HASKELL

MANUAL

YESENIA ROBLES

NOVIEMBRE 2016

HASKELL

Manual teórico-práctico

Yesenia Enith Robles Rivera

Noviembre del 2016

ÍNDICE

INTRODUCCIÓN

El presente trabajo estará dividido en 18 temas. En cada uno de ellos habrá ejemplos en donde se apliquen los conceptos descritos, es recomendable que se pongan en práctica conforme se vaya avanzando en la lectura de este documento. Cada ejemplo esta explicado detalladamente, para entender el funcionamiento de cada línea de código.

Este manual va dirigido para toda aquella persona que esté interesado en aprender un lenguaje más de programación, este es muy sencillo y simple. Haskell puede ayudarnos bastante por ejemplo en las matemáticas podemos crear una función que nos calcule fácilmente alguna operación, en comparación de hacer en cuaderno o en calculadora varias veces el mismo procedimiento llevándonos a la pérdida de tiempo.

Por ello me decidí a realizar este manual de tal manera que las personas que lo consulten o lo lean, se interesen en esta parte de la computación. Los ejemplos que contiene el manual, se han hecho para que el lector no le parezca tedioso si no algo que le llame la atención.

Manual para aprender Haskell

El lenguaje recibe su nombre en honor a Haskell Brooks Curry, por sus trabajos en lógica matemática que sirvieron como fundamento para el desarrollo de lenguajes funcionales. Haskell está basado en el *cálculo lambda*, por lo tanto el símbolo lambda es usado como logo.

¿Por qué usar Haskell?

Escribir programas grandes que funcionen correctamente es difícil y costoso. Mantener esos programas es aún más difícil y costoso también. Los lenguajes de programación funcional, tales como Haskell, pueden hacerlo mucho más fácil y económicos.

Actualmente este lenguaje no es de los más utilizados pero en algún momento fue necesario.

Desde la página oficial de Haskell (Fig. 1.0) es donde se descarga la plataforma de desarrollo, la instalación es muy sencilla solo tienen que seguirse los pasos seleccionar siguiente hasta finalizar.

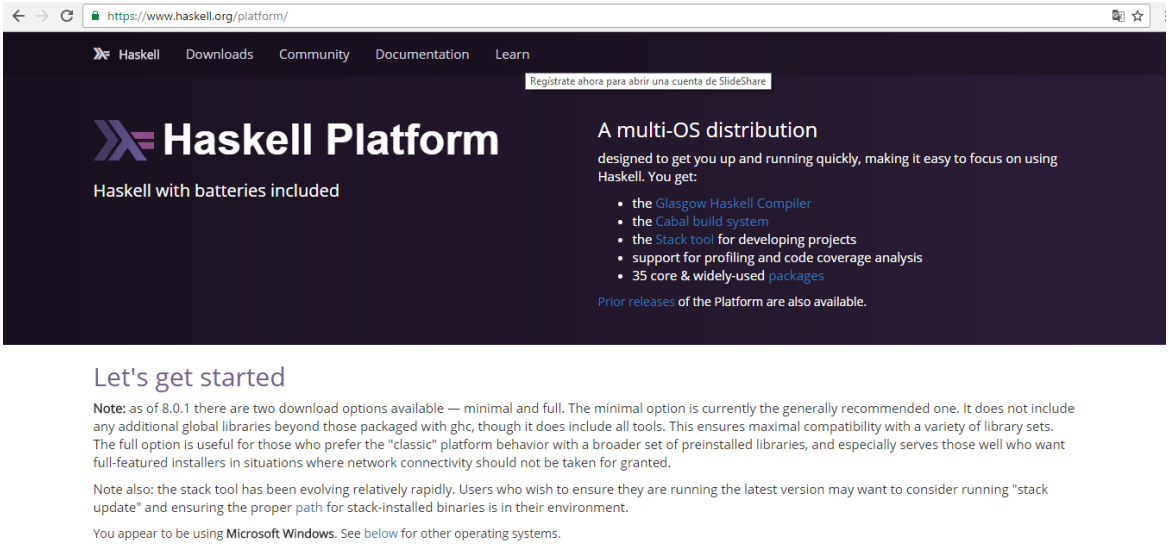

Fig. 1.0 | Pagina oficial de plataforma Haskell.

Una vez descargada e instalada la plataforma de desarrollo es muy sencilla de utilizar, en la Fig. 1.1 se muestra la plataforma de desarrollo, donde se puede escribir el código Haskell o también se pueden utilizar archivos donde se tenga guardado el código.

