



TECNOLOGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE MEXICALI



PROGRAMACION LOGICA FUNCIONAL

MANUAL DE HASKELL

Alumna: Cristal Pelayo García.
Matricula: 12490889.
Maestro: Jorge Atempa Camacho.

Índice

- INTRODUCCION.
- ANTECEDENTES.
- INSTALACION.
- TIPO DE DATOS.
- VARIABLES.
- OPERADORES.
 - OPERADORES ARITMETICOS.
 - REGLA DE PRECEDENCIA.
 - OPERADORES LOGICOS.
 - OPERADORES DE COMPARACION.

Índice

- FUNCIONES SUCC, MIN, MAX.
 - FUNCION SUCC.
 - FUNCION MIN.
 - FUNCION MAX.
- CREAR FUNCIONES.
- ESTRUCTURA IF.
- LISTAS.
 - CONCATENACION DE LISTAS.
- INDICE DE LISTAS.

Índice

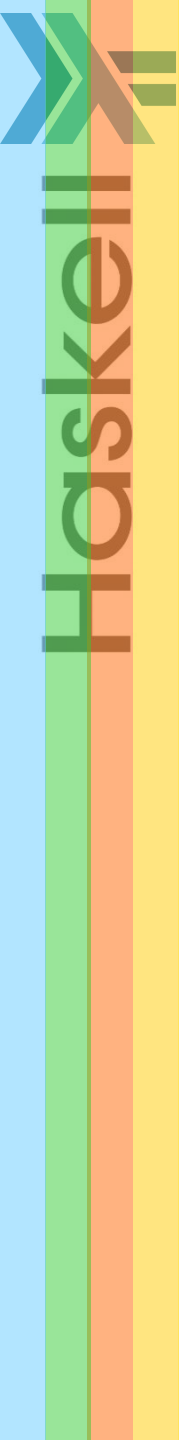
- FUNCIONES DE LISTAS.
 - FUNCION LENGTH.
 - FUNCION HEAD.
 - FUNCION TAIL.
 - FUNCION LAST.
 - FUNCION INIT.
 - FUNCION REVERSE.
 - FUNCION TAKE.
 - FUNCION DROP.
 - FUNCION MINIMUM.

Índice

- [FUNCION MAXIMUM.](#)
- [FUNCION SUM.](#)
- [FUNCION PRODUCT.](#)
- [FUNCION `ELEM`.](#)
- [RANGOS.](#)
- [FUNCIONES DE LISTAS INFINITAS.](#)
 - [REPEAT.](#)
 - [CYCLE.](#)
- [LISTAS INTENCIONALES.](#)
- [LISTAS INTENCIONALES DOBLES.](#)

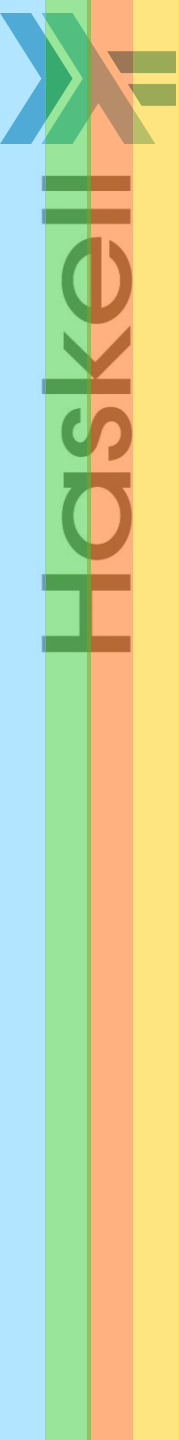
Índice

- TUPLAS VS LISTAS.
- FUNCIONES DE TUPLAS.
- LISTAS DE TUPLAS ZIP.
- COMANDO :t.
- CONVERSORES SHOW Y READ.
 - CONVERSIONOR SHOW.
 - CONVERSIONOR READ.
- CONCLUSION.
- REFERENCIAS.



INTRODUCCION

- Haskell es un lenguaje de programación. Específicamente, Los lenguajes de programación funcionales trabajan de forma diferente. En lugar de realizar acciones en secuencia, evalúan expresiones.
- 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.



