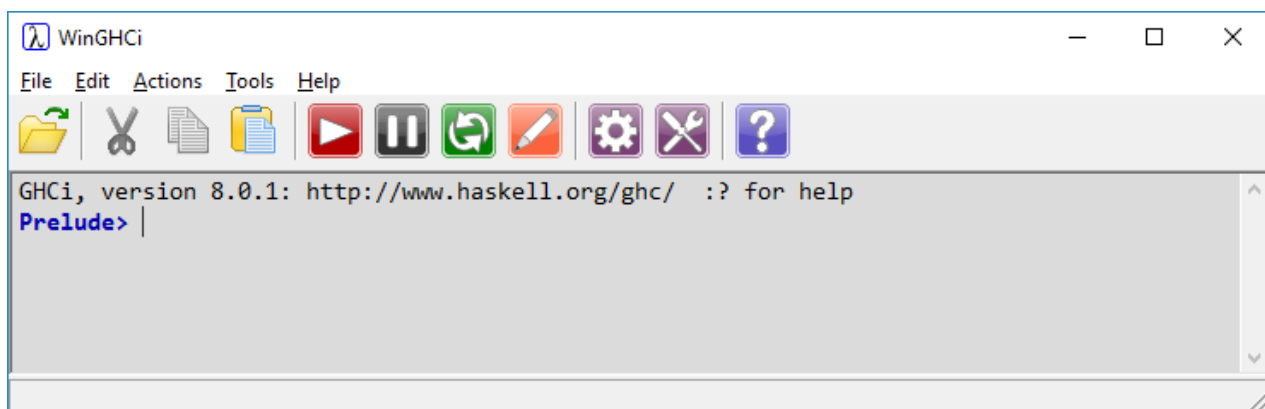


Fig. 4 Haskell Versión para Windows 64 bits

Una vez realizado lo anterior y si no se presentó ningún problema se podrá dar inicio con las diferentes instrucciones que conforman la sintaxis de la programación funcional de Haskell.

2-Operadores

Una vez abierto el intérprete WinGHCi el símbolo `>` significa que está listo para evaluar una expresión.

Operadores matemáticos básicos

Símbolo	Descripción
+	suma
-	resta
/	división
*	multiplicación

Tabla 1 Operadores matemáticas básicos

Ejemplo:

```
> (5+3) * 2
```

```
16
```

```
> 6 / (3 * 2)
```

```
1.0
```

```
> 2 * 2
```

```
4
```

```
> 6 - 7
```

```
-1
```

Hay funciones adicionales en la división que ayudan a obtener el cociente y su residuo.

Ejemplo:

```
> 5 `div` 2
```

```
2
```

```
> 5 `mod` 2
```

```
1
```

Operadores lógicos de comparación

En la siguiente tabla se muestran los operadores booleanos, estos darán como resultado únicamente true o false (verdadero o falso):

Operador	Descripción
True	El valor booleano será verdadero
False	El valor booleano será falso
&&	Operador and
	Operador or
==	Comparación de igualdad, dará como resultado true o false
/=	Comparación de desigualdad, dará como resultado True y False
>	Mayor que
<	Menor que
>=	Mayor e igual que
<=	Menor e igual que
not	Negación

Tabla 2 Operadores lógicos

Ejemplo:

```
> True && False
```

```
False
```

```
> True || False
```

```
True
```

```
> 7 == (4 + 3)
```

```
True
```

```
> 7 /= 3
```

```
True
```

```
> 7 >= 9
```

```
False
```

```
> not True
```

```
False
```

Concatenación

Haskell es muy estricto en la concatenación y no suma atributos de diferentes tipos por ejemplo una cadena de caracteres con números, esto provocara que Haskell envíe un mensaje de error, la concatenación de caracteres no se hace con el operador `+`, si no con `++`

Forma incorrecta de concatenar

```
> 5 ++ "h"
<interactive>:19:1: error:
  No instance for (Num [Char]) arising from the literal '5'
  In the first argument of '(++)', namely '5'
  In the expression: 5 ++ "h"
  In an equation for 'it': it = 5 ++ "h"

>5 + "h"
<interactive>:23:1: error:
  • No instance for (Num [Char]) arising from a use of '+'
  • In the expression: 5 + "h"
    In an equation for 'it': it = 5 + "h"

>5 ++ 6
<interactive>:21:1: error:
  • Non type-variable argument in the constraint: Num [a]
    (Use FlexibleContexts to permit this)
  • When checking the inferred type
    it :: forall a. Num [a] => [a]
```


