Instituto tecnológico de Mexicali

Carrera:

Ingeniería en sistemas computacionales

Materia:

Prog. Lógica y funcional

Tema:

Manual de HASKELL

Nombre del alumno:

Valenzuela Leal Jesus Emilio #12490470

Mexicali, Baja California 05 de Noviembre de 2016

Contenido

INTRODUCCION

El propósito de este manual es proporcionar una introducción agradable al entorno haskell, definiendo algunos términos de la manera más sencilla posible, se exponen algunos de los aspectos básicos del lenguaje con varios ejemplos para que sea mas entendible el lenguaje

HASKELL

Haskell es un lenguaje de programación estandarizado multi-propósito puramente funcional con semánticas no estrictas y fuerte tipificación estática. Su nombre se debe al lógico estadounidense Haskell Curry. En Haskell, "una función es un ciudadano de primera clase" del lenguaje de programación. Como lenguaje de programación funcional, el constructor de controles primario es la función. El lenguaje tiene sus orígenes en las observaciones de Haskell Curry y sus descendientes intelectuales.

INSTALACION

Para la instalación de haskell en Windows primero hay que visitar la siguiente página:

https://www.haskell.org/platform/windows.html (para Windows)

https://www.haskell.org/platform/linux.html (para Linux)

https://www.haskell.org/platform/mac.html (para Mac)

En caso que su sistema operativo sea Windows como la figura 1 lo indica la versión completa de haskell está disponible a partir de Windows 8.0.1 si tiene Windows 7 o anterior descargue la versión mínima.

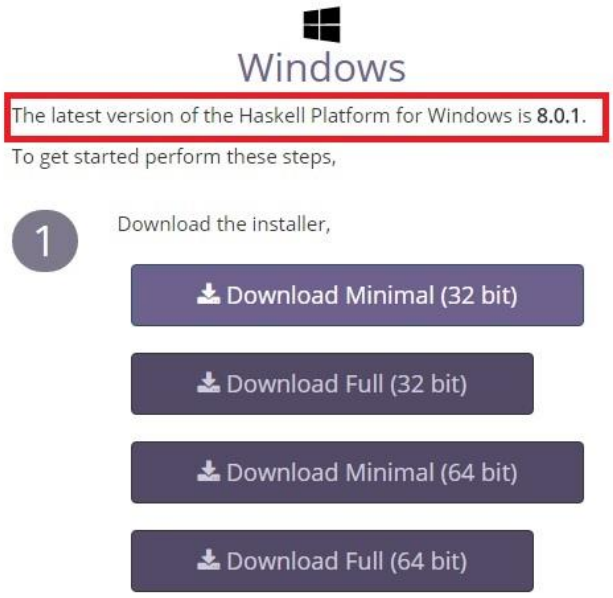

Figura 1

Ya que se haya descargado el instalador ejecutarlo, y solo oprimir siguiente hasta que finalice de instalarlo, una vez terminada la instalación en el menú de Windows aparecerá este icono:

Figura 2

Si la versión que descargaste fue la mínima en el buscador de Windows poner GHCI y aparecerá haskell en su versión de consola:

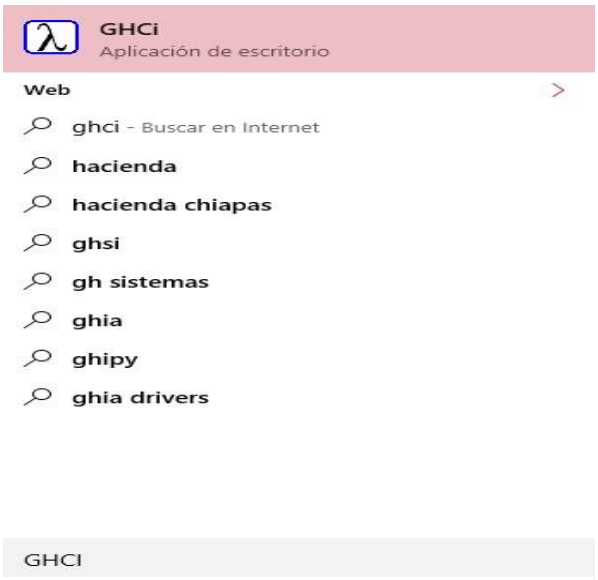

Figura 3

OPERADORES

Haskell soporta cinco operadores matemáticos básicos:

Operador	Símbolo
Suma	+
Resta	-
Multiplicación	*
División	/
Exponenciación	^

A continuación se mostraran unos ejemplos con los operadores en haskell:

```
Prelude> 5+2
7
Prelude> 5-2
3
Prelude> 5*2
10
```

La sentencia `div` se usa para dividir pero devuelve un numero entero y la sentencia `mod` devuelve el residuo.

```
Prelude> 5/2
2.5
Prelude> 5 `div`2
2
Prelude> 5 `mod` 2
1
```