ANTECEDENTES HISTORICOS

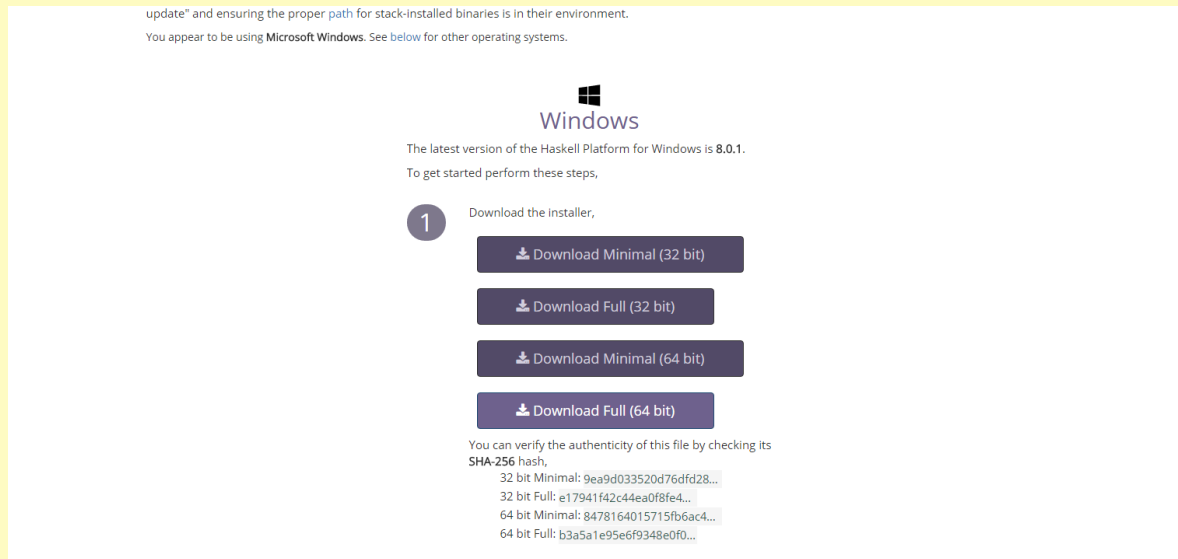
- 1930s: Alonzo Church desarrolla el lambda cálculo (teoría básica de los lenguajes funcionales).
- 1950s: John McCarthy desarrolla el Lisp (lenguaje funcional con asignaciones).
- 1960s: Peter Landin desarrolla ISWIN (lenguaje funcional puro).
- 1970s: John Backus desarrolla FP (lenguaje funcional con orden superior).

ANTECEDENTES HISTORICOS

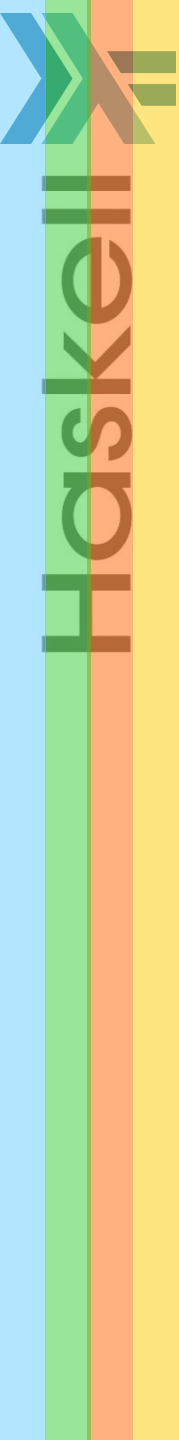
- 1970s: Robin Milner desarrolla ML (lenguaje funcional con tipos polimórficos e inferencia de tipos).
- 1980s: David Turner desarrolla Miranda (lenguaje funcional perezoso).
- 1987: Un comité comienza el desarrollo de Haskell.
- 2003: El comité publica el “Haskell Report”.

INSTALACION

- Para la instalación de Haskell, acceder a la pagina para la descarga.
<https://www.haskell.org/platform/>

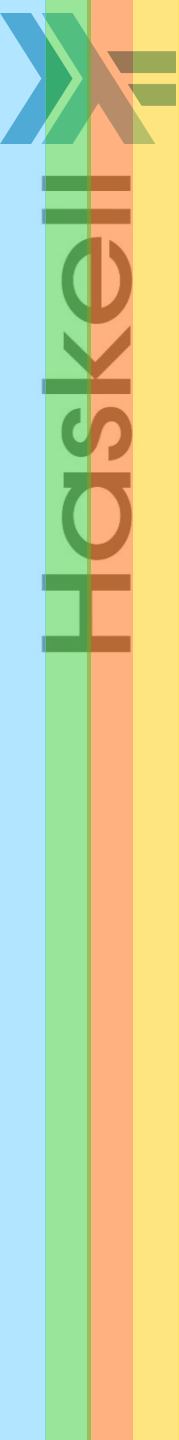


- A continuación comenzara la descarga.



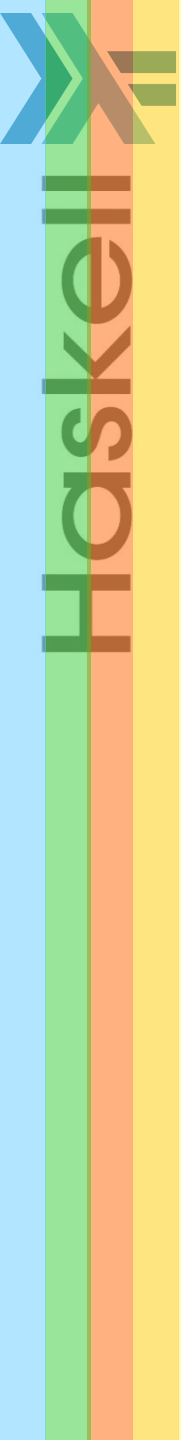
TIPOS DE DATOS.