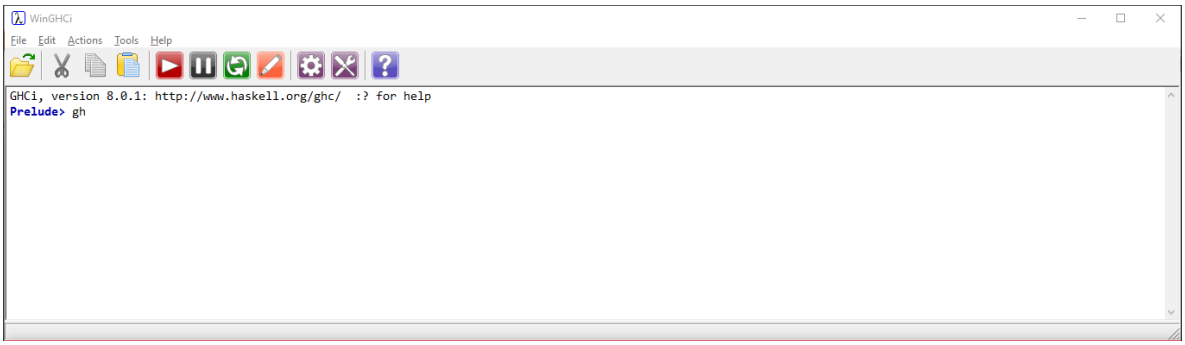

Fig. 1.1 | Plataforma de desarrollo Haskell (WinGHCi).

En la Fig. 1.2 se puede apreciar el compilador Aquí es donde se utilizan los archivos, los archivos para que puedan ser compilador se guargan con la extencion **.hs.**

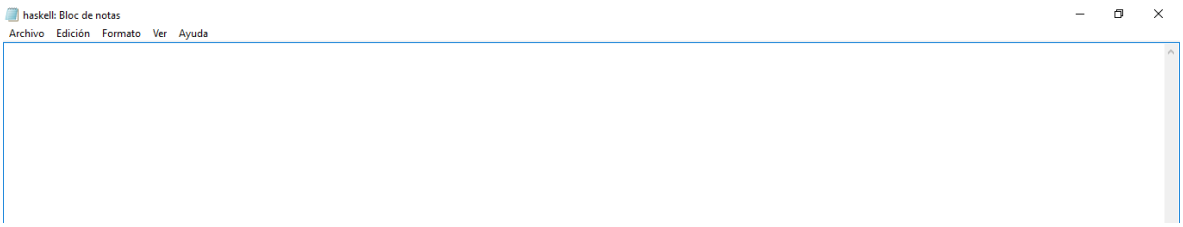

Fig. 1.2 |

Operadores logicos, comparativos y aritmeticos

Se pueden utilizar operadores aritmeticos para sumar, restar, dividir, etc. Tal como se muestra en el ejemplo (Fig. 1.3).

```
Prelude> 2 * 10
20
Prelude> 15 - 10
5
Prelude> 6 / 2
3.0
Prelude> 5 + 1
6
```

Fig. 1.3 | Operaciones aritméticas.

En otro caso como el de una división se puede mandar llamar una función donde `div` muestra el cociente entero y `mod` muestra el residuo de la operación, Fig. 1.4.

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

Fig. 1.4 | Operaciones con funciones.

También se puede utilizar varias operaciones en una línea de código siguiendo la jerarquía de signos como en el ejemplo donde se utilizan paréntesis, Fig. 1.5

```
Prelude> (10 * 50) -100
400
Prelude> 10 * 50 - 100
400
```

Fig. 1.5 | Operaciones con paréntesis.

Haskell trabaja con operadores lógicos True y False pero con su inicial mayúsculas, un ejemplo de escribir mal el operador, Fig. 1.6.

```
Prelude> true
<interactive>:18:1: error:
    • Variable not in scope: true
    • Perhaps you meant data constructor ‘True’ (imported from Prelude)
Prelude> True
True
Prelude> False
False
```

Fig. 1.6 | Operaciones lógicos.

Los operadores para decir la condición **Y** se representan así **&&** mientras que || representa **O, not** niega True a False y viceversa, Fig., 1.7.

```
Prelude> True && False
False
Prelude> True && True
True
Prelude> False || True
True
Prelude> not True
False
Prelude> not (False && True)
True
```

Fig. 1.7

Comprobación de igualdad:

```
Prelude> 5 == 5
True
Prelude> 4 /= 4
False
Prelude> "hola" == "hola"
True
```

Fig. 1.8 | igualación.

¿Qué sucede si sumamos 3 + “hola”? Sucede que Haskell no nos permite juntar estos datos y nos muestra un error, Fig. 1.9:

```
Prelude> 3 + "hola"

<interactive>:41:1: error:
    • No instance for (Num [Char]) arising from a use of ‘+’
    • In the expression: 3 + "hola"
      In an equation for ‘it’: it = 3 + "hola"
```

Fig. 1.9 | Error al sumar dos datos de distinto tipo.

Pero si puede concatenar dos textos tal que este es “Hola” ++ “Mundo”, como se darán cuenta estamos utilizando “**++**” para poder concatenar como se muestra, Fig. 1.10:

```
Prelude> "Hola" ++ "Mundo"
"HolaMundo"
```

Fig. 1.10 | Concatenación de dos textos.

Funciones básicas

Haskell tiene internamente unas listas de datos, entonces si le ingresas un dato este te devuelve el siguiente de esa lista que ya tiene almacenada.

Succ2: esta función devuelve el número consecutivo al que ingresamos, así como al introducir una letra:

```
Prelude> succ 10
11
Prelude> succ 'a'
'b'
```

Fig. 2.0 | numero consecutivo.