Operadores lógicos

SIMBOLO	SIGNIFICADO
&&	AND
||	OR
==	IGUAL QUE
/=	DIFERENTE QUE
>	MAYOR QUE
<	MENOR QUE
>=	MAYOR O IGUAL
<=	MENOR O IGUAL
Not	NEGACION
TRUE	VERDAD
FALSE	FALSO

Haskell para los operadores lógicos true y false la primera letra debe de ir en mayúsculas ya que si no haskell te mandara error

```
Prelude> true

<interactive>:16:1: error:
    • Variable not in scope: true
    • Perhaps you meant data constructor ‘True’ (imported from
Prelude)
Prelude> TRUE

<interactive>:17:1: error: Data constructor not in scope: TRUE
Prelude> True
True
Prelude>
```

Los operadores lógicos pueden utilizarse para formular preguntas, a continuación unos ejemplos de operadores lógicos:

```
Prelude> True
True
Prelude> 7 == 7
True
Prelude> 7 /= 7
False
Prelude> 7 > 7
False
Prelude> 7 >= 7
True
Prelude> 7 < 7
False
Prelude> 7 <= 7
True
Prelude> 7 == (4+3)
True
```

En haskell la negación no se hace de esta forma !True, haskell mandara error la manera adecuada es not True.

```
Prelude> !True

<interactive>:26:6: error:
    parse error (possibly incorrect indentation or mismatched
brackets)
Prelude> not True
False
Prelude>
```

Se pueden sumar números sin especificar si son enteros o flotantes pero los atributos que tienen distintos tipos no podrá hacerlo por ejemplo:

```
Prelude> 7 + 7.2
14.2
Prelude> 7 + "Hola"

<interactive>:29:1: error:
    • No instance for (Num [Char]) arising from a use of ‘+’
    • In the expression: 7 + "Hola"
      In an equation for ‘it’: it = 7 + "Hola"
Prelude> "hola" + "mundo"

<interactive>:30:1: error:
    • No instance for (Num [Char]) arising from a use of ‘+’
    • In the expression: "hola" + "mundo"
      In an equation for ‘it’: it = "hola" + "mundo"
Prelude> "hola " ++ "mundo"
"hola mundo"
Prelude>
```

Para poder sumar o concatenar 2 cadenas de caracteres siempre se deberán usar las comillas (“ ”) y se utilizara doble signo de mas (++).

FUNCIONES SUCC, MIN Y MAX

SUCC: Esta función manda el valor siguiente de el que se le introduzca, ya sea número o letra.

```
Prelude> succ 7
8
Prelude> succ 'a'
'b'
```

MIN: Esta función devolverá el dato de menor valor de los que se haya introducido, solo acepta dos valores para la comparación y al igual que la función succ funciona tanto con números como caracteres.

```
Prelude> min 4 5
4
Prelude> min 'a' 'd'
'a'
```

MAX: Max es lo contrario que min ya que devuelve el valor mayor.

```
Prelude> max 7 8
8
Prelude> max 'a' 'd'
'd'
```

Para combinar las funciones succ, min y max se utilizan paréntesis para hacer combinaciones más complejas. Los paréntesis sirven para conectar y llamar a las demás funciones.

```
Prelude> succ (max 8 9)
10
Prelude> succ (max 8 (min 35 90))
36
```

En la primera línea devuelve el succecuente del máximo de 8 y 9, y en la segunda devuelve el succecuente del máximo de 8 y el mínimo de 35 y 90.

CREAR FUNCIONES

En haskell se pueden mandar llamar funciones para realizar funciones más complejas, estas funciones se hacen en un editor de texto agregándole la extensión .hs a los archivos.

En el bloc de notas o el editor que se esté utilizando se agregaran las variables y funciones de la siguiente manera.

```
triple x = x + x + x
```

El nombre de la función es triple, la variable es "x" y *los parámetros son "x + x + x".*

Haciendo esto una función para sacar el valor triple de un número.

Para cargar la función se utiliza el comando *:l "nombre de archivo".hs,* al cargar el archivo se notara el cambio en haskell ya que en pantalla ya no aparecerá *Prelude>* cambiara a **Main>.* Ya cargada la función se introduce el nombre de la función y el valor numérico que se le asignaría a la variable X.

```
Prelude> :l ejemplo.hs

[1 of 1] Compiling Main             ( ejemplo.hs, interpreted )
Ok, modules loaded: Main.

*Main> triple 3
9
```

También se puede combinar con otras funciones por ejemplo:

```
*Main> triple 3
9
*Main> succ (triple 3)
10
```

También se pueden hacer sumas llamando dos o más variables:

```
sumaNumeros x y = x + y
```

Se le agrega una segunda variable "Y", recibiendo "x" el primer valor y "y" el segundo

```
Prelude> :l sumaN.hs
[1 of 1] Compiling Main             ( sumaN.hs, interpreted )
Ok, modules loaded: Main.
*Main> sumaNumeros 10 5
15
*Main>
```

ESTRUCTURA IF

La primera forma de emplear estructuras condicionales en haskell es con la estructura if-then-else, su sintaxis es: **if** expresión lógica, **then** acción en caso positivo **else,** acción en caso negativo.