- Int: representa e tipo de datos enteros, ósea un 7 puede ser un entero, pero un 7.3 no.
- Float: es un numero real coma flotante o decimal, de simple precisión.
- Double: es un numero real decimal, de doble precisión.



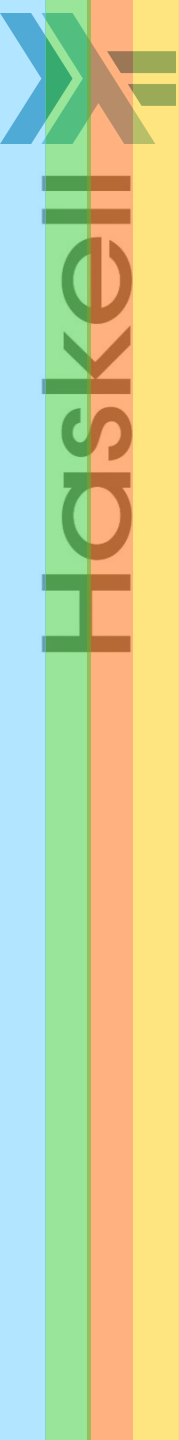
TIPOS DE DATOS.

- Bool: es un tipo de dato lógico en el cual sus únicos resultados pueden ser VERDADERO y FALSO.
- Char: representa un tipo de dato carácter representado por comillas simples, una lista de caracteres es una cadena.
- Listas: una colección de datos del mismo tipo de dato.
- Tupla: una colección de datos de diferente tipo de dato.



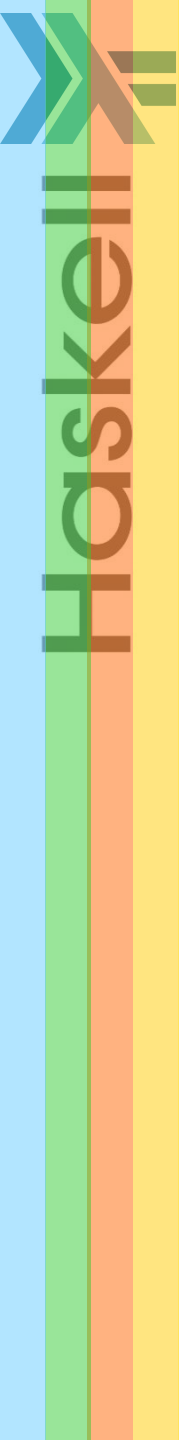
VARIABLES

- Haskell puede ser utilizado como una calculadora, pero esto puede ser útil en cálculos cortos, para cálculos mas largo en ocasiones se requieren de resultados intermedios.
- Para ello, en Haskell es posible almacenar valores mediante la asignación de nombres. Estos nombres son llamados variables.
- Cuando nuestro programa es ejecutado la variable se sustituye por el valor asignado previamente.



VARIABLES

- La ventaja de la utilización de variables, es que nos ayuda a la repetición innecesario de la información y de la memorización de cada dato a utilizar.
- Además de hace mas claro el código, esto no beneficiara en forma de que contara con mayor precisión, ya que al ser mas comprensible será menos probable que halla una confusión en el uso de los datos.
- Una característica muy importante de las variables es el hecho de que los valores de estas variables son inmutables. Es decir, que una vez asignado el valor este permanecerá durante todo el programa.

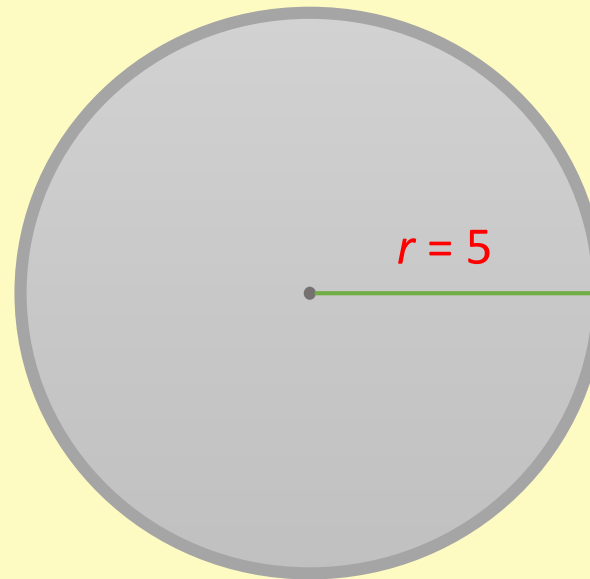


VARIABLES

- Un ejemplo, puede ser en el calculo de área de un circulo, en el cual su formula es $A = \pi * r^2$.
- El valor de r es equivalente al radio del circulo el cual puede variar según el valor a asignar, pero el valor de π siempre deberá de ser el mismo.
- Aquí es donde entran en juego las variables.