Forma correcta de operar y concatenar

```
> 5 + 6  
11  
  
> "54" ++ "h"  
"54h"
```

3-Funciones básicas succ, min, max

succ: Devuelve el numero o carácter consecutivo que le estamos pasando

min: Requiere 2 parámetros numéricos y devuelve el número menor

max: Requiere 2 parámetros numéricos y devuelve el número mayor

Ejemplo:

```
> succ 8  
9  
  
> min 5 2  
2  
  
> max 12 78.3  
78.3  
  
> max 4 (succ 7)  
8  
  
> succ (max 8 6)  
9  
  
> succ (max 8 (min 34.6 90))  
35.6
```

4-Crear funciones

Antes de iniciar será necesario abrir sublime para crear nuestro archivo con código Haskell el cual deberá tener una extensión “.hs” para que Haskell pueda cargarlo, ejemplo “nombreArchivo.hs”

En este tutorial se está utilizando Windows así que se recomienda ampliamente que los archivo realizados con código Haskell sean grabado dentro del directorio donde se encuentra WinGHCi por ejemplo el **PATH** o **CAMINO** seria: “E:\Program Files\Haskell Platform\8.0.1\winghci” para que no se presente ningún problema cuando se ejecute la carga de los archivo con los siguiente comandos:

Comando	Descripción
:l	Carga el código Haskell almacenado en un archivo
:load	Carga el código Haskell almacenado en un archivo

Tabla 3 Comandos para cargar código desde un archivo

Se utiliza **:l** o también **:load** las 2 maneras son correctas, seguido por un espacio y el nombre del archivo con o sin su extensión, esto es cuando se carga por primera vez, cabe mencionar que una vez cargado el archivo el símbolo “>” cambiara por “***Main>**”, esto significa que esta compilado el código y podemos trabajar con el.

Ejemplo:

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

Tutorial Haskell

Si se abrió un archivo con “load” y se sigue trabajando con el pero se realizaron cambios, no será necesario volver a escribir todos los pasos anteriores para esto hay una utilidad para recargar el archivo con el siguiente comando:

- `:r` Se utiliza este comando solo, sin poner nombre del archivo, Haskell ya conoce el nombre

Ejemplo:

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

Dicho lo anterior podremos pasar a la creación de una función, una de las primeras reglas es que las funciones no deberán de empezar con mayúsculas.

Se mandara llamar la función **sumaDiez** que se definió en el archivo **sumaDiez.hs** a la que le introduciremos un valor “X”, al cual le sumara 10.

Pasos:

1. Dar un nombre a la función, utilizaremos como ejemplo “sumaDiez”
2. Daremos a la función la ecuación que realizará la operación “x+10”.
3. Nombre la variable que dará la entrada del valor capturado sera “x”

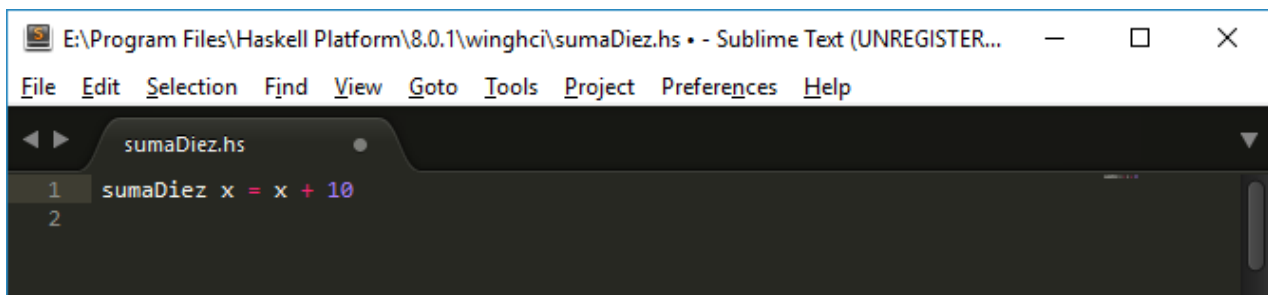


Fig. 4 Utilizando Sublime para la creación de código Haskell

Es importante entender correctamente el código y que no se mal entienda, veamos más a detalle que significa el código escrito:

La función sumaDiez = x + 10 y la “x” restante en el código sería el equivalente al INPUT (entrada) del valor capturado cuando se corra la función.