A continuación un ejemplo de una función con la sentencia if:

```
divisible x y = if (x `mod` y) == 0
                then "Son divisibles"
                else "No son divisibles"
```

La función divisible recibe dos valores, los cuales con la sentencia `mod` divide los valores y se pregunta si el residuo es igual a 0, la función if pregunta si el residuo es igual a 0 imprimirá el mensaje "Son divisibles" y si no es así "No son divisibles".

```
Prelude> :l ejm.hs
 [1 of 1] Compiling Main             ( ejm.hs, interpreted )
Ok, modules loaded: Main.
*Main> divisible 3 3
"Son divisibles"
*Main> divisible 7 5
"No son divisibles"
```

Un ejemplo más sencillo seria:

```
mayor x y = if x > y
        then "Es mayor"
        else "No es mayor"
```

La función mayor pregunta si x es mayor que y, si es así devolverá el mensaje “Es mayor” si no devolverá el mensaje “No es mayor”.

```
Prelude> :l mayor.hs
[1 of 1] Compiling Main             ( mayor.hs, interpreted )
Ok, modules loaded: Main.
*Main> mayor 55 34
"Es mayor"
*Main> mayor 34 55
"No es mayor"
```

En esta función pregunta si x es mayor que 20 si lo es le sumara 20 si no lo es devolverá el primer valor.

```
sumadiez x = if x > 20
        then x + 10
        else x
```

```
Prelude> :l sumadiez.hs
 [1 of 1] Compiling Main             ( mayor.hs, interpreted )
Ok, modules loaded: Main.
*Main> sumadiez 19
19
*Main> sumadiez 25
35
```

LISTAS

Una lista es una estructura de datos que representa un **conjunto de datos de un mismo tipo**, es muy usada e importante en el lenguaje haskell.
Su declaración es muy simple, por ejemplo:

TIPO	SIGNIFICADO	EJEMPLO
INT	Representa una lista de enteros	[4, 5, 9, 25, 60]
CHAR	Representa una lista de chars	[‘l’, ’i’, ’n’, ’u’, ’x’]
BOOL	Representa una lista de valores booleanos	[True, False, True]
STRING	Representa una lista de strings	[“buenas”, ”noches”]
FLOAT	Representa una lista de flotantes	[2.5, 5.8, 4.3, 7.1]

Como dice anteriormente haskell solo acepta listas de datos de un mismo tipo asi que si una lista tiene datos de distintos tipos devolverá error por ejemplo una lista compuesta por números y letras [4, 5, 8, 1, ’a’]:

```
Prelude> [4, 5, 8, 1, 'a']

<interactive>:9:2: error:
    • No instance for (Num Char) arising from the literal ‘4’
    • In the expression: 4
      In the expression: [4, 5, 8, 1, ....]
        In an equation for ‘it’: it = [4, 5, 8, ....]
```

Para combinar o concatenar listas se suman igual que al concatenar Strings ya que una String por ejemplo “hola” haskell la entiende como [‘h’, ’o’, ’l’, ’a’] pero a una lista solo se le pueden sumar listas sumar una lista con un valor no es posible como *[1, 2, 3, 4] ++ 2*, haskell no lo permite y manda error pero si se encierra en corchetes haciéndolo una lista *[1, 2, 3, 4] ++ [2]* no habría problema ya que sería una lista de un solo elemento, por ejemplo:

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

Para hacerlo con listas con strings se hace de la misma manera:

```
Prelude> ['h','o'] ++ ['l','a'] ++ [' ','m','u'] ++ ['n','d','o']
   "hola mundo"
```

Hay otra forma de combinar listas utilizando el operador ":" por ejemplo:

```
Prelude> 1 : [3,4,5]
[1,3,4,5]
Prelude> 'H' : "ola mundo"
   "Hola mundo"
```

La parte "ola mundo" no ocupo ir entre corchetes ya que haskell toma las strings como una lista de caracteres de esta forma *['o', 'l', 'a', ' ', 'm', 'u', 'n' ,'d' ,'o']* pero si trata de agregar un valor con doble comilla por ejemplo en vez de *'H' sea "H"*.

```
Prelude> "H" : "ola mundo"

<interactive>:18:7: error:
    • Couldn't match type 'Char' with '[Char]'
      Expected type: [[Char]]
        Actual type: [Char]
    • In the second argument of '(:)', namely '"ola mundo"'
      In the expression: "H" : "ola mundo"
         In an equation for 'it': it = "H" : "ola mundo"
```

Haskell no lo permitiría ya que a una lista de caracteres solo permite caracteres y poniéndole comillas dobles es como agregar una cadena en una lista de caracteres.

