

INSTITUTO TECNOLÓGICO DE MEXICALI



Ingeniería en Sistemas Computacionales

Programación lógica y funcional.

Arturo Ignacio Garcia Garcia.

12490882.

Atempa Camacho Jorge Antonio.

Manual de haskell.

Mexicali, B.C., A 04 de Noviembre, 2016.

Índice:

Operadores matemáticos.	3.
Funciones en haskell.	6.
Cargar archivos .hs en haskell	7.
Estructura IF.	8.
Listas.	8.
Rangos.	13.
Listas infinitas.	14.
Listas intencionales.	15.
Diferencia entre lista de listas y lista de duplas.	15.
Funciones para duplas.	16.
Función zip.	16.
Función :t.	17.
Función show.	17.
Función read.	17.
Conclusión.	18.

Operadores matemáticos:

En haskell se puede hacer uso de los operadores matemáticos como lo son `+`, `-`, `*`, `/` para ejemplo observe la Figura 1.0.

```
Prelude> 5 + 5
10
Prelude> 5 - 5
0
Prelude> 5 / 5
1.0
Prelude> 5 * 5
25
```

Figura 1.0.

Otra forma de dividir es utilizando el ``div`` y para obtener el residuo se utiliza el ``mod``.

Haskell puede utilizar signos de agrupación como lo son `[]` y `()` estos últimos los utilizan para hacer cuantas matemáticas más complejas, observe Figura 1.1.

```
Prelude> 25 - (5 * 5)
0
```

Figura 1.1.

Haskell también puede utilizar operadores lógicos de true o false, haskell tiene la particularidad de solo aceptar estos operadores cuando están en mayúsculas True y False de otra manera mostrara un error como se observa en la Figura 1.2.

```
Prelude> true
<interactive>:13:1 error:
. Variable not in scope: true
. Perhaps you meant data constructor 'True' (imported from Prelude)
Prelude> True
True
```

Figura 1.2.

Además de sumas también podemos hacer comparaciones de verdadero y falso con haskell observe la Figura 1.3.

```
Prelude> 5 > 2  
True
```

Figura 1.3.

Hacemos una comparación si cinco es mayor que dos, haskell responde con un True ya que es verdadero. También nos permite saber si son diferentes o iguales, observe la Figura 1.4.

```
Prelude> 5 == 5  
True
```

Figura 1.4.

En otros lenguajes de programación utilizamos el `!=` para saber si son diferentes el uno del otro pero en haskell se utiliza el `/=`, observe imagen 1.5.

```
Prelude> 5 /= 5  
False
```

Figura 1.5.

Como podemos apreciar 5 no es diferente de sí mismo por eso obtenemos un false como resultado.

Con haskell podemos negar un valor con not véase Figura 1.6.

Para negar el valor de True se pone not True que el resultado sería un false.

```
Prelude> not True  
False
```

Figura 1.6.

Para concatenar valores se utiliza de signos de más entre lo que se desea concatenar, véase Figura 1.7.

```
Prelude> "hola " ++ "mundo"  
"hola mundo"
```

Figura 1.7

Haskell tiene unas listas ya integradas que cuando usamos el comando succ nos devuelve el siguiente valor, siempre mayor, al valor que le damos. Por ejemplo véase Figura 1.8.

```
Prelude> succ 1
2
Prelude> succ 'a'
'b'
```

Figura 1.8.

Como se puede observar la función succ funciona con números y caracteres.

Podemos pedirle a haskell que nos dé el número menor comparando dos números de entrada, esto se hace con el comando min que nos regresara el número menor comparando dos por ejemplo véase Figura 1.9.

```
Prelude> min 10 2
2
Prelude> min 1000 100
100
```

Figura 1.9.

Como se puede observar el comando min puede determinar que 10 es mayor que 2 y que 100 es menor que 1000.

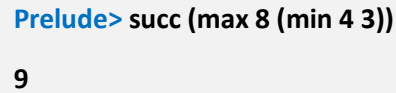
También en haskell podemos pedir el número mayor entre dos números (lo contrario de min), por ejemplo véase Figura 2.0.

```
Prelude> max 10 2
10
Prelude> max 195 5
195
```

Figura 2.0.

Como se puede observar max nos devuelve el valor mayor comparando dos números en este caso 10 es mayor que 2 y 5 es menor que 195.