Tutorial Haskell

Ejemplo:

```
> :l sumaDiez  
[1 of 1] Compiling Main      (sumaDiez.hs, interpreted )  
Ok, modules loaded: Main.  
*Main>sumaDiez 90  
100
```

5-Estructura IF

if es un condicional muy parecido al que se maneja en otros lenguajes de programación pero con la diferencia que en HASKELL es obligatorio utilizar el **then** y **else**, en otras palabras siempre que se utilice la condición **if** será obligatorio poner qué ocurre si es correcto (**then**) y que ocurre si es incorrecto (**else**), como se muestra en la Fig.5

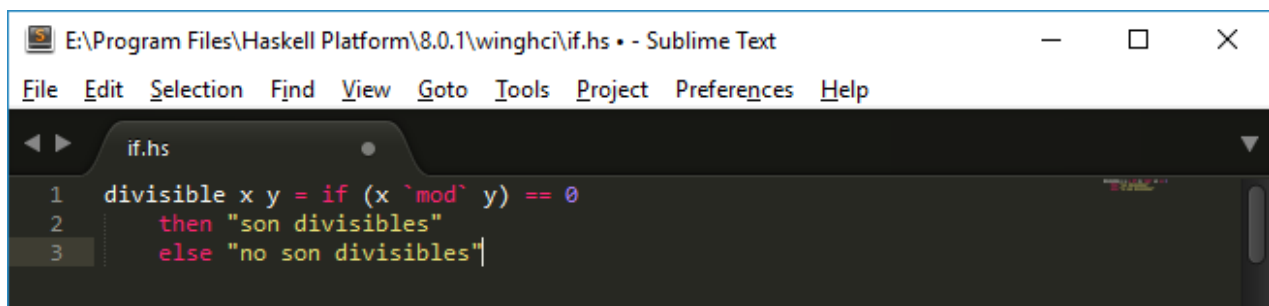


Fig. 5 Código utilizando la estructura if

Ejemplo:

```
Prelude> :l if  
[1 of 1] Compiling Main      ( if.hs, interpreted )  
Ok, modules loaded: Main.  
*Main> divisible 5 6  
"no son divisibles"  
*Main> divisible 6 6  
"son divisibles"
```

6-Listas

Para trabajar con listas la condición es que deberán contener datos del mismo tipo, por lo tanto no se puede generar una lista que contenga números y caracteres, Haskell interpreta de diferente forma una cadena string a una cadena de números, por ejemplo para Haskell un string “hola” es lo mismo que una lista ['h','o','l','a']

Ejemplo:

```
Prelude> ['h','o','l','a']  
"hola"  
Prelude> *Main>
```

Otra condición es que a una lista solo se le podrá introducir otra lista, en el caso de concatenar con ++ e introducir un número que no corresponda a una lista esto producirá un error

Ejemplo:

```
Prelude> [5,1,2] ++ [8,9]  
[5,1,2,8,9]  
Prelude> [5,1,2] ++ 8  
<interactive>:12:1: error:  
• Non type-variable argument in the constraint: Num [a]  
  (Use FlexibleContexts to permit this)  
• When checking the inferred type  
  it :: forall a. (Num [a], Num a) => [a]
```

Tutorial Haskell

Si se desea introducir un solo número se tendrá que manejar como una lista de un solo elemento

Ejemplo:

```
Prelude> [5,1,2] ++ [8]  
[5,1,2,8]
```

En el caso de caracteres será el mismo procedimiento

Ejemplo:

```
Prelude> ['h','o'] ++ ['l','a']  
"hola"
```

Para resolver la concatenación de un número o cadena de caracteres que no pertenezca a una lista se puede resolver utilizando 2 puntos :

Ejemplo:

```
Prelude> 59 : [1,2]  
[59,1,2]  
Prelude> 'H' : "ola mundo"  
"Hola mundo"
```

Anteriormente se mencionó que Haskell a una cadena la considera como una lista de caracteres por ese motivo se escribió 'H' para introducir un carácter a una lista de caracteres, si se escribiera "H" se generaría un error.

7-Índice en Listas

En este tema haremos referencia a diferentes elementos de una lista con respecto a la posición que ocupan.

Dentro de una lista [1..n] debemos entender que 1, inicia con la posición 0 dentro del arreglo y así consecutivamente, para poder ver el elemento en la posición 0 deberemos utilizar el operador doble **!!** Seguido de la posición que se desee, se escribiría de la siguiente manera **!!0**. Para el siguiente ejemplo utilizaremos la palabra reservada **let** para crear una lista.