INDICE EN LISTAS

Un índice de listas hace referencia a diferentes elementos de nuestra lista con la posición que ocupan, para utilizar los índices en listas se necesita una palabra reservada del lenguaje haskell que es “let”.

Let: Nos permite crear una lista ponerle un nombre y asociarlo y así poderlo usarlo con ese nombre. La sentencia seria *let x = [1, 2, 3],* entonces nuestra lista pasaría a llamarse “x” y podríamos usarla sin tener que escribirla completa.

```
Prelude> let x = [1,2,3]
Prelude> x
   [1,2,3]
```

Como se puede apreciar con la palabra reservada “let” se le asocio a “x” la lista [1, 2, 3].

Pero como en un array la lista también tiene posiciones quedando 1 en la posición 0, 2 en la posición 1 y 3 en la posición 2, y para hacer referencia a ellos se utiliza el operador “!!”, por ejemplo si en esta lista quiero hacer al elemento 1 me devolverá un 2, ejemplo:

```
Prelude>  x !! 1
   2
```

El operador tomo de la lista el valor que estaba en la posición 1 que era 2.

```
Prelude> x !! 3
   *** Exception: Prelude.!!: index too large
```

En este ejemplo manda error ya que nuestra lista no tiene valores hasta la posición 3 ya que es diferente longitud que posición y el operador “¡!” hace referencia a la posición.

Haskell también puede incluir lista de listas por ejemplo:

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

Aparecen separados ya que cada uno pertenece a elementos distintos de la lista ya que si se hace referencia por ejemplo a la posición 0 no solo devolvería el 1 si no también el dos ya que en la posición 0 está la lista [1, 2].

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

Pero ahora si lo que quieres es que solo te mande el 1 o alguna otra posición en específico entonces se haría de la siguiente manera:

```
Prelude> lista !!0 !!1
2
```

Como se puede apreciar devuelve un 2 accediendo dentro al elemento en el índice 0 y de ese accedo al elemento con el índice 1 en este caso seria 2. Para que devolviera 1 seria *lista !!0 !!0* o que devuelva el valor 3 seria *lista !!1 !!0*.

```
Prelude> lista !!0 !!0
1
Prelude> lista !!1 !!0
3
Prelude> lista !!1 !!1
4
```

FUNCIONES DE LISTAS

A continuación unos ejemplos de cómo aplicar funciones en listas:

La función “length” sirve para darnos la longitud o tamaño de una lista.

```
Prelude> let lista = [1,2,3,4,5,6,7,8,7]
Prelude> length lista
9
```

La función “lenght” me dice los elementos que tiene una lista, si yo tuviera *“let listaDeLista = [[‘a’, ’b’], [‘c’, ‘d’]]”*, quiero saber cuál es la longitud de *“length listadelista”* me devolverá 2 ya que *listadelista* se compondría de 2 elementos que también son listas.

```
Prelude> let listadelista = [['a','b'],['c','d']]
Prelude> length listadelista
2
```

Ahora si por ejemplo yo tengo una lista de longitud n y quiero sacar o saber el primer elemento y solo el primero se utilizaría la función “head” haciendo referencia a la cabeza de la lista y si quiero saber el resto de contenido de la lista utilizamos la función “tail”.

```
Prelude> head lista
1
Prelude> tail lista
[2,3,4,5,6,7,8,9]
```

Si quiero ver el cuerpo de la lista hago un *“tail lista”* y devuelve *“[2,3,4,5,6,7,8]”* todo lo que no es la cabeza, desde el segundo hasta el final, así que aquí dividimos la cabeza de la lista de todo el cuerpo con “head” y “tail”.

También se puede hacer lo mismo pero al revés, para saber el último elemento de la lista, pues pongo “last lista” y devuelve el ultimo 7 y si quiero saber todo lo que hay en el inicio de la lista pero hasta llegar hasta el final se usa “init lista” y me sale todo el contenido de la lista menos el ultimo.

```
Prelude> head lista
1
Prelude> tail lista
[2,3,4,5,6,7,8,9]
```

Así que estas dos “last”, “init” son las inversas de estas dos “head”, “tail” con head veo la primera y con “tail” veo el resto y con “last” el ultimo y con “init” el resto.

Hay unas funciones que nos pueden ayudar para, cambiar listas, sacar productos y/o cálculos con ellos.

La función reverse lo que hace es voltear o invertir prácticamente todo el contenido de la lista pero no invertir el valor si no la posición ejemplo:

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

Pero como pueden apreciar la lista invirtió todas sus posiciones y la lista sigue teniendo la misma longitud solo cambio de lugar sus valores. Ahora un sencillo ejemplo de la funcion reverse en una lista de listas:

```
Prelude> listadelista
["ab","cd"]
Prelude> reverse listadelista
["cd","ab"]
```

Como se puede apreciar el reverse cambio de posición las listas pero solo esoya que “c,d y a,b” mantienen su primer orden.

