Instituto Tecnológico de Mexicali.

“Tutoriales de Haskell”.

Rafael Sanchez Angulo.

12490466

Programación lógica y funcional.

Jorge Antonio Atempa Camacho.

6 de noviembre del 2016

Mexicali Baja California, México.

TABLA DE CONTENIDO.

TABLA DE CONTENIDO.

TABLA DE CONTENIDO.

VARIABLES.

- En Haskell es posible guardar valores de cualquier tipo mediante la asignación de un nombre.
- Esto es una ventaja porque así nos ayuda a tener mas claro el código y nos evita la redundancia de los datos, además de tener un mejor control de nuestra información.
- Una de las característica de las variables es que al momento de asignar el valor de la variable este permanecerá almacenado ahí durante todo el programa.

VARIABLES.

- Para declarar un variable en Haskell se hace de la siguiente manera:
- Se escribe el comando LET a continuación el nombre que se desea para la variable seguido del signo = y después el valor o los valores a asignar.
- Ejemplo:
- Crearemos la variable con el nombre X y su valor asignado será 1.
- Codificado en Haskell quedara de la siguiente manera.

```
"Prelude> let x = 1
Prelude> x
1
```

- De esta forma hemos creado nuestra primera variable con su valor asignado.

OPERADORES.

- En Haskell es útil usar operadores que nos ayuden a llevar acabo una función necesaria para nuestro programa.
- Un operador es un símbolo que indica que tipo de operación se realizara sobre los operandos.
- Estos al juntar son interpretados como funciones.
- En Haskell existen distintos tipos de operadores :

OPERADORES.

- **Aritméticos. (Suma, Resta, Multiplicación, División, Modulo).**
- Los operadores aritméticos se utilizan para realizar muchas de las operaciones aritméticas habituales que implican el cálculo de valores numéricos representados en variables y otras expresiones llamadas funciones.
- **Lógicos. (And, Or, Not).**
- Mientras que los operadores aritméticos se usan principalmente con números, los operadores lógicos están pensados para usarse con valores lógicos (verdadero y falso).
- **De comparación. (==, =, <, >, /=,>=, <=).**
- Un operador de comparación compara sus operandos y devuelve un valor lógico basado en si la comparación es verdad o no, los operando pueden ser números o variables.

OPERADORES.

- Veamos a continuación algunos ejemplos de Operadores en Haskell.

```
Prelude> let x = 5 + 7
Prelude> x
12
Prelude> let x = True && True
Prelude> x
True
Prelude> let x = 7
Prelude> x
7
```

FUNCIONES SUCC, MIN, MAX.

- Las funciones que a continuación veremos son funciones predeterminadas por Haskell y que tienen un objetivo en especifico.
- Para definir una función en Haskell se forma de la siguiente manera.
- Las funciones son llamadas escribiendo primero el nombre de la función, un espacio y sus parámetros.
- Es decir: **Función parámetro1**

FUNCION SUCC.

- La función SUCC toma un valor que tenga definido un sucesor y devuelve ese sucesor.
- **Ejemplo:**
- Codificado en Haskell que da la siguiente forma:

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

FUNCION MIN.

- La función min toma 2 valores de entrada, y determina cual de los 2 valores es el menor y lo muestra.
- **Ejemplo:**
- Codificado en Haskell que da la siguiente forma:

```
Prelude> min 52 7
7
Prelude> min 1 2
1
Prelude> min 11 10
10
```

FUNCION MAX.

- La función max toma 2 valores de entrada, y determina cual de los 2 valores es el mayor y lo muestra.
- **Ejemplo:**
- Codificado en Haskell que da la siguiente forma:

```
Prelude> max 11 10
11
Prelude> max 52 7
52
Prelude> max 1 2
2
```