Ejemplo:

```
Prelude> let lista = [6,7,8]
Prelude> lista
[6,7,8]
Prelude> lista !!0
6
Prelude> lista !!2
8
Prelude> lista !!3
*** Exception: Prelude.!!: index too large
```

Esta lista solo tiene elementos hasta la posición 2 tomando en cuenta que inicia en 0, por eso al poner lista !!3, marca error porque no existe ningún elemento en esa posición.

Haskell puede manipular lista de listas

Ejemplo:

```
Prelude> let lista = [[1,2],[3,4]]
Prelude> lista
[[1,2],[3,4]]
Prelude> lista !!0
[1,2]
Prelude> lista !!1
[3,4]
```

Tutorial Haskell

Para acceder al contenido de una lista contenida en otra lista se hará de la siguiente forma

Ejemplo:

```
Prelude> lista
[[1,2],[3,4]]
Prelude> lista !!0 !!1
2
Prelude> lista !!1 !!1
4
```

8-Funciones de Listas 1 de 2

Si se desea conocer la cantidad de elementos contenidos en una lista se puede utilizar la función **length** el cual regresa como valor el número de elementos que contiene una lista, o la lista de lista.

Ejemplo:

```
Prelude> let lista = [1,2,3,4,5,6,7,8]
Prelude> length lista
8
Prelude> let lista = [['h','o'], ['l','a']]
Prelude> length lista
2
```

La función **head** nos ayuda a conocer el primer elemento de una lista

Ejemplo:

```
Prelude> lista
[1,2,3,4,5,6,7,8]
Prelude> head lista
1
```


Tutorial Haskell

Para conocer el cuerpo de la lista o sea el resto de la lista se utiliza la función **tail**.

Ejemplo:

```
Prelude> lista
[1,2,3,4,5,6,7,8]
Prelude> tail lista
[2,3,4,5,6,7,8]
```

Para ver el último elemento de la lista es con la función **last**.

Ejemplo:

```
Prelude> lista
[1,2,3,4,5,6,7,8]
Prelude> last lista
8
```

Y si se desea ver toda la lista pero sin el último elemento se utiliza la función **init**.

Ejemplo:

```
Prelude> lista
[1,2,3,4,5,6,7,8]
Prelude> init lista
[1,2,3,4,5,6,7]
```

9-Funciones de Listas 2 de 2

Para que se muestre los elementos de una lista de manera inversa se utiliza la función **reverse**. Aclarando que esta función no altera el contenido de la lista.

Ejemplo:

```
Prelude> lista
[1,2,3,4,5,6,7,8]
Prelude> reverse lista
[8,7,6,5,4,3,2,1]
Prelude> let lista = [['h','o'], ['l','a']]
Prelude> reverse lista
["la","ho"]
```

La función **take n** lee los elementos iniciales de una lista, donde **n** significa la cantidad de elemento a leer de una lista, esto significa que solo leerá los elementos indicados ignorando el resto. La función opuesta es **drop** el cual ignora los elementos iniciales definidos y dando como resultado los elementos restantes.

Estas funciones no alteran el contenido de la lista

Ejemplo:

```
Prelude> lista
[1,2,3,4,5,6,7,8]
Prelude> take 5 lista
[1,2,3,4,5]
Prelude> drop 5 lista
[6,7,8]
Prelude> lista
[1,2,3,4,5,6,7,8]
```

Tutorial Haskell

Al inicio de este tutorial se mencionó que la función **min** y **max** solo aceptaba 2 valores como máximo para realizar el comparativo, para resolver este inconveniente cuando se trabaja con una lista de más de 2 elementos, se utiliza la función **minimum** para buscar el valor mínimo y **maximum** el valor máximo.

Otra virtud de estas 2 funciones es que trabajan también con caracteres.

Ejemplo:

```
Prelude> lista
[9,8,3,5,47,6]
Prelude> minimum lista
3
Prelude> maximum lista
47
Prelude> lista
"hola"
Prelude> maximum lista
'o'
Prelude> minimum lista
'a'
```

Si se desea realizar cálculos con las listas hay varias operaciones aritméticas, la función **sum** se encarga de la adición (matemática) y **product** de la multiplicación.