La función take lo que hace tal como su nombre lo indica toma cierto número de valores de una lista o más bien el que se les indique, por ejemplo si hago un take 3 me devolverá los primeros 3 numeros o caracteres que estén dentro de la lista por ejemplo.

```
Prelude> take 3 lista
[1,2,3]
Prelude> take 5 lista
[1,2,3,4,5]
Prelude> take 23 lista
[1,2,3,4,5,6,7,8,7]
```

En la última se hizo un "*take 23"* pero no tenemos 23 posiciones dentro de esa lista, lo que hace haskell es solamente aunque sea cierto que no tengas ese número de posiciones te manda todo el contenido de la lista.

Otra funcion parecida a take es drop pero la funcion de drop es todo lo contrario que take ya que lo que hace es devolverte el contenido de la lista excepto los valores que hayas indicado por ejemplo:

```
Prelude> drop 3 lista
[4,5,6,7,8,7]
Prelude> drop 5 lista
[6,7,8,7]
Prelude> drop 23 lista
[]
```

Como se puede apreciar primero se devuelve la lista menos los primeros 3, después los primeros 5 y como la lista "lista" no tiene 23 elementos lo que hace es devolvernos una lista vacía todo lo contrario a la función "take".

Con la función mínimum se puede hacer algo parecido que con la función *min*, lo que hace la función *min* es devolver el dato de menor rango entre 2 parámetros, pero no se le podía agregar más de dos parámetros, ahora si lo que quieres hacer es una comparación entre más parámetros se utilizan las listas y la función mínimum como por ejemplo:

```
Prelude> let listam = [5,6,88,45,77,1]
Prelude> minimum listam
1
```

Como se aprecia la función mínimum tomo el número de menor valor dentro la lista.

La función máximum como lo indica su nombre me devolverá el parámetro de mayor valor dentro la lista por ejemplo:

```
Prelude> let listam = [5,6,88,45,77,1]
Prelude> maximum listam
88
```

Ahora si se desea hacer cálculos con listas como sumar o multiplicar el contenido de una lista se hace de la manera siguiente:

```
Prelude> let lista = [1,2,3,4,5,6,7]
Prelude> sum lista
28
Prelude> 1+2+3+4+5+6+7
28
```

Lo que hizo haskell con la función sum es sumar todo el contenido de la lista, una lista con valores de 1 a 7 su total es 28 y también se puede utilizar para sacar el producto de la lista se utiliza la función product:

```
Prelude> let lista = [1,2,3,4]
Prelude> product lista
24
Prelude> 1*2*3*4
24
```

Si se quiere saber si cierta letra o número se encuentra de una lista se utiliza la función `elem`, lo que hace esta función es tomar el valor que introduzcas lo compara con la lista y pregunta si ese valor o carácter se encuentra en ella por ejemplo:

```
Prelude> lista
[1,2,3,4]
Prelude> 5 `elem` lista
False
Prelude> 3 `elem` lista
True
```

Nuestra lista contiene los números de 1 a 4 al principio cuando `elem` compara la lista con un 5 devuelve un false ya que no hay nuero 5 dentro de esa lista pero cuando se compara con un 3 devuelve un true.

También se puede utilizar con listas de listas solo que se ocupa ser un poco mas especifico por ejemplo:

```
Prelude> listadelista
["ab","cd"]
Prelude> 'b' `elem` listadelista!!0
True
Prelude> 'b' `elem` listadelista!!1
False
```

Lo que hace haskell es buscar a la letra b dentro de la posición 0 de la lista y devulve un true ya que ahí se encuentra b.

RANGOS

Rangos que podemos utilizar en nuestras listas, es decir, si queremos por ejemplo en nuestra lista solo tener todos los números impares o pares del 1 al 100 sería algo complicado ya que la lista se tendría que declarar de la siguiente manera [2, 4, 6, 8, 10….] seria muy tedioso lo bueno es que haskell hay un sistema de mini-inteligencia que sirve para que el compilador entienda que tipo de lista queremos generar por ejemplo:

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

Como se aprecia aunque no se le especifico que se quería una lista con solo los números pares del 1 a 100 haskell con su mini-inteligencia toma el patrón de la lista que declaramos y los completa, pero ¿qué pasaría si el total de la lista no llega solo a 100 sino que a 105?:

```
Prelude> let lista = [2, 4 .. 105]
Prelude> lista
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,4
8,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92
,94,96,98,100,102,104]
```