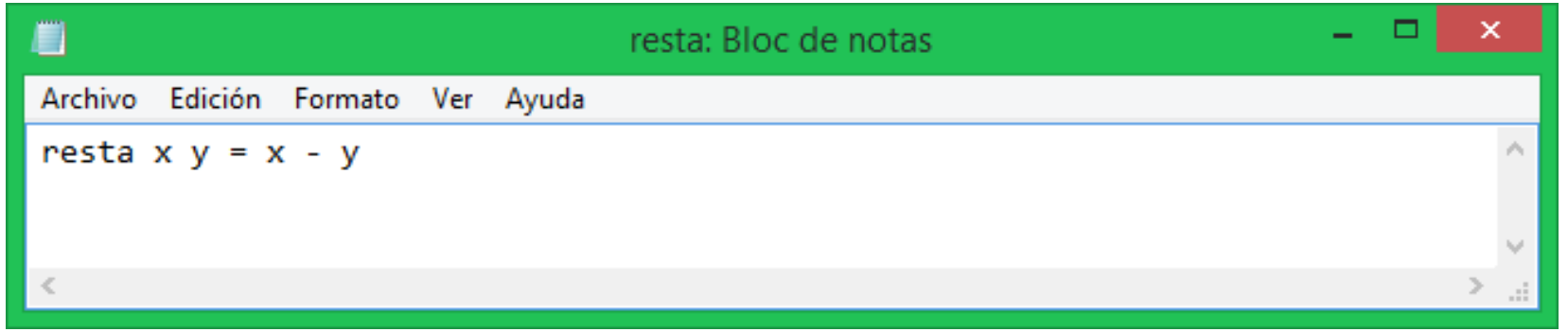
CREAR FUNCIONES.

- En Haskell tenemos la posibilidad de crear funciones en editores de texto externos siempre y cuando estos tengan la extensión .hs , en este archivo es donde guardaremos el comportamiento de dicha función.
- En Haskell las funciones son creadas de la siguiente forma: el nombre de la función, seguido de un espacio, después el parámetro de entrada, a continuación un signo de =, seguido de el comportamiento de la función.
- Es decir:
- **Función parámetroDeEntrada = comportamiento**

CREAR FUNCIONES.

- ✓ Si queremos una función de la resta de dos números tendremos que crear una función como la siguiente:
- ✓ **Resta x y = x – y**

  - ✓Resta es el nombre de la función.
  - ✓X Y son los parámetros de entrada.
  - ✓X - Y es el comportamiento.



CREAR FUNCIONES.

- Cuando ya tenemos creada nuestra función la mandamos a llamar en Haskell.
- Abrimos nuestro archivo de la siguiente manera:
- :load o :l y enseguida el nombre de nuestro archivo.
- En Haskell queda de la siguiente forma.

```
Prelude> :l resta.hs
[1 of 1] Compiling Main             ( resta.hs, interpreted )
Ok, modules loaded: Main.
*Main> resta 6 7
-1
*Main> resta 10 1
9
```

ESTRUCTURA IF.

- Como ya sabemos la sentencia IF es una condición que permite realizar una acción y si no realizar otra diferente.
- En Haskell usamos la sentencia if dentro de una función para asignarle una condición correspondiente al comportamiento de la misma.
- Es necesario respetar los parámetros de la sentencia if en Haskell, para que esta funcione de manera correcta.
- Su estructura es:
- **Función parámetrosDeEntrada = if evaluación**
  - **then accion1**
  - **else accion2**

ESTRUCTURA IF.

- ✓ A continuación veremos un ejemplo de como se vera en Haskell:
- ✓ Enviaremos 2 parámetros de entradas x y, con IF a evaluar si xy son divisibles enviara divisible de lo contrario enviara no son divisibles .

```
[1 of 1] Compiling Main                ( if1.hs, interpreted )
Ok, modules loaded: Main.
*Main> divisible 4 2
"Divisible"
*Main> divisible 7 2
"No divisible"
```

LISTAS.

- ✓ En Haskell una lista es un conjunto de datos de un mismo tipo, es muy usada e importante en el lenguaje Haskell.
- ✓ Algo que define a una lista es que los elementos que están dentro tienen que ser todos del mismo tipo de dato.
- ✓ La estructura de una lista es la siguiente:
- ✓ **[elemento1, elemento2, elemento3, elemento4… elemento *n*]**
- ✓ **Donde:**
- ✓ **Los [ ] indican que será una lista.**
- ✓ En Haskell se mira de la siguiente manera.

```
Prelude> [1,2,3,4]
[1,2,3,4]
Prelude> ["a","b","c"]
["a","b","c"]
Prelude> ['a','b','c']
"abc"
```

LISTAS.