VARIABLES

- Podemos ver un ejemplo utilizando el área de un círculo, en el cual el valor de r será igual a 5, y π como ya es conocido es 3.141592653.
- Entonces:
 $r = 5.$
 $\pi = 3.141592653.$



$$A = 3.141592653 * 5^2$$

Imagen 1: radio de círculo.

VARIABLES

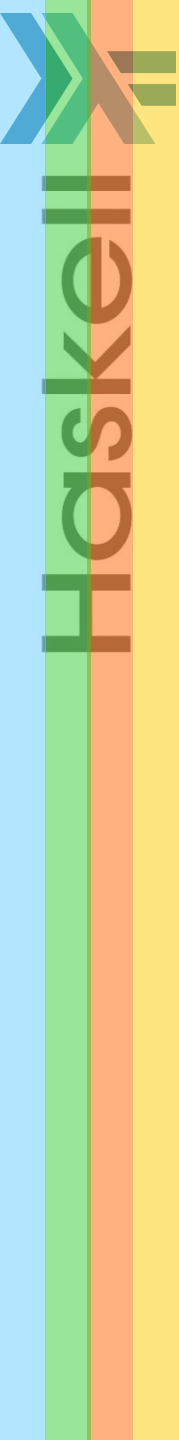
- Para declarar un variable se hace de la siguiente manera:
- Se escribe en nombre de la variable que se desea y a continuación el signo =, y después de este el valor a asignar.
- Entonces la forma será: **nombreVariable = valor.**

VARIABLES

- El ejemplo aplicado en Haskell se vería así.

```
Prelude> pi = 3.141592653  
Prelude> pi*5  
15.707963265
```

- Y listo! ya podemos utilizar un método para Haskell que nos hará mas fácil y comprensible el uso de valores, tanto en cálculos cortos y cálculos largos.



OPERADORES

- Un operador es un símbolo o serie de símbolos que indica que se realizará una operación específica, sobre unos ciertos parámetros llamados operandos.
- Los operadores junto con los operandos suelen interpretarse como funciones.
- Las funciones que realizaremos son llamadas funciones infijas ya que su operador va entre los dos parámetros de entrada.

OPERADORES

- Los operadores en Haskell son utilizados de la siguiente manera:
- parámetro, a continuación el operador, después el siguiente parámetro.
- Esto seria: **parámetro operador parámetro**.

OPERADORES

- En Haskell existen distintos tipos de operadores los cuales son:
 - Aritméticos.
 - Lógicos.
 - De comparación.

OPERADORES ARITMÉTICO

- Los operadores aritméticos toman los valores numéricos (literales o variables) como sus operando y devuelve un solo valor numérico. Los operadores aritméticos normales son:

OPERADOR	NOMBRE	DESCRIPCIÓN
+	Suma	Suma dos números.
-	Substracción	Resta dos números.
*	Multiplicación	Multiplica dos números.
/ ---- div	División	Divide dos números.
mod	Módulo	Devuelve el resto de dividir ambos números.

Tabla 1: operadores aritméticos.

OPERADORES ARITMÉTICO

- Ahora realizaremos ejemplos de operadores aritméticos en Haskell.
- $+$: suma dos números.

```
Prelude> 5+3  
8
```

- $-$: Resta dos números.

```
Prelude> 5-3  
2
```

OPERADORES ARITMÉTICO

- `/` : divide dos números.

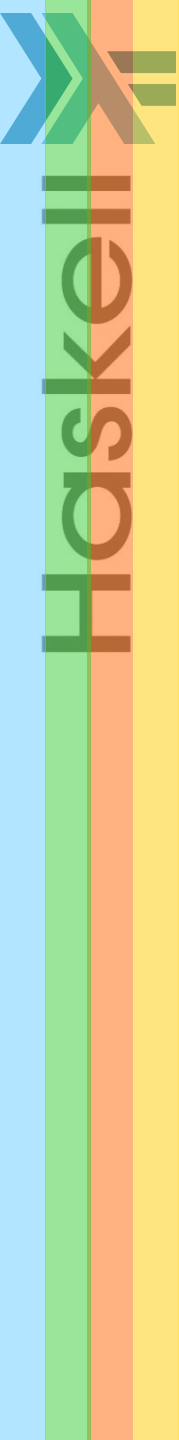
```
Prelude> 5/3  
1.6666666666666667
```

- ``div`` : divide dos números y su resultado se muestra en entero.

```
Prelude> 5 `div` 3  
1
```

- ``mod`` : devuelve el resto de la división de 2 numero.

```
Prelude> 5 `mod` 3  
2
```

REGLA DE PRECEDENCIA

- Los operadores aritméticos cuentan con algo llamado regla de precedencia, que se aplican cuando varios actúan juntos.
- En los operadores aritméticos, por ejemplo, la multiplicación y la división se ejecutan antes que la suma y la resta, es decir, tienen precedencia.
- Para alterar estas reglas de precedencia, se pueden usar paréntesis.

REGLA DE PRECEDENCIA

- Así por ejemplo $5 + 3 * 2$ es equivalente a $5 + (3 * 2)$ porque la precedencia de la multiplicación es superior a la de la suma. Sin embargo, $(5 + 3) * 2$ altera la precedencia.