Haskell solo llega hasta el 104 ya que 105 no es un número par. Pero si yo agrego una lista de este tipo [2, 4, 8 .. 100] haskell me devolvería error ya que solo se pueden utilizar dos parámetros para hacer el salto pero como hace haskell para saber que es lo que se le pide, Bueno eso Haskell no lo puede saber y tampoco podemos darle varios parámetros como pistas, haskell lo que hace por defecto, es considerar la relación que hay entre los primeros 2 parámetros ya sea como suma o resta por ejemplo:

```
Prelude> let lista = [1, 5 .. 50]
Prelude> lista
[1,5,9,13,17,21,25,29,33,37,41,45,49]
Prelude> let lista = [4,7.. 50]
Prelude> lista
[4,7,10,13,16,19,22,25,28,31,34,37,40,43,46,49]
Prelude> let lista = [50,45 .. 0]
Prelude> lista
[50,45,40,35,30,25,20,15,10,5,0]
Prelude> let lista = ['a','c' .. 'z']
Prelude> lista
"acegikmoqsuwy"
Prelude> let lista = ['a','b' .. 'z']
Prelude> lista
"abcdefghijklmnopqrstuvwxyz"
```

Se puede manejar de 5 en 5 de 3 en 3 o incluso saltarse letras del alfabeto o mandar el alfabeto completo, pero si al hacer esto no se le asigna un rango limite o tope lo que hara haskell es mandarnos una lista infinita de caracteres por ejemplo *[2,4 ..]* haskell nos arrojara una lista con valores de 2 en 2 hasta el infinito y son llamadas listas infinitas.

FUNCIONES CON LISTAS INFINITAS

Hay algunas funciones, que quizás parezcan un poco ridículas o no tienen mucho sentido, por ejemplo, la función repeat de algún valor por ejemplo *repeat 4* haskell me mandara constantemente el numero 4 repitiéndolo una y otra vez pero no de forma infinita.

La función cycle lo que hace es lo mismo que repeat pero con listas por ejemplo:

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

Otra función es “replicate”, y lo que hace es replicar un número, carácter o lista el número de veces que se le haya indicado y meterlo en una lista, por ejemplo:

```
Prelude> replicate 4 22
[22,22,22,22]
Prelude> replicate 5 'd'
"ddddd"
Prelude> replicate 2 ['f','s']
["fs","fs"]
```

También se pueden combinar las funciones repeat y cycle con otras por ejemplo:

```
Prelude> take 3 (cycle "hola")
"hol"
Prelude> take 6 (cycle "hola mundo")
"hola m"
Prelude> take 80 (cycle "hola mundo")
   "hola mundohola mundohola mundohola mundohola mundohola
   mundohola mundohola mundo"
```

Aquí lo que hace haskell es que toma solo los valores indicados por la funcion take dándole asi un rango o tope a nuestra lista, y funciona de igual manera con el repeat:

```
Prelude> take 3 (repeat 2)
[2,2,2]
Prelude> take 11 (repeat 2)
[2,2,2,2,2,2,2,2,2,2,2]
```

LISTAS INTENCIONALES

Haskell incluye una sintaxis especial para representar de forma más compacta operaciones sobre listas.

Estructura de una lista intencional:

Let lista = [Parámetro de salida | Lista filtro, Condiciones]

EJEMPLO:

[(x, y) | x <- xs, y <- xs]

Debe leerse como "para cada elemento x de xs y para cada elemento y de ys, debemos devolver la pareja (x, y)". Nótese que lo que estamos haciendo no es algo equivalente a lo que hace la función zip. Lo que realmente se está haciendo es combinar cada posible elemento de la primera lista con cada posible elemento de la segunda. Es decir, si ejecutamos

[(x, y) | x <- [1, 3], y <- [2, 5, 8]]

Obtendremos como resultado [(1,2), (1,5), (1,8), (3,2), (3,5), (3,8)]. Es decir, cada elemento de la primera lista se combina con cada elemento de la segunda, y el orden en el que se hace es exactamente el mostrado.

```
Prelude> [(x, y) | x <- [1,3], y <- [2,5,8]]
[(1,2),(1,5),(1,8),(3,2),(3,5),(3,8)]
```

Ahora supongamos que deseamos los números impares de 1 a 30, se haría de la manera siguiente:

```
Prelude> [x | x <- [1..30], x `mod` 2 == 1 ]
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29]
```

X pasa a tomar los valores de 1 a 30, después de tomar un valor la función `mod` lo divide y saca el residuo, si el residuo es uno muestra el valor, pero y si queremos los números pares y no solo eso sino que también multiplique por 10 cada elemento de la lista, se haría de la manera siguiente:

```
Prelude> [x*10 | x <- [1..30], x `mod` 2 == 0 ]
[20,40,60,80,100,120,140,160,180,200,220,240,260,280,300]
```

También se pueden hacer y llamar funciones externas con listas intencionales:

```
cuentaCifras lista = [if x < 10 then "una cifra" else "dos cifras"
                      | x <- lista
                      , odd x ]
```

nota: odd es un comando que pregunta si el valor es impar.

```
Prelude> :l cuenta.hs
[1 of 1] Compiling Main             ( cuenta.hs, interpreted )
   Ok, modules loaded: Main.
*Main> cuentaCifras [1..30]
*Main> ["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"]
```