Min: esta función devuelve el número más pequeño de los dos parámetros que se ingresen, con se muestra en la Fig. 2.1:

```
Prelude> min 2 3
2
Prelude> min 10 2
2
```

Fig. 2.1 | Regresa el parámetro más pequeño.

Max: este es parecido a “min” al igual recibe dos parámetros pero este devuelve el valor más grande como se muestra en la Fig. 2.2:

```
Prelude> max 102 20
102
Prelude> max 10 20
20
```

Fig. 2.2 | Regresa el parámetro más mas grande.

¿Pudieran combinarse estas dos funciones?

Bien, claro que pueden combinarse pero debe tomarse en cuenta que Haskell empieza a leer de izquierda a derecha, entonces si queremos que primero se ejecute la función de la derecha entonces debemos poner su función entre paréntesis espacio y el parámetro. Se ejecuta primero lo que está dentro de los paréntesis y después lo de fuera. Se muestra en el ejemplo la forma correcta de combinar las funciones y otro ejemplo donde se muestra el cómo no utilizar las

```
Prelude> succ (max 8 6)
9
Prelude> succ 1 max 8 6
<interactive>:64:1: error:
    • Non type-variable argument
        in the constraint: Num ((a -> a -> a) -> t -> t1 -> t2)
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a t t1 t2.
              (Ord a, Num ((a -> a -> a) -> t -> t1 -> t2), Num t1, Num t,
               Enum ((a -> a -> a) -> t -> t1 -> t2)) =>
              t2
```

Fig. 2.3 | combinación de funciones.

Crear funciones

Vamos a crear funciones en archivos externos y después haremos la llamada para utilizarlos.

Paso 1: desde el compilador creamos la función en este caso llamada **sumaDiez** donde este recibirá un parámetro **x** y esta función será igual a **x** (lo que ingresen) **+ 10**. Entonces esto quiere decir que la función **sumaDiez** es igual a **x + 10** tal como se muestra en el ejemplo de la Fig. 2.4:

```
sumaDiez x = x + 10
```

Fig. 2.4 | creando una función.

Después de crear la función **sumaDiez** nos vamos a la plataforma y cargamos el archivo, ¿Cómo hacemos esto?, ingresamos → **:l** y el nombre del archivo:

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

Fig. 2.5 | compilando el archivo externo.

Ahora mandamos llamar la función **sumaDiez** le pasamos el parámetro 7, entonces le resultado seria **17** como se muestra en la siguiente Fig. 2.6.

```
*Main> sumaDiez 7
17
```

Fig. 2.6 | llamando a la función.

Además esta función externa se puede combinar con una función básica:

```
*Main> succ (sumaDiez 14)
25
```

Fig. 2.7 | Combinación de funciones.

Si se requiere modificar la función externa en este caso **sumaDiez:** abrimos nuestro archivo aplicamos los cambios necesarios: **sumaDiez x = x – 10** guardamos ahora desde la plataforma volvemos a cargar (Fig. 2.8):

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

Fig. 2.8 | Cargando de nuevo el archivo.

Llamamos a la función, ingresamos el numero 20 entonces se muestra el resultado, Fig. 2.9:

```
*Main> sumaDiez 20
10
```

Fig. 2.9 | Mostrando el cambio de la función.

Una manera más sencilla de volver a cargar el archivo es **:r**

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

Fig. 2.10 | Recargando la función.

Estructura IF

Iniciamos creando una función llamada **divisible,** la cual analizara si dos números son divisibles. Utilizando dos variables **X** y **Y**, entonces el valor de la función será igual a si el resultado (x `mod` y) es igual a 0 entonces el valor que tendrá la función divisibles es un mensaje “Son divisibles”, y si no entonces mostrara otro mensaje “No son divisibles”.

divisible x y = if (x `mod` y) == 0

then "Son divisibles"

else "No son divisibles"

```
*Main> :load "if01.hs"
[1 of 1] Compiling Main             ( v05_IF.hs.hs, interpreted )
Ok, modules loaded: Main.
*Main> divisible 10 5
"Son divisibles"
*Main> divisible 10 9
"No son divisibles"
*Main> divisible 10 9
```

Fig. 3.0 | Mostrando el cambio de la función.

NOTA: en Haskell es obligatorio que todo IF tenga su then y else, indicando que sucede si es correcta la condición o si no es correcta.

Otro ejemplo sencillo una función que regresa si el parámetro **x** es mayor que el parámetro **y,** Fig. 3.1:

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

```
*Main> :load "if02.hs"
[1 of 1] Compiling Main             ( if02.hs, interpreted )
Ok, modules loaded: Main.
*Main> esMayor 7 5
"Es mayor"
*Main> esMayor 7 10
"No es mayor"
```

Fig. 3.1 | Comparación de dos parámetros cual es mayor.

Por ultimo otro ejemplo de la utilización de la estructura IF, donde sumara 10 solo a los números mayores a 20. La función se llamara **sumaDiezAMayoresQueVeinte** este recibe un parámetro **X**, si X es mayor que 20 entonces el valor que tendrá la función será X +10, pero si no es así el valor que tendrá la función será **X** no cambiara, Fig. 3.2.