Podemos utilizar estos comandos juntos al mismo tiempo por ejemplo en otros lenguajes pasamos los parámetros entre paréntesis pero en haskell lo que pasamos entre paréntesis es todo el comando en si para ser procesado y resuelto para dar su resultado y así seguir con los demás comandos observe la Figura 2.1.



```
Prelude> succ (max 8 (min 4 3))  
9
```

Figura 2.1.

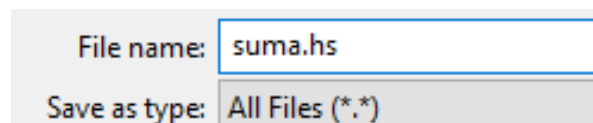
Mínimo de 4 y 3 es 3 el máximo de 8 y 3 es 8 y el siguiente de 8 es 9 así es como funciona.

Funciones en haskell:

En haskell podemos utilizar archivos fuera del programa de compilación donde tenemos escrito nuestro programa así de esta manera desde el IDE podemos llamar a este archivo y hacer uso de la función que tenga escrito el mismo programa para hacer lo siguiente vea las imágenes.

Utilizaremos el programa llamado WinGHCI donde podemos programar en haskell en Windows desde un entorno gráfico.

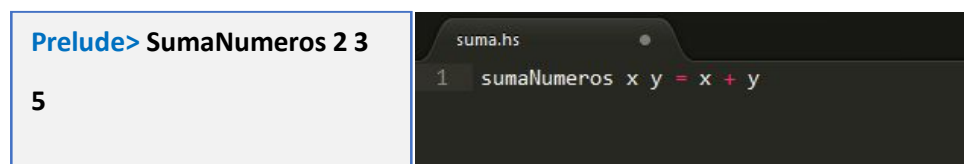
Para llamar a un archivo desde WinGHCI primero debemos de tener el archivo ya creado desde un editor de texto (editor de su preferencia.) el archivo debe de tener la terminación .hs Figura 2.2.



File name: suma.hs
Save as type: All Files (*.*)

Figura 2.2.

En nuestro archivo .hs llamado suma crearemos una función que nos permita sumar dos números así solo tendremos que darle los parámetros para obtener el resultado de la suma y no escribir toda la función en el WinGHCI véase Figura 2.3.



```
Prelude> SumaNumeros 2 3  
5
```

```
suma.hs  
1 sumaNumeros x y = x + y
```

Figura 2.3.

En haskell las funciones son las que se igualan al resultado no lo parámetros por ejemplo la función `sumaNumeros` da la impresión de que `y` es igual a `x + y` pero no es así ya que en haskell la función es igual a `x + y` en este caso `sumaNumero = x+y`.

Cargar archivos .hs en haskell:

Ahora para poder probar si la función funciona tenemos que cargar el archivo al entorno gráfico y así poder compilarlo, desde el WinGHCi correremos el comando `:l` (nombre de nuestro .hs) Figura 2.4.

```
Prelude> :l suma.hs  
  
[1 of 1] compiling Main (suma.hs, interpreted)  
  
Ok, modules loaded: Main.  
  
Main>
```

Figura 2.4.

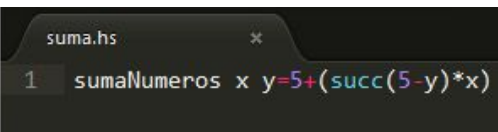
Ya que entramos a nuestro main pondremos el nombre de nuestra función que se encuentra dentro de nuestro .hs más los 2 valores de nuestros parámetros `x` y `y`, en este caso será 3 y 2 y el resultado es 5 Figura 2.5.

```
Main> sumaNumeros 3 2  
  
5
```

Figura 2.5.

Además de sumar con las funciones se pueden hacer muchas cosas más por ejemplo podemos hacer todas las funciones matemáticas además también podemos utilizar las operaciones `succ`, `min`, `max`, véase Figura 2.6.

```
Main> sumaNumeros 5 4  
  
15
```



```
suma.hs  
1 sumaNumeros x y = 5 + (succ (5 - y) * x)
```

Figura 2.6.

Estructura IF:

En haskell cuando usamos if es obligatorio que todo if tenga su then y else, si la condición del if se cumple se realizara la acción que este dentro del then, si en caso contrario se harán las acciones del else. Por ejemplo si queremos saber si un número es mayor que otro número podemos usar un if para averiguarlo Figura 2.7.

```
Main> numeroMayor 2 3
"es menor"
```

```
suma.hs
1  numeroMayor x y = if x > y
2                      then "es mayor"
3                      else "es menor"
```

Figura 2.7.