ORDEN DE EVALUACIÓN	OPERADOR	DESCRIPCIÓN
de izquierda a derecha	<code>[]</code> , <code>()</code>	Llamadas a funciones y agrupamiento de expresiones
de izquierda a derecha	<code>*</code> , <code>/</code> , <code>%</code>	Multiplicación, división, división módulo
de izquierda a derecha	<code>+</code> , <code>-</code>	Suma y concatenación de cadenas, resta

REGLA DE PRECEDENCIA

- La regla de precedencia dice que primero se realizara la multiplicación y después la suma, por lo cual primero se multiplicara $3*2$ que es igual a 6 y después se le sumara 5 por lo cual el resultado es 11 .

```
Prelude> 5 + 3 * 2  
11
```

- Con el uso de paréntesis se altera la regla de precedencia, entonces primero se sumaran $5 + 3$ el cual es 8 y se multiplica por 2, el resultado será 16.

```
Prelude> (5 + 3) * 2  
16
```

A vertical Haskell logo on the left side of the slide, featuring the word "Haskell" in a stylized font with a blue and green arrow pointing right above it.

OPERADORES LOGICOS

- Mientras que los operadores aritméticos se usan principalmente con números, los operadores lógicos están pensados para usarse con valores lógicos (verdadero y falso).
- Hay solo tres operadores lógicos, Utilice `&&`, `||` con dos parámetros; utilice `NOT` con un parámetro.
- Los operadores lógicos pueden crear condiciones en una función, con lo que se deben cumplir dos o más condiciones para elegir un determinado método de cálculo.

OPERADORES LOGICOS

- Estos son los operadores lógicos que se utilizan en Haskell.

SÍMBOLO	DESCRIPCIÓN
&&	Verdadero sólo si los dos elementos son verdaderos
	Verdadero si cualquiera de las expresiones es verdadera
not	Cambia el valor de Falso a Verdadero y viceversa

Tabla 3: operadores logicos.

OPERADORES LOGICOS

- Ejemplo en Haskell.
- `&&`: representa el AND lógico, Verdadero sólo si los dos elementos son verdaderos.

```
Prelude> True && True
True
Prelude> True && False
False
Prelude> False && True
False
Prelude> False && False
False
```

OPERADORES LOGICOS

- Ejemplo en Haskell.
- `||`: representa el OR lógico, Verdadero si cualquiera de las expresiones es verdadera.

```
Prelude> True || True
True
Prelude> True || False
True
Prelude> False || True
True
Prelude> False || False
False
```

OPERADORES LOGICOS

- Ejemplo en Haskell.
- not: representa la negación lógica, Cambia el valor de Falso a Verdadero y viceversa

```
Prelude> not False
```

```
True
```

```
Prelude> not True
```

```
False
```

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

```
False
```

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

```
True
```


OPERADORES DE COMPARACION

- Un operador de la comparación compara sus operando y devuelve un valor lógico basado en si la comparación es verdad o no. Los operando pueden ser números o variables.

OPERADOR	NOMBRE	DESCRIPCIÓN
==	Igual a	Devuelve true si los operandos son iguales
/=	No igual a	Devuelve true si los operandos no son iguales
>	Mayor que	Devuelve true si el operador de la izquierda es mayor que el de la derecha.

Tabla 4: operadores de comparación.

OPERADORES DE COMPARACION

OPERADOR	NOMBRE	DESCRIPCIÓN
<code>>=</code>	Mayor o igual que	Devuelve true si el operador de la izquierda es mayor o igual que el de la derecha.
<code><</code>	Menor que	Devuelve true si el operador de la izquierda es menor que el de la derecha.
<code><=</code>	Menor o igual que	Devuelve true si el operador de la izquierda es menor o igual que el de la derecha.

Tabla 5: Operadores de comparación.

OPERADORES DE COMPARACION

- Ejemplo en Haskell.
- `==` : Igual a, Devuelve true si los operandos son iguales.

```
Prelude> 3 == 3
True
Prelude> 3 == 5
False
```

OPERADORES DE COMPARACION

- Ejemplo en Haskell.
- `/=` : No igual a, Devuelve true si los operandos no son iguales.

```
Prelude> 3 /= 5
```

```
True
```

```
Prelude> 3 /= 3
```

```
False
```

OPERADORES DE COMPARACION

- Ejemplo en Haskell.
- `>` : Mayor que, Devuelve true si el operador de la izquierda es mayor que el de la derecha.

```
Prelude> 3 > 3
```

```
False
```

```
Prelude> 3 > 5
```

```
False
```

```
Prelude> 10 > 5
```

```
True
```

OPERADORES DE COMPARACION

- Ejemplo en Haskell.
- `>=` : Mayor o igual que, Devuelve true si el operador de la izquierda es mayor o igual que el de la derecha.

```
Prelude> 3 >= 3
True
Prelude> 3 >= 5
False
Prelude> 10 >= 5
True
```