LISTAS INTENCIONALES DOBLES

Después de ver la estructura de una lista intencional, ahora vamos a ver un poco cómo podemos hacer para combinar varias. La estructura es la misma que se ha utilizado la diferencia es que si antes solo utilizábamos una lista pero ahora usaremos dos, por ejemplo:

```
Prelude> [x+y| x<- [1..20], y<-[1..100], x<10, y `mod` 10 == 0]
[11,21,31,41,51,61,71,81,91,101,12,22,32,42,52,62,72,82,92,102,13,
23,33,43,53,63,73,83,93,103,14,24,34,44,54,64,74,84,94,104,15,25,3
5,45,55,65,75,85,95,105,16,26,36,46,56,66,76,86,96,106,17,27,37,47
,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]
```

¿Bien?, entonces, ¿Qué tenemos aquí?, pues, tengo por un lado una lista del 1 al 20 con X y otra del 1 al 100 con la Y. Y, técnicamente lo que hace es: tomar de la primera lista de X siempre que X sea menor que 10; así que así que cogeremos del 1 al 9 por otro lado, se tomara de Y, los números que sean divisibles por 10; es decir, el 10, 20, 30, 40, al final toma los valores de cada lista que cumpla las condiciones y los suma. Así que saldrá, pues, el 1 con el 10 me saldrá 11, el 1 con el 20 me saldrá 21, el 1 con el 30, 31 y así hasta llegar al tope de la lista.

También se puede utilizar para sacar por ejemplo cuantas veces se repite una letra o vocal en una oración, por ejemplo si quisiera que me devolviera las letras “C” que se encuentren en una oración se haría de la siguiente manera:

```
letras frase = [letra | letra<-frase, letra == 'c']
```

```
Prelude> :l cuenta.hs
[1 of 1] Compiling Main             ( cuenta.hs, interpreted )
Ok, modules loaded: Main.
*Main> letras "la calle cuenta con carros"
"cccc"
```

También se puede saber las vocales de la oración y no solo la letra “c”:

```
letras frase = [letra | letra<-frase, letra `elem`['a','e','i','o','u'] ]
```

```
*Main> :r
[1 of 1] Compiling Main             ( cuenta.hs, interpreted )
Ok, modules loaded: Main.
*Main> letras "la calle cuenta con carros"
"aaeueaoao"
```

Y si lo que quieres no es que te mande las letras que tenga si no el número de veces que esta c en la frase se hace de la manera siguiente:

```
mostrarC frase = [letra | letra<-frase, letra == 'c']

sumarC cadenaC = sum[1|    x<-(mostrarC cadenaC)]
```

Como pueden observar no se limita a solo una operación si no que se pueden realizar varias y hasta llamados.

```
Prelude> :l cuenta.hs
 [1 of 1] Compiling Main             ( cuenta.hs, interpreted )
Ok, modules loaded: Main.
*Main> sumarC "la calle cuenta con carros"
4
```

TUPLAS VS LISTAS

Tupla: es la colección de datos que permite mesclar tipos de datos, algo que en listas no se puede hacer. A diferencia de las listas que se declaran con "[]" las tuplas o duplas se declaran con "()".

Ejemplo:

```
Prelude> let lista = [1,2]
Prelude> let lista = [1,"dos"]

<interactive>:65:14: error:
    • No instance for (Num [Char]) arising from the literal ‘1’
    • In the expression: 1
      In the expression: [1, "dos"]
      In an equation for ‘lista’: lista = [1, "dos"]
Prelude> let dupla = (1,2)
Prelude> let dupla = (1,"dos")
Prelude> dupla
(1,"dos")
```

Diferencia entre lista de listas y lista de duplas ya que una lista de listas podias tener varias listas de varios tamaños sin importar como por ejemplo *[[1, 2], [3, 4, 5, 6], [7, 8, 9]]* pero si se trata de hacer lo mismo que las duplas declarando de esta manera *((1, 2), (3, 4, 5, 6), (7, 8, 9))*nos mandara error ya que las listas de duplas deben tener la misma longitud algo que no aplica en lista de listas.

```
Prelude> [[1,2],[3,4,5,6],[6,8,9]]
[[1,2],[3,4,5,6],[6,8,9]]
Prelude> [(1,2),(3,4,5,6),(6,8,9)]

<interactive>:76:8: error:
    • Couldn't match expected type ‘(t, t1)’
                  with actual type ‘(Integer, Integer, Integer,
Integer)’
    • In the expression: (3, 4, 5, 6)
      In the expression: [(1, 2), (3, 4, 5, 6), (6, 8, 9)]
      In an equation for ‘it’: it = [(1, 2), (3, 4, 5, 6), (6, 8,
```

Todas las Tuplas cuando pertenezcan a la misma lista deben tener el mismo tamaño y formato por ejemplo:

```
Prelude> [(1,"dos"),(3,"cuatro"),(5,"seis")]
[(1,"dos"),(3,"cuatro"),(5,"seis")]
```

Aquí el formato seria por dupla de un valor entero y una cadena si ese patrón se altera mandara error.