También podemos sumar o restar en los then y else por ejemplo si x es mayor que y se restaran los números si es menor se sumaran, véase Figura 2.8.

```
Main> numeroMayor 2 3
5
Main> numeroMayor 3 2
1
```

```
suma.hs
1  numeroMayor x y = if x > y
2                      then x - y
3                      else x + y
```

Figura 2.8.

Listas:

En haskell podemos trabajar con listas, para trabajar con listas lo tenemos que hacer con un solo tipo de variable, el cuerpo de listas son compuestas por corchetes por ejemplo una lista de números seria [1,2,3] y una lista de caracteres seria de la siguiente manera ['a','b','c'], las listas se pueden sumar pero estas tienen que ser del mismo tipo de variable, por ejemplo para sumar la lista [1,2,3] con la lista [4,5] lo haríamos con un ++ Figura 2.9.

```
Prelude> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

Figura 2.9.

En listas se pueden agregar más caracteres por ejemplo si queremos agregar un 5 a la lista [1,2,3,4] se utiliza el : esta forma de concatenar listas también funcionan con listas de caracteres por ejemplo si queremos unir la h con la lista ola se haría de la siguiente manera, Figura 3.0.

```
Prelude> 'h' : "ola"  
"hola"  
Prelude> 5 : [1,2,3,4]  
[5,1,2,3,4]
```

Figura 3.0.

No es necesario que ola este entre corchetes ya que haskell lo interpreta automáticamente como una lista de caracteres.

Para hacer referencia a un valor dentro de una lista se utilizan los índices de lista que se acceden con el parámetro !! por ejemplo si queremos saber el valor que se encuentra en la posición 3 de la lista de números [1,2,3,4,5] sería de la siguiente manera, Figura 3.1.

```
Prelude> let lista = [1,2,3,4,5]  
Prelude> lista !! 3  
4
```

Figura 3.1.

En las listas su funcionamiento en base a las posiciones es igual a los arrays en otros lenguajes de programación, su primer elemento que es 1 no está en la posición 1 si no en la 0 por lo tanto el valor de la posición 3 contando desde 0 sería 4.

En haskell podemos acceder a una lista dentro de otra lista con el mismo parámetro !! de la siguiente manera, primero crearemos una lista de listas, Figura 3.1.

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

Figura 3.1.

En el ejemplo Figura 3.1, estamos accediendo a un valor de la lista y como el valor al que accedemos se trata de otra lista podemos acceder a un valor de esta misma lista, de esta manera el valor es 4.

Las listas en haskell tienen distintas funciones por ejemplo head que nos devuelve el primer elemento de una lista o last que nos devuelve el último elemento de una lista, además de estos existen más funciones que se verán a continuación.

Con la función head como fue mencionado anteriormente devuelve el primer elemento de la lista Figura 3.2.

```
Prelude> let lista = [1,2,3,4,5]
Prelude> head lista
1
```

Figura 3.2.

La función tail nos devuelve el todos los valores de la lista excepto la cabeza (el primer elemento), Figura 3.3.

```
Prelude> tail lista
[2,3,4,5]
```

Figura 3.3.

La función init nos devuelve todos los elementos de la lista excepto el último elemento, Figura 3.4.

```
Prelude> init lista
[1,2,3,4]
```

Figura 3.4.

La función last nos devuelve el último elemento de la lista, Figura 3.5.

```
Prelude> last lista
5
```

Figura 3.5.

También con la función `length` podemos saber que tan larga es nuestra lista por ejemplo ahorita en el ejemplo anterior tenemos una lista de longitud 5, Figura 3.6.

```
Prelude> length lista  
5
```

Figura 3.6.

En listas no es lo mismo longitud que el índice ya que en este caso nuestro índice mayor sería 4, Figura 3.7.

```
Prelude> lista !! 4  
5
```

Figura 3.7.

La función `take` nos devuelve la cantidad de valores que le pedimos comenzando desde la cabeza, Figura 3.8.

Recuerde que nuestra lista cuenta con 5 elementos con `take` le pedimos 3 por ende nos regresará `[1,2,3]`.

```
Prelude> take 3 lista  
[1,2,3]
```

Figura 3.8.

Si con `take` le pedimos más valores de los que nuestra lista cuenta este nos devolverá todos los valores por ejemplo observe Figura 3.9.