OPERADORES DE COMPARACION

- Ejemplo en Haskell.
- `<` :Menor que, Devuelve true si el operador de la izquierda es menor que el de la derecha.

```
Prelude> 3 < 3
```

```
False
```

```
Prelude> 3 < 5
```

```
True
```

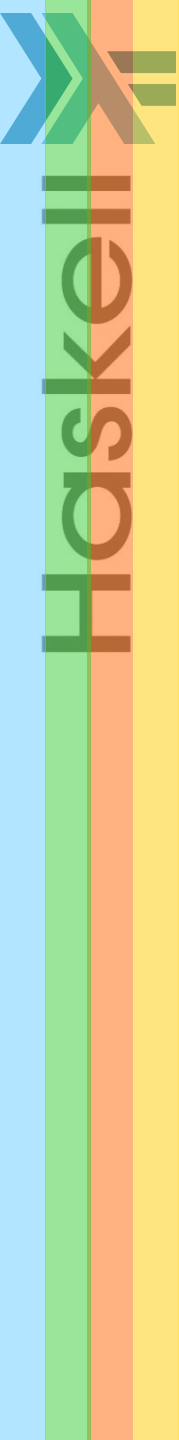
```
Prelude> 10 < 5
```

```
False
```

OPERADORES DE COMPARACION

- Ejemplo en Haskell.
- `<=` :Menor o igual que, Devuelve true si el operador de la izquierda es menor o igual que el de la derecha.

```
Prelude> 3 <= 3
True
Prelude> 3 <= 5
True
Prelude> 10 <= 5
False
```

FUNCIONES SUCC, MIN, MAX

- Las funciones se utilizan en notación prefija, es decir, el nombre de la función se coloca antes de sus parámetros.
- El valor que arroja el llamar esta función, esta dentro de una lista que ya tiene definida Haskell internamente.

FUNCIONES SUCC, MIN, MAX

- En Haskell, las funciones son llamadas escribiendo su nombre, un espacio y sus parámetros, separados por espacios.
- Es decir: **nombreFunción parámetro1 parametro2**

FUNCION SUCC

- La función succ toma cualquier valor que tenga definido un sucesor y devuelve ese sucesor.
- Ejemplo en Haskell.

```
Prelude> succ 8
```

```
9
```

```
Prelude> succ 203.11
```

```
204.11
```

```
Prelude> succ (-3)
```

```
-2
```

```
Prelude> succ 'a'
```

```
'b'
```

FUNCION MIN

- La función min toma 2 valores de entrada, el cual determina quien tiene el valor mínimo.
- Ejemplo en Haskell.

```
Prelude> min 9 6
```

```
6
```

```
Prelude> min 'z' 'a'
```

```
'a'
```

```
Prelude> min 203.88 307.77
```

```
203.88
```

```
Prelude> min (-8) (-5)
```

```
-8
```

FUNCION MAX

- La función max toma 2 valores de entrada, el cual determina quien tiene el valor máximo.
- Ejemplo en Haskell.

```
Prelude> max 9 6
```

```
9
```

```
Prelude> max 'z' 'a'
```

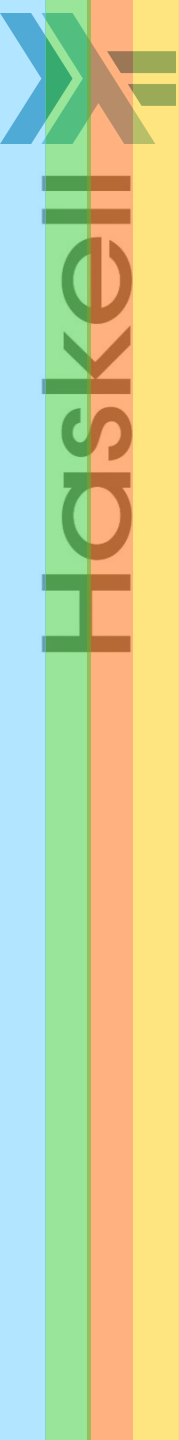
```
'z'
```

```
Prelude> max 203.88 307.77
```

```
307.77
```

```
Prelude> max (-8) (-5)
```

```
-5
```



CREAR FUNCIONES

- Para crear nuestras propias funciones es necesario un editor de textos, ya que ahí es donde guardaremos el comportamiento de esta función.
- Es muy importante que el archivo donde este nuestra función se guardado con una extensión tipo .hs, ya que con esta es la que trabaja Haskell.

CREAR FUNCIONES

- Las funciones son creadas de la siguiente manera: el nombre de la función, después el o los parámetros de entrada, signo de =, seguido de el comportamiento de esta función.
- Por lo tanto: **nombreFunción parámetrosEntrada = comportamiento**

CREAR FUNCIONES

- Por ejemplo, si queremos una función de suma de dos números tendremos que hacer algo como esto:

`suma x y = x+y`

- Donde:
 - suma es igual al nombre de la función.
 - x y son los parámetros de entrada.
 - x+y es el comportamiento.