Ejemplo:

```
Prelude> lista
[1,2,3,4,5,6,7,8]
Prelude> sum lista
36
Prelude> product lista
40320
```

Tutorial Haskell

Para comprobar si un elemento que cualquiera que fuera se encuentra dentro de una lista se utiliza la función ``elem`` (importante no omitir las comillas francesas), este regresara como resultado un valor booleano `True` o `False`.

Ejemplo:

```
Prelude> lista
[1,2,3,4,5,6,7,8]
Prelude> 9 `elem` lista
False
Prelude> 4 `elem` lista
True
Prelude> let lista = ['h','o'], ['l','a']
Prelude> lista
["ho","la"]
Prelude> "la" `elem` lista
True
Prelude> 'a' `elem` lista!!1
True
Prelude> 'h' `elem` lista!!0
True
Prelude> 'h' `elem` lista!!1
False
```

10-Rangos

Se vio anteriormente algunos ejemplos de cómo generar listas, pero cabe mencionar que la utilización de `[]`, tiene aún más funciones que nos facilitan el trabajo cuando se desea generar listados con alguna secuencia determinada, para esto Haskell tiene la habilidad de realizar 2 tipos de operaciones cuando se desea generar lista, es la de adición y sustracción.

Ejemplo:

a) Adición

```
Prelude> let lista = [2,4..100]
Prelude> lista
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,
44,46,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,
84,86,88,90,92,94,96,98,100]
```

b) Sustracción

```
Prelude> let lista = [100,97..0]
Prelude> lista
[100,97,94,91,88,85,82,79,76,73,70,67,64,61,58,55,52,49,46,
43,40,37,34,31,28,25,22,19,16,13,10,7,4,1]
```

c) Adición y sustracción con caracteres

```
Prelude> let lista = ['a','c'..'z']
Prelude> lista
"acegikmoqsuwy"
Prelude> let lista = ['z','x'..'a']
Prelude> lista
"zxvtrpnljhfdb"
```

11-Funciones de listas infinitas

- **repeat**
- **cycle**
- **replicate**

Estas funciones tienen la propiedad de generar listas infinitas pero cada una de ellas tiene propiedades diferentes por ejemplo **replicate** genera una lista o lista de listas, pero exige se le proporcione un valor inicial de las veces que se repetirá el ciclo que genera, por lo que se puede decir que esta función no es del todo infinita.

Por otro lado la función **repeat** genera listas continuas muy contrario a **cycle** que también genera una lista infinita pero en forma de cadena.

Ejemplo:

```
Prelude> take 10 (repeat "hola mundo ")
["hola mundo ","hola mundo ","hola mundo ","hola mundo 
","hola mundo ","hola mundo ","hola mundo ","hola mundo 
","hola mundo ","hola mundo "]
Prelude> take 54 (cycle "hola mundo ")
"hola mundo hola mundo hola mundo hola mundo hola 
mundo"
Prelude> replicate 2 [1..10]
[[1,2,3,4,5,6,7,8,9,10],[1,2,3,4,5,6,7,8,9,10]]
Prelude> replicate 3 "hola"
["hola","hola","hola"]
Prelude> replicate 2 ["ho","la"]
[["ho","la"],["ho","la"]]
Prelude> replicate 2 ['h','o','l','a']
["hola","hola"]
```

12-Listas Intencionales

Las listas intencionales es una lista como la que se han mencionado anteriormente pero en esta filtramos, que elementos de las listas queremos, según las condiciones definidas.

Estructura de las listas intencionales

`let lista = [Lo que vamos a mostrar | x <- [lista de la que filtramos] , condiciones]`

Donde **x** es como se llamara cada elemento de la lista, seguido de la notación `<-`

Para entender un poco más el cómo funciona las listas intencionales, llevaremos a cabo un ejemplo donde se muestre una lista de números impares y una lista de números pares multiplicados por diez

Ejemplo:

```
Prelude> let lista = [ x | x <- [1..20] , x `mod` 2==1]
Prelude> lista
[1,3,5,7,9,11,13,15,17,19]
Prelude> let lista = [ x*10 | x <- [1..20] , x `mod` 2==0]
Prelude> lista
[20,40,60,80,100,120,140,160,180,200]
```

Veremos otro ejemplo más utilizando la **estructura if** junto con una nueva función **odd** (impar) que se utiliza para confirmar si un número es impar y devuelve como resultado un valor booleano