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

```
*Main> :load "if03.hs"
[1 of 1] Compiling Main             ( if03.hs, interpreted )
Ok, modules loaded: Main.
*Main> sumaDiezAMayoresQueVeinte 5
5
*Main> sumaDiezAMayoresQueVeinte 30
40
```

Fig. 3.2 | Comparación de dos parámetros cual es mayor.

Listas

Anteriormente hablamos de que Haskell tiene implementadas unas listas por defecto, y se pueden mandar llamar las funciones para utilizarlas. Un ejemplo de ello se tiene una lista de dos números y una lista de letras cuando utilizábamos la función **succ** nos devuelve el siguiente número o letra que ingresemos.

¿Cómo se pueden almacenar listas de datos?

Una nota importante es: en una lista de datos no debemos mezclar datos. Creemos una lista, esta se define por contener los datos dentro de corchetas "**[ ]**", [5,2,6,9] haciendo referencia a lo anterior comentado si ponemos una letra dentro de esa lista de números mandara un error porque no pueden mezclarse los datos en la siguiente Fig. 4.0 se puede observar de una manera más clara:

```
Prelude> [5,2,6,9]
[5,2,6,9]
Prelude> [5,2,6,9,'f']

<interactive>:104:2: error:
    • No instance for (Num Char) arising from the literal ‘5’
    • In the expression: 5
      In the expression: [5, 2, 6, 9, ....]
      In an equation for ‘it’: it = [5, 2, 6, ....]
```

Fig. 4.0 | Creando listas.

Para sumar dos listas para tenerlas juntas, anteriormente se habló del como concatenar cadenas, se utiliza el operador ++

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

Fig. 4.1 | Concatenación de cadenas.

Esto sucede porque las cadenas se consideran listas, porque al insertar lo anterior Haskell entiende esto, (Fig. 4.2):

```
Prelude> ['h','o','l','a']
"hola"
Prelude> [1,2,3,4] ++ [5,6,7,8]
[1,2,3,4,5,6,7,8]
```

Fig. 4.2 | Lista.

Un error seria que intentemos agregarle un número que no está en una lista a una lista, Fig. 4.3:

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

<interactive>:111:1: 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]
Prelude> [1,2,3,4] ++ [8]
[1,2,3,4,8]
```

Fig. 4.3 | Error de concatenación.

Otra manera de ingresar un número a una lista de números o una letra a una lista de letras es por ejemplo:

```
Prelude> 0 : [1,2]
[0,1,2]
```

Fig. 4.4 | Listas.

Se puede observar que al ingresar de esta manera este se pone a la cabeza de la lista.

Ahora veamos con una cadena, estas son consideradas caracteres entonces es una lista de caracteres y estos van con comillas simples **‘ ‘**, como se muestra en la Fig. 4.4:

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

Fig. 4.5| Caracteres y listas de caracteres.

Índice en listas

Let es una palabra reservada de Haskell, que nos permite crear una lista, asignarle un nombre asi como se muestra, (Fig. 5.0):

```
Prelude> let lista = [6,7,8]
Prelude> lista
[6,7,8]
```

Fig. 5.0| Almacenar una lista en una variable.

Esto es un arreglo donde cada uno de los elementos de un arreglo los ocupa una posición tiene un índice, la primera es la posición 0, la segunda es la posición 7, etc.

para hacer que nos regrese una posición de la lista se utilizan los signos de admiración "**!!**", en este caso llamaremos al primer elemento que ocupa la posición 0 y en caso se pedir la posición 3 este lanzaría un error porque no existe la posición 3, tal como se muestra en la siguiente figura (Fig. 5.1):