CREAR FUNCIONES

- Entonces, el ejemplo en el editor seria algo como esto:

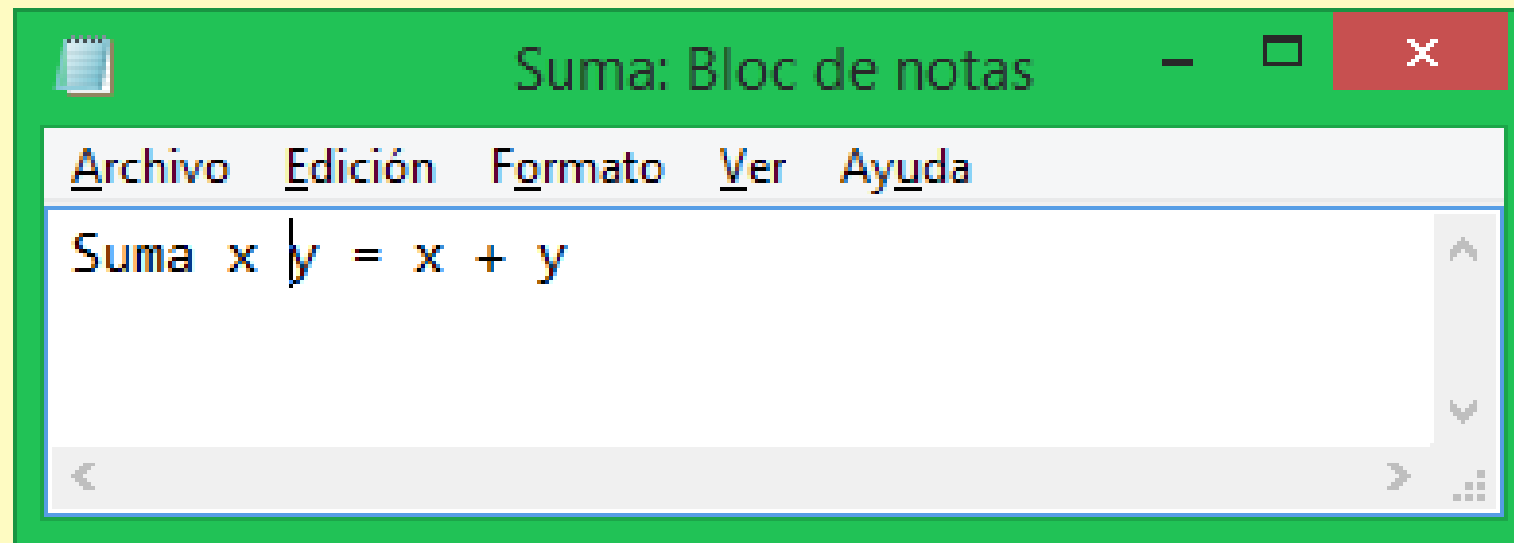


Imagen 2: Ejemplo de funciones en Haskell.

CREAR FUNCIONES

- Una vez realizada nuestra función en el editor de textos, seguimos en Haskell.
- Abrimos nuestro archivo de la siguiente manera, `:load` o `:l` y el nombre de nuestro archivo.

```
Prelude> :load "Suma.hs"  
[1 of 1] Compiling Main                ( Suma.hs, interpreted )  
Ok, modules loaded: Main.
```

CREAR FUNCIONES

- Ahora realizaremos una pruebas para ver si nuestra función se comporta como debería de hacerlo.

```
*Main> suma 5 6  
11
```

- Si queremos realizar algún cambio como este:

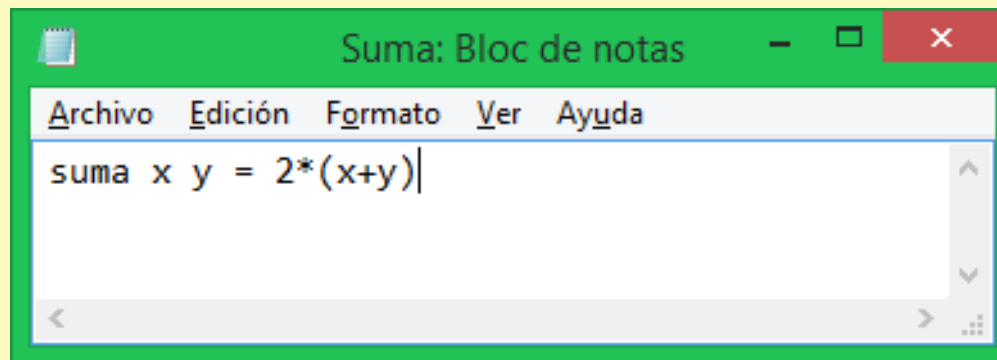


Imagen 3: Ejemplo de funciones en Haskell.

CREAR FUNCIONES

- En Haskell tendríamos que hacer un `:reload` o `:r`.