- ✓ En Haskell también podemos declarar listas de una manera secuencial, es decir si queremos que nuestra lista valla de 1 hasta 10.
- ✓ Esto lo podemos lograr con 2 puntos seguidos (..)
- ✓ Visualizado en Haskell seria de la siguiente manera:

```
Prelude> [1,2..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> ['a','b'..'j']
"abcdefghij"
```

LISTAS.

- ✓ En Haskell también podemos crear una variable que contenga una lista todo esto por medio de la palabra reservada LET:
- ✓ Donde:
- ✓ LET nos indica que se creara una nueva variable donde se guardara una lista.
- ✓ Veamos como seria en Haskell.

```
Prelude> let x=[1,2..10]
Prelude> x
[1,2,3,4,5,6,7,8,9,10]
Prelude> let x=['a','b'..'j']
Prelude> x
"abcdefghij"
```

CONCATENACION DE LISTAS.

- ✓ La concatenación es la unión de 2 o mas elementos.
- ✓ En Haskell podemos concatenar listas mediante un operador predeterminado (++).
- ✓ La concatenación de las listas, solo es posible entre listas.

```
Prelude> ['a','b'..'j'] ++ ['l','m'..'z']
"abcdefghijlmnopqrstuvwxyz"
Prelude> ['a','b'..'j'] ++ ['l']
"abcdefghijl"
Prelude>  [1] ++ [2,3..11]
[1,2,3,4,5,6,7,8,9,10,11]
Prelude>  [1,2..10] ++ [11]
[1,2,3,4,5,6,7,8,9,10,11]
Prelude>  [1,2..10] ++ [11,12..15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

INDICE DE LISTAS.

- Un índice de listas nos devuelve el valor de un elemento que se encuentra en cierta posición de la lista.
- Ejemplo:
- Si tenemos una lista con estos valores [1,2,3].
- Y de esta lista queremos mostrar el elemento de la posición 2 que se encuentra en dentro de esta misma lista.
- Nos mostrara el 3 porque en Haskell se empieza a contar la lista desde la posición 0, en este caso la posición 0 es =1 la posición 1=2 y la posición 2 = 3 .

INDICE DE LISTAS.

- ✓ Para lograr que nos muestre la posición hay un comando muy fácil a utilizar en Haskell.
- ✓ Este comando se denota de la siguiente manera (!!).
- ✓ En Haskell quedaría así:

```
Prelude>  let x = [1,2..10]
Prelude> x
[1,2,3,4,5,6,7,8,9,10]
Prelude> x !! 9
10
Prelude>  let x=[[1,2..10] , [11,12..15]]
Prelude> x
[[1,2,3,4,5,6,7,8,9,10],[11,12,13,14,15]]
Prelude> x !! 0
[1,2,3,4,5,6,7,8,9,10]
```

FUNCIONES DE LISTAS.

- Las listas también cuentas con funciones especiales, como ya vimos las funciones nos permiten realizar un cierto comportamiento sobre un elemento.
- En las listas esto es muy importante si queremos manipular cada elemento de la lista, algunas de las funciones mas utilizadas en las listas son:

Length.	**Head.**	**Tail.**
Last.	**Init.**	**Reverse.**
Take.	**Drop.**	**Minimun**
Sum.	**Product.**	**Element.**

FUNCION LENGTH.

- Esta función nos devuelve la longitud o tamaño de la lista. Por ejemplo si tenemos la siguiente lista:
- **X = [1,5,6,8,2,5,7]**
- La longitud de nuestra lista deberá ser de 7.
- En Haskell Se vería de la siguiente manera:

```
Prelude> x =[1,2..7]
Prelude> x
[1,2,3,4,5,6,7]
Prelude> length x
7
```

FUNCION HEAD.

- La función head nos devuelve la cabeza de la lista.
- La listas esta conformada de la siguiente manera:
- **[1, 2 ,3 , 4 , 5]**
- Donde:
- La cabeza de la lista es el primer elemento que se encuentra dentro de la lista.
- Todos lo demás elementos son el cuerpo de la lista o la cola.
- En Haskell queda de la siguiente manera.