```
Prelude> lista !!0
6
Prelude> lista !!3
*** Exception: Prelude.!!: index too large
```

Fig. 5.1| Solicitando posiciones con **!!**.

Nota: Es muy diferente la longitud a la posición de los elementos de una lista.

Bueno sigamos con las listas, Haskell puede tener listas de listas es decir una lista dentro de otra lista como se ve en la Fig. 5.2, de tal manera que si solicitamos la posición !!0 esta nos devolverá la primer lista [1.2]:

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

Fig. 5.2| Lista de listas.

Probablemente recordemos que dentro de una lista no podemos mezclar diferentes datos, y en lista de listas tampoco se puede mezclar.

Volviendo al ejemplo anterior es ¿posible pedir solicitar la primera lista y su segundo elemento? Si, esto es posible, recordemos que Haskell ejecuta de izquierda a derecha, entonces primero solicitamos la lista de la posición 0 y después el elemento de la posición 1, `lista = [[1,2] , [3,4]]` estos es:

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

Fig. 5.3| Lista de listas.

Funciones de listas

Veremos diferentes funciones de administración de listas, primero creamos una lista llamada **lista** que contiene los números del 1 al 7, y si queremos saber cuántos elementos tiene la lita o que tan larga es llamamos la función “length lista” que devolverá que tiene 7 elementos la lista pero recordemos que la lista comienza desde la posición 0, entonces el último elemento de la lista en este caso el número 7 se encuentra en la posición 6 de la lista, véase el ejemplo en la Fig. 6.0:

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

Fig. 6.0 | longitud y posiciones de los elementos de una lista.

Entonces devuelve cuantos elementos contiene una lista, y si creamos una lista de lista de caracteres:

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

Fig. 6.1 | lista de listas y sus posiciones.

Entonces es correcto nos dice que tiene 2 elementos porque contiene dos diferentes listas dentro de la lista tal como se ve en la Fig. 6.1. Bien y si queremos que nos devuelva la cabeza de la lista tendremos que utilizar la función “**head**” esta hace referencia a la cabeza de la lista veamos el ejemplo expresado en la Fig. 6.2:

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

Fig. 6.2 | Cabeza de una lista y resto de la lista sin cabeza.

Y si queremos saber cuál es el cuerpo de la lista, por medio de la función "**tail**" (Fig. 6.2) muestra todo lo que no es la cabeza desde el segundo elemento hasta el final, dividiendo la cabeza de la lista del resto del cuerpo con "**head**" y "**tail**".

Para solicitar el último elemento de la lista se utiliza "**last**" y muestra el último elemento pero si además se quiere mostrar todo menos el último usamos "**init**" (como se muestra en la Fig. 6.3)

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

Fig. 6.3 | Ultimo elemento de la lista y lista sin el último elemento.

Cálculos con Funciones con listas

Con ayuda de estas funciones podemos cambiar listas, obtener productos, realizar cálculos, etc. Veamos ejemplos, creamos dos listas una de letras **listaDeLista** donde esta contiene dos listas internas y otra de números **lista** que contiene siete elementos, como se ve en la Fig. 7.0:

```
Prelude> let listaDeLista = [['a','b'] , ['c','d']]
Prelude> let lista = [1,2,3,4,5,6,7]
Prelude> reverse lista
[7,6,5,4,3,2,1]
Prelude> lista
[1,2,3,4,5,6,7]
Prelude> reverse listaDeLista
["cd","ab"]
Prelude> listaDeLista
["ab","cd"]
```

Fig. 7.0 | Función reverse.

Reverse es una función que permite voltear el elemento, es decir, como el ejemplo de la Fig. 7.0 al aplicarle la función esta volverá la lista pero solo para mostrarla porque la lista si la volvemos a invocar sigue como la declaramos anteriormente, al igual sucede con la lista de letras solo que como tiene dos listas internas este solo voltea las listas internas y los elementos no los mueve.

Otra función de muchas es take, nos muestra el número de elementos que solicitamos de la lista (Fig. 7.1):

```
Prelude> take 3 lista
[1,2,3]
Prelude> take 100 lista
[1,2,3,4,5,6,7]
```

Fig. 7.1 | Función take.

Pedimos que muestre los 100 elementos de la lista con take pero como la lista no tiene tantos elementos solo nos muestra los que tiene. Al igual que reverse solo es para mostrar lo que queremos de la lista, esta no se ve alterada.

Ahora si queremos que de la lista solo nos quite los primeros n elementos y nos muestre el resto entonces para ello se requiere utilizar la función drop, observe el ejempló de la Fig. 7.2:

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

Fig. 7.2 | Función drop.

Con estos ejemplos ya vimos el cómo administrar lo que se muestra de una lista o lo que no.

A estas listas también se le puede pedir el máximo o el mínimo elemento, anterior mente vimos que una función básica puede hacer esto, pero si lo hacemos así: “min lista” Haskell nos regresaría un error enorme, entonces para las listas se utiliza el **mínimum y máximum** véase en el ejemplo de la Fig. 7.3 como se utilizan estas:

```
Prelude> lista
[1,2,3,4,5,6,7]
Prelude> minimum lista
1
Prelude> maximum lista
7
Prelude> maximum [1,5,10,2,0]
10
```

Fig. 7.3 | Minimum y maximum.

Este mostro que el mínimo de la lista es el elemento de la posición cero que es el número 1, y el máximo es el elemento de la posición seis el número 7. Otra manera es pasarle al máximum directamente una lista, aquí los elementos no están ordenados se puede ver que funciona perfectamente. También se puede utilizar con la lista de letras veamos en la Fig. 7.4:

```
Prelude> let listaDeLista = [['a','b'] , ['c','d']]
Prelude> maximum listaDeLista
"cd"
Prelude> minimum listaDeLista
"ab"
```

Fig. 7.4 | Minimum y máximum.

¿Cómo podemos hacer cálculos con las listas?

comencemos con realizar una suma, utilizamos **sum** como ejemplo sumaremos todos los elementos de la lista de números (Fig. 7.5):

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

Fig. 7.5 | suma de los elementos de la lista de números.

O también podemos obtener el producto (multiplicación) de una lista, es decir, se multiplican todos los elementos, por ejemplo: product [1,2,3] como en la Fig. 7.6:

```
Prelude> product [1,2,3]
6
```

Fig. 7.6 | Producto de los elementos de la lista de números.

¿Cómo podemos preguntar si algún elemento se encuentra en la lista? Con `elem` preguntamos si el numero pertenece a la lista y si es correcto devuelve un True de los contrario devolvería un False, ejemplo en la Fig. 7.7:

```
Prelude> lista
[1,2,3,4,5,6,7]
Prelude> 3 `elem` lista
True
Prelude> 10 `elem` lista
False
```

Fig. 7.7 | Elementos que pertenecen a una lista o no.

Aplicando `elem` a la lista de letras se vería de la siguiente forma, Fig. 7.8:

```
Prelude> listaDeLista
["ab","cd"]
Prelude> "cd" `elem` listaDeLista
True
Prelude> 'c' `elem` listaDeLista

<interactive>:181:12: error:
    • Couldn't match type ‘[Char]’ with ‘Char’
      Expected type: [Char]
        Actual type: [[Char]]
    • In the second argument of ‘elem’, namely ‘listaDeLista’
      In the expression: 'c' `elem` listaDeLista
      In an equation for ‘it’: it = 'c' `elem` listaDeLista
```

Fig. 7.8 | Elementos que pertenecen a una lista de letras.

Si preguntamos si "**cd**" pertenece nos dice que es correcto pero si preguntamos solo por el carácter '**c**' este nos regresa un error porque '**c**' solo es un elemento de una lista interna, la forma correcta de ver si esta es:

```
Prelude> 'c' `elem` listaDeLista!!1
True
```

Fig. 7.9 | Elementos que pertenecen a la lista interna de la lista.

Al preguntar por el carácter 'c' este nos devuelve un True, se debe a que preguntamos el carácter 'c' se encuentra en la listaDeListas en la posición 1, esta es la forma correcta.

Rangos en listas

Cuando en una lista queremos que todos los números pares, es mucha fatiga y complicado por ejemplo hasta el 100 "let lista = [2,4,6,8,10, etc….]"entonces para ello Haskell tiene un sistema de "mini-inteligencia" esto sirve para que el compilador entienda que tipo de lista se quiere generar. ¿Cómo se hace? Solamente se tiene que poner los dos primeros números de la lista y esta automáticamente interpreta el resto, para entenderlo mejor veamos el ejemplo de la Fig. 8.0:

```
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]
```

Fig. 8.0 | Números pares de una lista.

Nota: Solo se le pasan dos parámetros a la lista después de dos puntos seguidos y el número que marcara el límite dela lista.

Ahora si queremos decirle que la lista contenga los números que sean la potencia de dos, entonces pasamos los dos parámetros como lo mencionamos anteriormente [2,4 .. 50], si hacemos estos no nos dará lo que queremos Haskell no sabrá.

Esto también se aplica a la lista de letras, Fig. 8.1:

```
Prelude> let listaLetras = ['a','b'..'z']
Prelude> listaLetras
"abcdefghijklmnopqrstuvwxyz"
```

Fig. 8.1 | Lista de letras.

Funciones para listas infinitas

Un ejemplo de ello es si queremos repetir un elemento infinitamente con la función **repeat** espacio y el elemento que queremos, Fig. 9.0:

```
Prelude> repeat 7
,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
Prelude> Interrupted.
```

Fig. 9.0 | Elemento infinito.

Nota: para para el proceso presionamos las teclas ctrl + c.

Función "**ciclo**", esta lo que hace es repetir de manera indefinida la lista que le demos, por ejemplo cyclo [2,4,6], Fig. 9.1:

```
Prelude> cycle [2,4,6]
2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4
,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,2,4,6,
Prelude> Interrupted.
```

Fig. 9.1 | Cyclo.

Existe otra función **replicate,** aquí no se repite las veces que se le establezca por ejemplo si le decimos que se repita diez veces el número 60, Fig. 9.2”

```
Prelude> replicate 10 60
[60,60,60,60,60,60,60,60,60,60]
```

Fig. 9.2 | Replicate.

También se le puede decir que repita ciertas veces los caracteres de una lista como la ‘h’ y la ‘ o’, Fig. 9.3:

```
Prelude> replicate 2 ['h','o']
["ho","ho"]
```

Fig. 9.3 | Replicar caracteres de una lista.

Combinemos todos estos ejemplos, pidamos que repita infinitamente el número tres y que solo muestre cuatro elementos de esa, Fig. 9.4:

```
Prelude> take 4 (repeat 3)
[3,3,3,3]
Prelude> take 4 (cycle "Hola mundo")
```

Fig. 9.4 | Combinación de funciones.

Listas intencionales

Una lista intencional es una lista como las anteriores que creamos pero donde filtramos que elementos de las listas queremos según las condiciones que aplicamos, para entenderlo mejor véase en el ejemplo, Fig. 10.0:
pedimos los números impares de una lista del 1 al 20, primero ponemos la estructura, después hacemos la lista con los números del 1 al 20 y a esto lo llamamos x, después le damos la condición decimos que sean los números impares ósea que el resto de dividir cada elemento de la lista x entre 2 tiene que ser 1 y lo que vamos a mostrar es x:

```
Prelude> let lista = [ x | x <- [1..20] , x `mod` 2 == 1 ]
Prelude> lista
[1,3,5,7,9,11,13,15,17,19]
```

Fig. 10.0 | Lista intencional.

La estructura de una lista intencional:

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

donde x es lo que queremos seleccionar siempre y cuando cumpla con la condición.

Ahora intentemos pero con una lista del 1 al 20 y queremos los números pares multiplicados por 10, Fig. 10.1:

```
Prelude> let lista = [ x * 10 | x <- [1..20] , x `mod` 2 == 0 ]
Prelude> lista
[20,40,60,80,100,120,140,160,180,200]
```

Fig. 10.1 | Lista intencional.

Además a estas listas intencionales podemos incluir en alguna función y poner especificaciones un poco más complejas, vamos a hacer un ejemplo:
creamos un archivo y dentro de este creamos una función que se llamara cuentaCifras que recibirá una lista y será igual a cada elemento de la lista que recibimos x, filtramos los pares y después decimos si es menor que diez entonces esto devolverá una cifra si es más entonces dos cifras, Fig. 10.2:

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

```
*Main> cuentaCifras [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"]
```

Fig. 10.2 | Lista intencional.

Nota: Función odd devuelve si un número es impar = True sino = False.

Listas intencionales Dobles

En este apartado veremos como como combinar diferentes listas intencionales, anteriormente mencionamos la estructura de una lista intencional, ahora combinaremos varias, primero creamos la lista con la misma estructura mencionada anteriormente.
Tenemos una lista del 1 al 20 con **x** y otra del 1 al 100 con **y**, ahora ponemos las condiciones primero tomamos la lista **x** siempre que **x** sea menor que 10 (entonces tomaremos del 1 al 9), por otro lado tomamos **y** los números que sean divisibles por 10 (múltiplos de 10). Después sumamos **x** + **y,** Fig. 11.0:

```
lista = [ x + y | x <- [1..20], y <- [1..100] , x < 10, y `mod` 10 == 0 ]
```

```
*Main> 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
,93,103,14,24,34,44,54,64,74,84,94,104,15,25,35,45,55,65,75,85,95,105,16,26,36,46,56,6
6,76,86,96,106,17,27,37,47,57,67,77,87,97,107,18,28,38,48,58,68,78,88,98,108,19,29,39,
49,59,69,79,89,99,109]
```

Fig. 11.0 | Lista intencional doble.

- Desarrollar el siguiente ejercicio:

  Realizar una función que calcule la longitud de la lista.

  Creamos una función llamada longitud que recibirá como parámetro una lista, a esa lista le llamaremos x, entonces por cada elemento mostraremos un 1. Por lo tanto saldrán puros 1 pero si fuera de la lista ponemos la función sum esta sumara todos los 1 y nos dirá cuantos elemento tiene la lista es decir mostrara la longitud, Fig. 11.1:

```
longitud lista =  sum [1|  x <- lista]
```

```
*Main> let lista = [1..30]
*Main> lista
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]
*Main> longitud lista
30
```

Fig. 11.1 | Calculo de longitud de lista.

- Desarrollemos toro ejercicio:

  realizar un función que muestre las vocales: creamos la función mostrarVocales recibe una frase tomamos la frase y a cada elemento lo llamaremos letra, en la condición pondremos si la letra pertenece a la lista de vocales ['a','e','i','o','u'] y solo mostraremos la letra, Fig. 11.2:

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

```
*Main> mostrarVocales "Hola Mundo"
"oauo"
*Main>
```

Fig. 11.2 | Listas intencionales dobles, ejemplo.

```
*Main> mostrarC "hola mi casa es naranja casa y me gusta mi casa"
"ccc"
```

Fig. 11.3 | Listas intencionales dobles, ejemplo.

```
*Main> sumarC "la casa es cara"
2
```

Fig. 11.4 | Listas intencionales dobles, ejemplo.

Tuplas VS Listas

Una tupla es la colección de datos, conjunto de datos que vamos a tener y esta se diferencia de una lista, porque una tupla permite mezclar tipos de datos, por ejemplo:

si se tiene una lista y se mezclan números con strings mandara un error, Fig. 12.0:

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

<interactive>:298:14: error:
    • No instance for (Num [Char]) arising from the literal ‘1’
    • In the expression: 1
      In the expression: [1, 2, "hola"]
      In an equation for ‘lista’: lista = [1, 2, "hola"]
```

Fig. 12.0 | Lista.

Pero si esto lo intentamos hacer en una dupla no habría error, recuerden que las tuplas son con ( ), Fig. 12.1:

```
Prelude> let dupla = (1,2,"hola")
Prelude> dupla
(1,2,"hola")
Prelude> let dupla = (1,2,3)
Prelude> dupla
(1,2,3)
```

Fig. 12.2 |dupla.

Diferencia de lista de listas a listas de tuplas, la lista interna de las listas de listas puede tener distinto rango:

```
*Main> superLista
[[1,2],[3,4,5]]
```

Fig. 12.3 | Lista de lista diferente longitud.

En cambio en la lista de tupas nos muestra error porque todas las tuplas de la lista deben tener el mismo rango. Otro ejemplo es que en las tuplas puede haber mezclas pero siempre teniendo el mismo patrón, Fig. 12.4:

```
*Main> listaDuplas
[(1,"uno"),(2,"dos"),(3,"tres")]
```

Fig. 12.4 | Lista de tupla mismo patrón.