Ejemplo:

```
Prelude> cifras lista = [ if x < 10 then "una cifra" else "dos
cifras" | x <- lista , odd x ]
Prelude> cifras [1..30]
["una cifra","una cifra","una cifra","una cifra","una cifra","dos
cifras","dos cifras","dos cifras","dos cifras","dos cifras","dos
cifras","dos cifras","dos cifras","dos cifras","dos cifras"]
```

13-Listas Intencionales Dobles

La estructura de las Listas Intencionales Dobles

let lista = [Lo que vamos a mostrar | x <- [1er lista] y <- [2da lista], condiciones]

Ejemplo:

```
Prelude> let lista = [ x+y | x <- [1..20], y <- [1..100], x < 10, y
`mod` 10 == 0 ]
Prelude> lista
[11,21,31,41,51,61,71,81,91,101,12,22,32,42,52,62,72,82,92,
102,13,23,33,43,53,63,73,83,93,103,14,24,34,44,54,64,74,84,
94,104,15,25,35,45,55,65,75,85,95,105,16,26,36,46,56,66,76,
86,96,106,17,27,37,47,57,67,77,87,97,107,18,28,38,48,58,68,
78,88,98,108,19,29,39,49,59,69,79,89,99,109]
Prelude> sum lista
5400
Prelude> length lista
90
```

Ejemplo:

```
Prelude> let mostrarVocales frase = [letras | letras <- frase,
letras `elem` ['a','e','i','o','u']]
Prelude> mostrarVocales "Hola mundo"
"oauo"
Prelude> let mostrarC frase = [letra | letra <- frase, letra == 'a']
Prelude> sumaA cadenac = sum[1 | x <- (mostrarC cadenac)]
Prelude> sumaA "tengo una manzana que esta muy rica"
6
```


14-Tuplas vs. Listas

Aunque se pudiera creer que las tuplas y listas que son parecidas por el hecho de que ambas son un conjunto de datos, mantienen una clara diferencia la cual es que las tuplas permiten mezclar tipo de datos y mientras que las listas no lo permiten.

- En una lista, los elementos todos son de un solo tipo
- En una tupla, el tipo de cada elemento es independiente del tipo de cualquier otro

Otra diferencia que mantienen estas diferencias es que mientras las listas son representadas con corchetes las tuplas se representan con paréntesis.

Representación de tuplas y listas

- En las tuplas se utiliza ()
- En listas se utiliza []

Desventaja:

Las listas tiene la flexibilidad de manejar varias listas contenidas en una lista (listas de listas) de diferentes dimensiones [[a, b] , [c, d, e]], mientras que las tuplas solo permite listas de listas que sean de las misma dimensiones [(a, b) , (c, d)], una forma equivocada seria la siguiente [(a, b) , (c, d, e)], generando automáticamente un error en haskell

Ejemplo:

```
Prelude> let lista = [1,"a"]
<interactive>:43:14: error:
  • No instance for (Num [Char]) arising from the literal ‘1’
  • In the expression: 1
    In the expression: [1, "a"]
    In an equation for ‘lista’: lista = [1, "a"]

Prelude> let tupla = (1,"a")
Prelude> tupla
(1,"a")
```

Ejemplo:

```
Prelude> let superLista = [[1,2],[3,4,5],[7,8,9,0]]
Prelude> superLista
[[1,2],[3,4,5],[7,8,9,0]]
Prelude> let listaDuplas = [(1,2),(3,4,5)]

<interactive>:60:26: error:
  • Couldn't match expected type '(t, t1)'
    with actual type '(Integer, Integer, Integer)'
  • In the expression: (3, 4, 5)
    In the expression: [(1, 2), (3, 4, 5)]
    In an equation for 'listaDuplas': listaDuplas = [(1, 2), (3, 4, 5)]
  • Relevant bindings include
    listaDuplas :: [(t, t1)] (bound at <interactive>:60:5)
Prelude> let listaDuplas = [(1,"a"),(3,"b")]
Prelude> listaDuplas
[(1,"a"),(3,"b")]Prelude>
```

Siempre que se trabaje con tuplas se debe considerar que estas deberán de tener el mismo formato y tamaño. Como dato adicional a las tuplas de 2 elementos se les llama **DUPLAS**, a las tuplas de 3 elementos **TRIPLAS**, a la de 4 elementos **CUADRUPLAS** y así sucesivamente.