```
Prelude> x =[1,2..5]
Prelude> x
[1,2,3,4,5]
Prelude> head x
1
```

FUNCION TAIL.

- ✓ La función Tail nos sirve para devolver el cuerpo de la lista o la cola de la lista.
- ✓ En Haskell quedaría de la siguiente manera:

```
Prelude> x =[1,2..5]
Prelude> x
[1,2,3,4,5]
Prelude> tail x
[2,3,4,5]
```

FUNCION LAST.

- ✓ La función LAST, nos sirve para saber el ultimo elemento de la lista.
- ✓ En Haskell queda de la siguiente manera.

```
Prelude> x =[1,2..5]
Prelude> x
[1,2,3,4,5]
Prelude> last x
5
```

FUNCION INIT.

- ✓ La función INIT sirve para mostrar todos los elementos de la lista quitando el ultimo elemento , el cual seria el last.
- ✓ En Haskell queda de la siguiente manera:

```
Prelude> x =[1,2..5]
Prelude> x
[1,2,3,4,5]
Prelude> init x
[1,2,3,4]
```

FUNCION REVERSE.

- La función reverse nos sirve para devolver los elementos de la lista de forma contraria.
- Es decir si tenemos una lista:
- [1,2,3,4,5]
- Al ponerle la función reverse [1,2,3,4,5]
- Nos volteara los valores y será [5,4,3,2,1]
- En Haskell se mira de la siguiente forma:

```
Prelude> x =[1,2..5]
Prelude> x
[1,2,3,4,5]
Prelude> reverse x
[5,4,3,2,1]
```

FUNCION TAKE.

- ✓ La función take nos sirve para tomar 1 o varios elementos de la lista.
- ✓ TAKE toma desde el primer elemento hasta el numero de posiciones indicados:
- ✓ En Haskell queda de la siguiente forma:

```
Prelude> x =[1,2..5]
Prelude> x
[1,2,3,4,5]
Prelude> take 3 x
[1,2,3]
Prelude> x =[1,2..10]
Prelude> x
[1,2,3,4,5,6,7,8,9,10]
Prelude> take 8 x
[1,2,3,4,5,6,7,8]
Prelude> take 1 x
[1]
```

FUNCION DROP.

- ✓ La función DROP, sirve para quitar de la lista un numero de elementos.
- ✓ Un ejemplo en Haskell queda de la siguiente manera:

```
Prelude> x =[1,2..10]
Prelude> x
[1,2,3,4,5,6,7,8,9,10]
Prelude> drop 5x
[6,7,8,9,10]
Prelude> x
[1,2,3,4,5,6,7,8,9,10]
Prelude> drop 1x
[2,3,4,5,6,7,8,9,10]
Prelude> x
[1,2,3,4,5,6,7,8,9,10]
Prelude> drop 9 x
[10]
```

FUNCION MINIMUM.

- ✓ La función MINIMUM sirve para obtener de la lista el elemento con el valor menor.
- ✓ En una lista, **[1,2,3,4,5]** el valor mínimo será **1.**
- ✓ **En Haskell queda de la siguiente forma:**

```
Prelude> x =[1,2..10]
Prelude> x
[1,2,3,4,5,6,7,8,9,10]
Prelude> minimum x
1
```

FUNCION SUM.

- ✓ La función SUM, sirve para sumar todos los elementos de la lista.
- ✓ En Haskell queda de la siguiente manera:

```
Prelude> x =[1,2..10]
Prelude> x
[1,2,3,4,5,6,7,8,9,10]
Prelude> sum x
55
Prelude> x =[1,2..20]
Prelude> x
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> sum x
210
```

FUNCION PRODUCT.

- ✓ La función PRODUCT, nos sirve para multiplicar todos los elementos de la lista.

```
Prelude> x =[1,2..20]
Prelude> x
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> product x
2432902008176640000
Prelude> x =[1,2..10]
Prelude> product x
3628800
Prelude> x =[1,2..5]
Prelude> product x
120
```

FUNCION `ELEM`.

- ✓ La función `ELEM`, sirve para saber si un valor determinado es un elemento de la lista.
- ✓ Si es así nos devuelve un valor booleano verdadero
- ✓ En caso de no estar nos devuelve falso.
- ✓ Codificado en Haskell queda de la siguiente forma:

```
Prelude> x =[1,2..5]
Prelude> x
[1,2,3,4,5]
Prelude> elem 3 x
True
Prelude> elem 6 x
False
```

RANGOS.

- ✓ Una nota importante es que Haskell solo considera la relación de los primeros 2 parámetros del rango solo como suma o como resta.
- ✓ Por lo tanto no se puede multiplicar, dividir ni tampoco elevar a alguna potencia.
- ✓ Ejemplos:

```
Prelude> let x=[4,12..80]
Prelude> x
[4,12,20,28,36,44,52,60,68,76]
Prelude> let x=[50,45..10]
Prelude> x
[50,45,40,35,30,25,20,15,10]
```

FUNCIONES DE LISTAS INFINITAS.

- ✓ En Haskell las listas también pueden tomar formatos infinitos,
- ✓ es decir que estas nunca terminen.
- ✓ El tener listas infinitas es un problema por nunca para el ciclo de datos.
- ✓ En Haskell existen funciones que nos ayudan a manipular este tipo de listas.
- ✓ Como lo son la función Repeat y Cycle.

REPEAT.

- ✓ Esta función nos sirve para repetir infinitamente un elemento determinado.
- ✓ Para esta función se necesita:
- ✓ Poner la función repeat y especificar el elemento.
- ✓ En Haskell se mira de la siguiente manera:

```
Prelude> x =[1,2..5]
Prelude> x
[1,2,3,4,5]
Prelude> repeat x
```

```
,[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]
,[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]
,[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]
,[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]
,[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]
,[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]
,[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]
```

CYCLE.

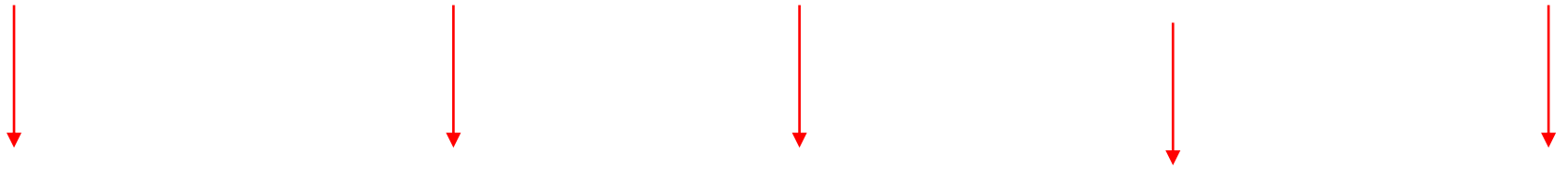
- ✓ La función Cycle nos sirve para repetir una serie de elementos infinitamente.
- ✓ En Haskell queda de la siguiente manera.

```
Prelude> x =[1,2..10]
Prelude> x
[1,2,3,4,5,6,7,8,9,10]
Prelude> cycle x
8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6
,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,
5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3
,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,
2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8Interrupted.
```

- ✓ A diferencia de repeat este repite elemento por elemento y no la lista completa.

LISTAS INTENCIONALES.

- ✓ Una lista intencional es una lista donde podemos filtrar los elementos que nosotros necesitemos por medio de condiciones.
- ✓ La estructura de una lista intencional cambia conforme a las listas que vimos anteriormente:
- ✓ Estructura de una lista intencional:
- ✓ LIntencional = [Lo que vamos a mostrar | x <- [Lista que vamos a filtrar] , Condiciones]



Nombre de la lista intencional.

Parámetros que vamos a mostrar.

Nombre de los elementos que vamos a filtrar.

Lista que vamos a filtrar .

Condiciones para filtrar los elementos que queremos de la lista.

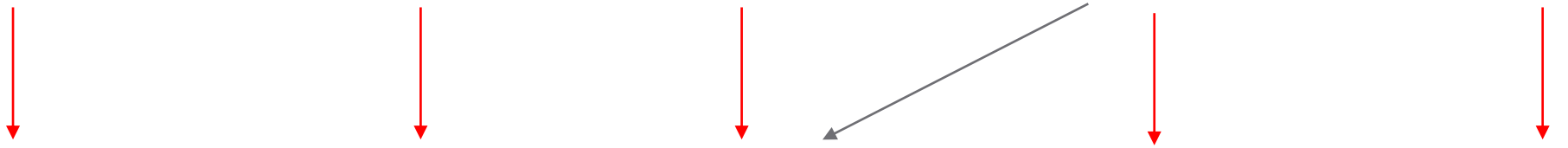
LISTAS INTENCIONALES.

- **Ejemplo:**
- Si solo queremos los elementos donde su doble sea mayor o igual a 10.
- Tendremos que crear una lista como la siguiente:
- **Lista = [x*2 | x <- [1,2..10] , x*2 >= 10]**
- Codificado en Haskell seria de la siguiente manera:

```
Prelude> let lista = [x*2 | x<- [1,2..10] , x*2 >= 10]
Prelude> lista
[10,12,14,16,18,20]
```

LISTAS INTENCIONALES DOBLES.

- ✓ Las listas intencionales dobles sirven para combinar 2 listas dentro de una lista intencional.
- ✓ Cuando filtramos elementos de varias listas, se producen todas las combinaciones posibles de dichas listas y se unen según la condición de salida propuesta.
- ✓ La Estructura de una lista intencional doble es similar a de la lista intencional sencilla:
- ✓ LIntencional = [Lo que vamos a mostrar | x <- [ListaFiltro1] y <-[ListaFiltro2] , Condiciones]



Nombre de la lista intencional.

Parámetros que vamos a mostrar.

Nombre de los elementos que vamos a filtrar.

Listas que vamos a filtrar .

Condiciones para filtrar los elementos que queremos de la lista.

LISTAS INTENCIONALES DOBLES.

- **Ejemplo:**
- Si tenemos 2 listas [2,8,10,12] [7,5,6,9] y queremos el producto de las combinaciones posibles de entre todos los elementos.
- Tendremos que crear una lista como la siguiente:
- **Lista = [x*y | x <- [2,8,10,12] , y <- [7,5,6,9]]**
- Codificado en Haskell seria de la siguiente manera:

```
Prelude> let lista = [x*y | x<- [2,8,10,12] , y<- [7,5,6,9]]
Prelude> lista
[14,10,12,18,56,40,48,72,70,50,60,90,84,60,72,108]
```

TUPLAS VS LISTAS.

- ✓ Las tuplas son utilizadas cuando sabes exactamente cuantos valores tienen que ser combinados y su tipo será de acuerdo al de estos elementos.
- ✓ A diferencia de las listas, las tuplas se denotan con paréntesis y al igual que en las listas sus valores se separan con comas.
- ✓ Otra diferencia clave entre las tuplas y las listas es que no tienen que ser homogéneas ya que las tuplas pueden contener una combinación de valores de distintos tipos.
- ✓ Tupla = (Elemento1 Int, Elemento2 char, Elemento 3 Int, Elemento n Char)
- ✓ Codificado en Haskell seria de la siguiente manera:

```
Prelude> let tupla = (1, 'a', 2)
Prelude> tupla
(1,'a',2)
```

TUPLAS VS LISTAS.

- De una tupla se pueden derivar varios nombres de acuerdo al numero de elementos que contenga.
- Una tupla con dos elementos es llamada dupla:
- Dupla = (1,'a')
- Una tupla de tres elementos es llamada tripla:
- Tripla = (1, 'a', 2)
- Una tupla con cuatro elementos es llamada cuatrupla.
- Cuadrupla = (1, 'a', 2, 'b')

```
Prelude> let dupla = (1,'a')
Prelude> dupla
(1,'a')
Prelude> let tripla = (1,'a',2)
Prelude> tripla
(1,'a',2)
Prelude> let cuatrupla = (1,'a',2, 'b')
Prelude> cuatrupla
(1,'a',2,'b')
```

FUNCIONES DE DUPLAS.

- ✓ Al igual que en las listas, en las duplas también hay funciones que nosotros podemos usar para sacar elementos de exactos que nosotros necesitemos de la tupla.
- ✓ Estos son:
- ✓ LA función **fst. (Esta función indica el primer elemento de la tupla).**
- ✓ La función **snd.(Esta función indica el segundo elemento de la tupla).**
- ✓ **Estas 2 funciones también se pueden usar sin tener que asignarle un nombre a la tupla, solo ponemos la función seguida de los elementos que contiene la tupla.**
- ✓ Veamos un ejemplo en Haskell:

```
Prelude> let dupla = (1,2)
Prelude> dupla
(1,2)
Prelude> fst dupla
1
Prelude> snd dupla
2
Prelude> fst (4.8 ,'a')
4.8
Prelude> snd (4.8, 'a')
'a'
```

FUNCION ZIP.

- En Haskell podemos entrelazar listas de una manera ordenada todo esto mediante una función llamada ZIP.
- Zip es una funciona que nos une 1 lista con la otra de forma que toma el primer valor de la primera lista y el primer valor de la segunda lista, así sucesivamente hasta unir todos los valores de las listas.

- Ejemplo:
- Si tenemos 2 listas **Alumnos=[“Jorge”, “Agustín”, ”Cristal”, ”Rafael”]**
- **Calificaciones=[9,10,10,9]**
- Al usar la función ZIP en Haskell queda de la siguiente forma:

```
Prelude> let alumnos= ["Jorge", "Agustin", "Cristal", "Rafael"]
Prelude> let calificaciones= [9,10,10,9]
Prelude> zip alumnos calificaciones
[("Jorge",9),("Agustin",10),("Cristal",10),("Rafael",9)]
```

COMANDO T: (TIPOS DE DATOS).

- ✓ En Haskell tenemos muchos tipos de datos y para saber el formato o valor de estos hay una función llamada :**t**
- ✓ **:t** nos indica el tipo de dato y el formato de la función.
- ✓ Por ejemplo:
- ✓ Si queremos saber el tipo de dato de la letra **a**, de los números **5, 5.8**
- ✓ Y de las funciones **head, zip,** codificadas en Haskell quedaría así:

```
Prelude> :t "a"
"a" :: [Char]
Prelude> :t 5
5 :: Num t => t
Prelude> :t 5.8
5.8 :: Fractional t => t
Prelude> :t head
head :: [a] -> a
Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

TIPOS DE CONVERSORES SHOW y READ.

- ✓ En Haskell tenemos diferentes tipos de comandos que nos sirven para realizar acciones necesarias para lo que sea que llevemos acabo en el momento.
- ✓ Hasta este punto hemos visto muchas funciones con las cuales podemos interactuar con Haskell.
- ✓ A continuación veremos por ultimo 2 funciones muy útiles que nos ayudaran al momento de estar programando en Haskell, estas son la función SHOW y la función READ.

FUNCION SHOW.

- ✓ La función show nos sirve para cambiar cualquier tipo de lista y convertirla en una cadena de caracteres.
- ✓ Dentro de esta función puede haber listas, listas dobles, tuplas, letras, números en fin todos los tipos de datos se pueden volver una cadena de caracteres con la función SHOW.
- ✓ Ejemplos.
- ✓ Si tomamos cualquier valor o cadena y usamos show en Haskell queda de la siguiente manera.

```
Prelude> show "a"
"\"a\""
Prelude> show 5
"5"
Prelude> show (5,"f")
"(5,\"f\")"
Prelude> show [1..5]
"[1,2,3,4,5]"
```

FUNCION READ.

- A diferencia de show con READ podemos convertir no solo a cadena de caracteres si no también a diferentes tipos de datos.
- READ nos permite leer cualquier dato con el tipo que nosotros queramos mientras que ese tipo sea el segundo operador.
- Por ejemplo en Haskell:

```
Prelude> read "5.8" +8.4
14.2
Prelude> read "True" && False
False
Prelude> read "[1,2,3]" ++ [4]
[1,2,3,4]
```

CONCLUSIONES.

- Haskell es un lenguaje dinámico y fácil de utilizar ya que usa parámetros y métricas sencillas de entender y que alguna vez aunque no sea en un lenguaje de programación, las hemos utilizado.
- Como se ha visto Haskell tiene muchas herramientas que nos facilitan la interacción con el lenguaje a la hora de programar, podemos hacer muchas cosas que en otros lenguajes no esta permitido lo cual tiene su lado bueno pero también malo, ya que podemos caer en las malas practicas al programar.
- Como en todo lenguaje de programación cuando empezamos a interactuar con el mismo se nos hace un poco difícil de entender pero con la practica toda va tomando un buen rumbo.

REFERENCIAS.

- José Javier Villena (2014). *Tutoriales Haskell*. Recuperado de https://www.youtube.com/watch?v=YAKTlImnS-g&list=PLraIUviMMM3fbHLdBJDmBwcNBZd_1Y_hC&index=1.
- *¡Aprende Haskell por el bien de todos!*. Recuperado de: http://aprendehaskell.es/main.html.