Si tuviéramos de tres elementos estas serían triplas, tuplas de 4 elementos serian cuadruplas, etc…

Funciones Duplas

Las funciones exclusivas para duplas sirven para devolvernos el elemento que este en según qué posición queremos encontrar, por ejemplo recordando un ejemplo anterior:

let lista = [“unos”, “dos”]

lista !!0

“uno”

Si tenemos una dupla y queremos acceder a esos elementos no es como anteriormente accedíamos a las listas, ponemos fst (first) después ponemos la función externa nos devolvería “uno” porque estamos pidiendo la primera posición, véase en el ejemplo dela Fig. 13.0:

```
*Main> fst dupla
"uno"
```

Fig. 13.0 | Fst.

Segunda posición de la dupla, snd (secondm Fig. 13.1):

```
*Main> snd dupla
1
```

Fig. 13.1 | Snd.

Listas de duplas

¿Cómo combinas diferentes listas convirtiendo listas de duplas? Esto es posible con una función llamada **zip**, véase en el ejemplo, Fig. 14.0:

tenemos dos listas una de cuatro nombres y otra con 4 datos de estaturas, y las combinamos de manera que se mezclen y coincida el primer elemento de la lista nombres con el primer elemento de la lista estaturas:

nombres= ["juan","alberto","manolo", "luis"]

estaturas= [1.70,1.90,1.80,1.60]

```
*Main> zip nombres estaturas
[("juan",1.7),("alberto",1.9),("manolo",1.8),("luis",1.6)]
```

Fig. 14.0 | Zip.

Como podemos ver funciono correctamente se mezclaron los nombres con las estaturas coincidiendo cada una de ellas. En caso de que una de las listas a mezclar sea más larga que otra Haskell recorta la más larga a la misma longitud de la más corta.

Ahora si queremos enumerar los nombres la función zip también puede servirnos, combinamos una lista infinita del q al infinito con la lista de nombres como se puede ver la lista infinita es más larga esta es cortada para quedar igual que la lista de nombres:

```
*Main> zip [1..] nombres
[(1,"juan"),(2,"alberto"),(3,"manolo"),(4,"luis")]
```

Fig. 14.1 | Zip.

Comando :t

Este ocmando tiene que ver con el tipo de dato, este no solo se utiliza para ver qué tipo es un valor. El comando **:t** y después de este valor se pone el valor del que se quiere saber el tipo.
se observa que al pedir el tipo de “a” nos dice que es de tipo char que es un carácter.

```
*Main> :t "a"
"a" :: [Char]
```

Fig. 15.0 | Comando :t.

Nos dice que True es de tipo Boleano.

```
*Main> :t True
True :: Bool
```

Fig. 15.1 | Comando :t.

Estos ejemplos fueron muy sencillos pero también podemos saber el formato utilizando este comando por ejemplo hemos visto que para la listas si queremos saber cuál es el primer elemento se utiliza la función **head**: let lista = [1,2,3,4,5, pedimos head lista y nos devuelve la posición 0 el número 1.

Apliquemos el comando :t a la función head para saber de qué formato es:

```
*Main> :t head
head :: [a] -> a
```

Fig. 15.2 | Formato de head.

Conversores Show y Read

Para transformar cualquier cosa a cadena de texto se utiliza la función show, de esta manera ponemos show 3 y nos devuelve una cadena de texto veamos el ejemplo de la Fig. 16.0:

```
*Main> show 3
"3"
*Main> show [1,2]
"[1,2]"
*Main> show ('a',1)
"('a',1)"
```

Fig. 16.0 | Función Show.

Read nos permite convertir una mayor variedad de tipos mientras que show solo lo convierte a cadena de texto, con read podemos hacer lo que queramos, por ejemplo:
hagamos una suma de una cadena de texto le sumamos un numero, y read funciona de la siguiente manera:
el primer parámetro que le ponemos lo toma como si fuera del mismo tipo que el segundo parámetro sin importar el operador que le demos.

```
*Main> read "3.4" + 6.4
9.8
*Main> read "False" && True
False
```

Fig. 16.1 | Función Read.

CONCLUSIÓN

A lo lardo de esta investigación de este manual se llegó a la conclusión de que el manual para aprender Haskell resulta algo interesante para el crecimiento de un programador.

Haskell es un lenguaje de programación funcional. Eso quiere decir varias cosas. Este se sitúa en una liga completamente diferente a la de los lenguajes de programación con más popularidad (C, Java, C, etc...).

Aprendí Haskell en la materia de Programación Lógica y Funcional además de utilizar unos videos de tutoriales en internet, honestamente Haskell me sorprendió, me fue sencillo utilizarlo. El lenguaje tiene bastantes características interesantes, no muy conocidas.

Espero haber ayudado a alguien a interesarse en este lenguaje de programación funcional y aclarar sus dudas.

REFERENCIAS

https://www.haskell.org/

https://www.youtube.com/watch?v=YAKTlImnS-g&list=PLraIUviMMM3fbHLdBJDmBwcNBZd_1Y_hC