```
Prelude> take 1000 lista  
[1,2,3,4,5]
```

Figura 3.9.

Con la función `drop` podemos quitar la cantidad de elementos de una lista empezando por la cabeza, Figura 4.0.

```
Prelude> drop 2 lista  
[3,4,5]
```

Figura 4.0.

Si con drop le pedimos que quite una cantidad mayor a nuestra lista esta función quitara todos los elementos de esta, Figura 4.1.

```
Prelude> drop 10000 lista  
[]
```

Figura 4.1.

Recordamos que con min y max podíamos obtener el número que fuera mayor o menor comparando dos números, en listas existen dos funciones `minimum` y `maximum` que nos devuelven el menor y el mayor número de una lista de esta manera podemos comparar una gran cantidad de números, Figura 4.2.

```
Prelude> maximum lista  
5  
Prelude> minimum lista  
1
```

Figura 4.2.

Podemos sumar y multiplicar todos los elementos dentro de una lista por ejemplo véase Figura 4.3.

`Sum` nos devuelve la suma realizada entre los valores de la lista `[1,2,3,4,5]` que su total es 15.

```
Prelude> sum lista  
15
```

Figura 4.3.

Con `product` nos regresaría la multiplicación entre los números de la lista `[1,2,3,4,5]` que el resultado sería 120, Figura 4.4.

```
Prelude> product lista  
120
```

Figura 4.4.

Con la función `elem` podemos saber si algún dato está dentro de una lista, por ejemplo en nuestra lista que contiene los números del 1 al 5 podemos preguntar por cualquiera de estos números y nos dirá que si esta en esta lista pero si preguntamos por algún otro número nos dirá que no se encuentra en esta lista, esta función regresa datos booleanos, Figura 4.5.

```
Prelude> 4 `elem` lista  
True  
Prelude> 8 `elem` lista  
False
```

Figura 4.5.

Rangos:

Haskell tiene la funcionalidad de permitirnos crear listas en base a rangos dados con parámetros en este caso podremos obtener toda una lista de números sin escribirlos por ejemplo del 1 al 1000, sería muy cansado y poco eficiente escribir numero por numero por eso haskell nos permite usar rangos para facilitar este trabajo, para trabajar con rangos se hace de la siguiente manera observe Figura 4.0.

```
Prelude> let lista = [1 .. 1000]
```

Figura 4.6.

Con el método de rangos podemos obtener una lista con los puros números pares hasta un rango de 1000 para este caso se haría de la siguiente manera Figura 4.1.

```
Prelude> let lista = [2, 4 .. 1000]
```

Figura 4.7.

Este método también funciona con letras de esta manera obtendremos todas las letras del abecedario o parte de este, esto se hace de la siguiente manera, Figura 4.2.

```
Prelude> let lista = ['a'..'z']
```

Figura 4.8.

Entonces de esta manera podemos obtener datos de listas por medio de rangos si queremos el rango de 7 a 50 de una lista que tiene 1000 números solo le pasaríamos el rango que queremos de dicha lista.

Listas infinitas:

En haskell nos permite crear listas infinitas de manera que nos regresa los valores de estas listas en forma repetida, esto es algo complicado de controlar ya que si no controlamos la salida de las listas infinitas el programa se puede ciclar, para crear una lista infinita utilizaremos el comando repeat, Figura 4.3.

```
Prelude> repeat 7
```

Figura 4.9.

Los que repeat hace es crear una lista repitiendo el numero siete dentro de sus elementos.

El comando cycle toma una lista y la cicla infinitamente, Figura 4.4.

```
Prelude> cycle [1,2,3]
```

Figura 5.0.

El comando replicate repite cuantas veces se le diga un numero dado, Figura 4.5.

```
Prelude> replicate 3 [1,2,3]
```

Figura 5.1.

Lista intencionales:

Estructura de las listas intencionales es la misma para todas solo varían los datos de las listas por ejemplo vea la Figura 5.2.

```
Prelude> let lista = [lo que mostraremos | nombre de lista <- [lista que filtramos], condición]
```

Figura 5.2.

Por ejemplo si podemos decir que el nombre de la lista sea x entonces si queremos mostrar el resultado de x solo ponemos x en lo que mostraremos, la condición se pone lo que sea necesario para resolver el problema por ejemplo si queremos los números impares vea la Figura 5.3.