15-Funciones Duplas

Para obtener el elemento de una tupla por su índice se utiliza las siguientes funciones exclusivas para tuplas

- `fst` Este se utiliza para leer el valor en la primer posición
- `snd` Con este se toma el valor de la segunda posición

El primer problema que nos enfrentamos es que no hay ningún tipo en Haskell básico que incluya entre sus valores las tuplas de todos los tamaños. Quiere decir, que no podemos escribir una función que acepte tuplas de cualquier tamaño como argumento.

Ejemplo:

```
Prelude> let dupla = (1,"a")
Prelude> fst dupla
1
Prelude> snd dupla
"a"
```

16-Listas de Duplas ZIP

Con `zip` se logra combinar diferentes listas, convirtiéndola en listas de duplas.

Ejemplo:

```
Prelude> let letras = ['a','b','c']
Prelude> let numeros = [1.7,2.8,3.9,5]
Prelude> zip letras numeros
[('a',1.7),('b',2.8),('c',3.9)]
```

En el caso de que las listas sean de diferentes dimensiones, Haskell dará como resultado de la combinación, una lista de tuplas en base a la lista más pequeña. Si se introdujo una lista de 4 elementos y otra de 3, dará como resultado una tripla.

Tutorial Haskell

Tomando en cuenta que **zip** combina listas, daremos otro ejemplo práctico utilizando el ejemplo anterior, enumerando cada una de las listas.

Ejemplo:

```
Prelude> zip [1..] letras
[(1,'a'),(2,'b'),(3,'c')]
Prelude> zip [1..] numeros
[(1,1.7),(2,2.8),(3,3.9),(4,5.0)]
```

17-Comando :t (Tipos de Datos)

Este comando es muy útil porque tiene que ver con el tipo de datos, más que nada nos ayuda para saber de qué tipo es un valor, como también nos ayuda a saber los valores que recibe una función y que da como resultado.

Ejemplo:

```
Prelude> :t "a"
"a" :: [Char]
Prelude> :t "hola"
"hola" :: [Char]
Prelude> :t True
True :: Bool
Prelude> :t 6
6 :: Num t => t
Prelude> :t 6.8
6.8 :: Fractional t => t
Prelude> :t head
head :: [a] -> a
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t snd
snd :: (a, b) -> b
```

18-Convertidores Show y Read

Para convertir cualquier cosa a cadena de texto se utiliza la función **show** y **read** convierte cualquier tipo de dato en un valor específico, de tal manera que al introducir 2 parámetros en read, tomara como parámetro el segundo valor para convertir el primero al mismo tipo que el segundo.

Ejemplo:

```
Prelude> show False
"False"
Prelude> show [1,2,3]
"[1,2,3]"
Prelude> read "5.6" +8.1
13.7
Prelude> read "False" && True
False
Prelude> read "[4,8,2,4]" ++ [9]
[4,8,2,4,9]
```

Conclusión

Este lenguaje tiene sus orígenes en las observaciones de Haskell Curry y sus descendientes intelectuales apareciendo en el año de 1990, en si la idea desarrollada por estas personas es buena y conociendo a fondo su sintaxis se puede lograr buenos resultados con menos escritura.

Haskell es un lenguaje demasiado estricto en todo el sentido de la palabra, llega a ser complejo el comprenderlo, pero conforme se va envolviendo uno más en su sintaxis, conociéndola, se logra comprender que tiene potencial, logrando el mismo resultado con una fracción de código a diferencia de otros lenguajes.

Se puede pensar que siendo tan práctico este lenguaje funcional y teniendo a favor el poco desarrollo de código obteniendo los mismos resultados, ¿que podría estar mal?

Desafortunadamente la manera de programar en este lenguaje se puede considerar que se lleva de una manera pura, al no contar con un IDE, se puede decir que el desarrollo de cualquier proyecto se tiene que realizar a papel y lápiz, a través de un editor de texto con un poco de soporte para la escritura del código o a través de su consola de comandos, lo cual dificulta la detección de errores al momento de escribir código, si se considera realizar un proyecto grande con este lenguaje se deberá de tener muy presente este punto porque muy posiblemente cuando Haskell emita un error de compilación cause algunos dolores de cabeza buscando la causa raíz del problema.

Ficha Bibliográfica

José Javier Villena. (2014). Tutoriales Haskell. Oct 11, 2016, de Jotajotavm Sitio web: https://www.youtube.com/playlist?list=PLralUviMMM3fbHLdBJDmBwcNBZd_1Y_hC