FUNCIONES DE DUPLAS

Estas funciones exclusivas para duplas, que nos sirven para devolvernos el elmento que este en la posición que deseamos encontrar, un ejemplo sencillo seria el operador "!!" que nos mandaba el valor de la posición que le pidiéramos, sin embargo en duplas no funciona asi ya que manda error, para poder hacer esto con duplas se hace de la manera siguiente:

Si quisiéramos sacar el primer elemento de una dupla se utiliza "fst" que es una abreviación de first *fst (1,"tres")* y nos arrojaría el 1 para sacar la segunda posición seria *snd* abreviación de second *snd(1,"tres"),* y asi lo mostraría haskell:

```
Prelude> fst (1,3)
1
Prelude> fst (1,"tres")
1
Prelude> snd (1,"tres")
"tres"
```

LISTAS DE DUPLAS ZIP

Si deseo sumar o concatenar 2 listas con tipos de datos distintos por ejemplo una con datos numéricos y otra con datos string se utiliza la función zip, empareja o las mezcla la posición 1 de la lista 1 con la posición 1 de la lista 2.

```
Prelude> let numeros =[[1,2], [3,4],[5,6,7]]
Prelude> let letras =[['a','b'], ['c','d'],['e','f','g']]
Prelude> zip letras numeros
[("ab",[1,2]),("cd",[3,4]),("efg",[5,6,7])]
```

Siendo 2 listas de distintos tipos la función zip puede concatenarlas sin problemas pero ¿Qué pasaría si las listas no fueran de la misma longitud? Ósea que una lista sea más larga que la otra:

```
Prelude> let numeros =[2,3,4,5,6,7,8,9]
Prelude> let letras =["juan","pedro","martin"]
Prelude> zip letras numeros
[("juan",2),("pedro",3),("martin",4)]
```

Lo que hace haskell es cortar la lista más larga para poder hacer el emparejamiento. Y si llegáramos a usar una lista infinita bueno pues de igual manera aunque sea infinita la función zip solo tomara el mismo número de valores que la lista con la que se le empareje por ejemplo:

```
Prelude> let numeros =[1,2..]
Prelude> let letras =["juan","pedro","martin"]
Prelude> zip letras numeros
[("juan",1),("pedro",2),("martin",3)]
```

Como la lista letras solo tiene 3 elementos no importa que la lista números sea infinita la función zip solo toma los que necesita.

COMANDO T (tipos de datos)

Esta función nos permite al utilizarla saber el tipo de dato que es de un valor en específico por 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 6.5
6.5 :: Fractional t => t
```

Pero eso no es todo lo que puede hacer la función :t sino que también puede indicarte el formato de las funciones por ejemplo:

```
Prelude> :t head
head :: [a] -> a
Prelude> :t tail
tail :: [a] -> [a]
Prelude> :t min
min :: Ord a => a -> a -> a
Prelude> :t max
max :: Ord a => a -> a -> a
Prelude> :t minimum
minimum :: (Foldable t, Ord a) => t a -> a
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t snd
snd :: (a, b) -> b
```

CONVERSORES SHOW Y READ

SHOW: la función show nos sirve para transformar cualquier tipo de dato a cadena de texto ya sea número entero, flotante, un valor booleano o incluso una lista:

```
Prelude> show 7
"7"
Prelude> show True
"True"
Prelude> show 4.44433
"4.44433"
Prelude> show [1,1,1]
"[1,1,1]"
```

Como pueden ver sin importar el tipo de dato lo convirtió a cadena de caracteres incluso la lista ya que al devolverla con los corchetes paso de ser una lista a una cadena de caracteres.

READ: nos permite cambiar a una variedad de tipos de datos y no solo strings como “show”, Read toma el primer parámetro y lo toma como si fuera el tipo del segundo dato por ejemplo:

```
Prelude> read "5.6"+3.5
9.1
Prelude> read "False" && True
False
Prelude> read "[1,2,3]"++[4,5,6]
   [1,2,3,4,5,6]
```

La función read toma el primer parámetro y lo transforma al mismo tipo del segundo parámetro.

CONCLUSION

Haskell es un lenguaje seguro y firme el cual tiene errores escasos, al cometer un error es muy específico al informarte para que puedas solucionar tu problema fácilmente, tiene muchas herramientas y funciones interesantes que hacen agradable el explorarlas ya que al principio no parece poder hacerse mucho con este lenguaje pero es uno muy completo.