```
Prelude> let lista = [x | x <- [1..20], x 'mod' 2 == 1]
```

Figura 5.3.

El ejemplo 5.3, x mostrara los números impares de la lista de 1 a 20, lo que la lista intencional intenta decir es que de la lista que es 1 a 20 con nombre x si el residuo de la división de x entre 2 es igual a 1 entonces muéstrame x de esta manera nos regresa los números impares en una lista.

Diferencia entre lista de listas y lista de duplas:

La principal diferencia entre estas dos es que las duplas pueden contener diferentes tipos de datos, y las listas tienen que tener el mismo tipo de dato por ejemplo si una lista es [1,2,3] si funcionara porque son los mismos tipos pero si le ponemos [1,2,"tres"] ya no funcionara en dado caso que (1,"dos") si funcionara ya que las duplas permiten diferentes tipos de datos al igual que (1,2) igual funciona.

Ahora en una lista de duplas las cosas son algo diferentes ya que todas las duplas dentro de la lista deben de tener las mismas características como en longitud y tipo de datos por ejemplo, Figura 5.4.

```
Prelude> let listaDuplas = [(1,2),(3,4),(4,"cinco")]
```

Figura 5.4.

En este caso nos marcaría un error ya que las duplas son diferentes ahora bien si las duplas son iguales no habrá ningún problema, Figura 5.5.

```
Prelude> let listaDuplas = [(1,"dos"),(3,"cuatro"),(4,"cinco")]
```

Figura 5.5.

Las duplas son llamadas duplas porque contienen 2 datos, si las duplas contienen más datos serán llamadas diferentes por ejemplo una dupla con 3 elementos es llamada tripla, con 4 elementos es cuádrupla y así sucesivamente.

Funciones para duplas:

Como las listas las duplas tienen sus propias funciones para acceder a sus elementos por ejemplo en las listas lo hacemos con los índices pero esto no funciona en duplas, con duplas utilizaremos los términos fst (primero, segundo) ya que las duplas solo tienen dos elementos, Figura 5.6.

```
Prelude> dupla = ("uno", 2)
```

Figura 5.6.

Para acceder a un elemento de la dupla pondremos fst o snd dependiendo de cuál elemento queremos que nos regrese, para que nos regrese la cadena "uno" pondremos fst ("uno", 2), Figura 5.7.

```
Prelude> fst ("uno", 2)
```

Figura 5.7.

Y para que nos regrese el segundo elemento de la dupla cambiamos fst por snd.

Función zip:

La función zip nos permite combinar 2 listas devolviendo 1 lista de tuplas, la forma en que la combina es el primer elemento de la primera lista con el primer elemento de la segunda lista y así sucesivamente, Figura 5.8.


```
Prelude> let lista1 = [1,2,3,4]
Prelude> let lista2 = [1,2,3,4]
Prelude> zip lista1 lista2
[(1,1),(2,2),(3,3),(4,4)]
```

Figura 5.8.

Función :t:

La función :t nos permite saber el tipo de un dato o formato de una función por ejemplo, Figura 5.9.

```
Prelude> :t "a"
"a" :: [char]
Prelude> :t fst
fst :: (a, b) -> a
```

Figura 5.9.

Función show:

La función show nos muestra cualquier dato que le pasemos como cadena de texto, Figura 6.0.

```
Prelude> show 1
"1"
Prelude> show [1,2,3]
"[1,2,3]"
```

Figura 6.0.

Show solo nos regresara cadenas de texto.

Función read:

Por otro lado la función read funciona de diferente manera al igual que show es un convertidor de datos pero este lo hace de una manera diferente, read toma cambia el tipo de dato al primer elemento en base al tipo de dato del segundo elemento como se muestra en el siguiente ejemplo, Figura 6.1.

```
Prelude> read "5" + 3  
8  
Prelude> read False && True  
False
```

Figura 6.1.

Conclusión:

Haskell es un lenguaje de programación algo diferente a lo que estamos acostumbrados pero es algo bueno investigar acerca de nuevas herramientas de trabajo, haskell en si es un lenguaje difícil de dominar ya que es exigente a la hora de declarar sus variables y métodos, es un lenguaje en el que tenemos que saber exactamente como se escriben las cosas ya que este no cuenta con una ayuda como lo hacen muchos lenguajes de programación que cuentan con un IDE.