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

- Y a continuación realizamos nuestra función ya recargada.

```
*Main> suma 5 6  
22
```

ESTRUCTURA IF

- La sentencia IF se le conoce como estructura de selección simple y su función es realizar o no una determinada acción.
- Esta estructura se basándose en el resultado de la evaluación de una expresión (verdadero o falso), en caso de ser verdadero se ejecuta la sentencia y en caso de que se falso otra acción.

ESTRUCTURA IF

- Entonces, IF se comporta de la siguiente manera:

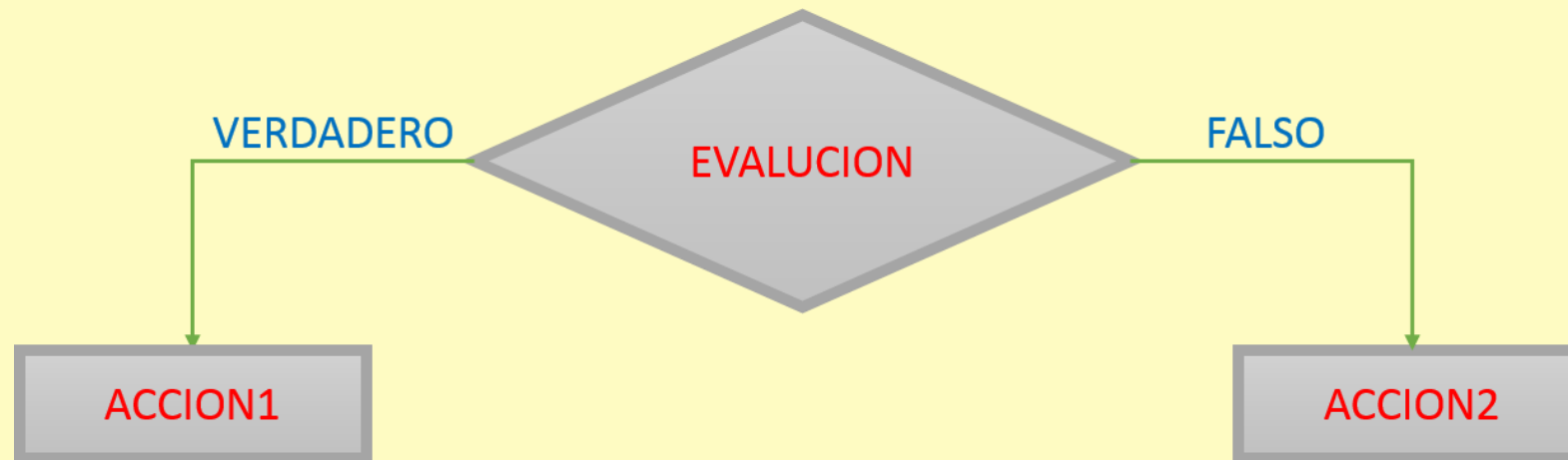


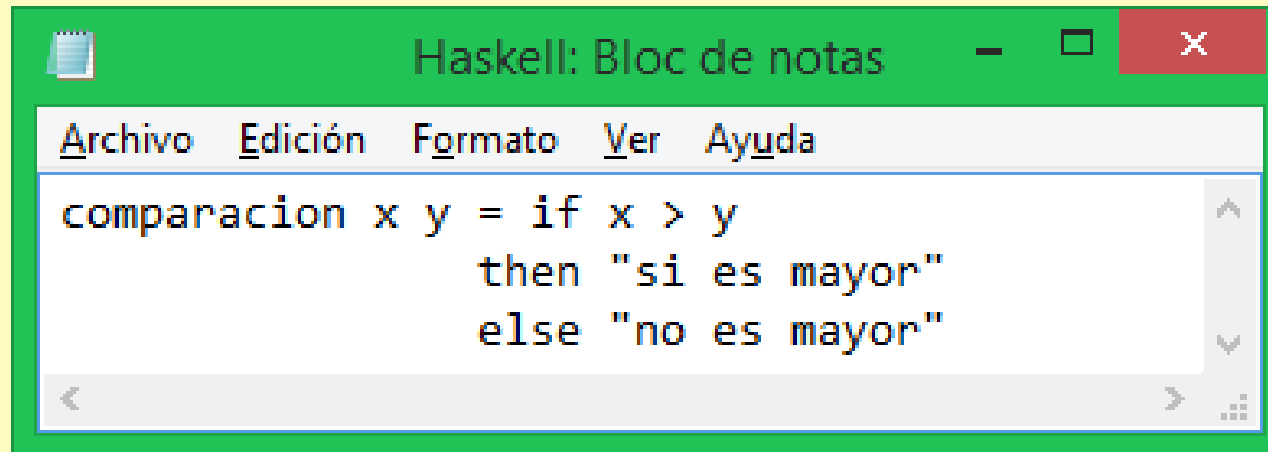
Imagen 4: diagrama estructura if.

ESTRUCTURA IF

- La manera de realizar un IF en Haskell seria: nombre de la función, seguido de los parámetros de entrada, signo =, la estructura IF su evaluación.
- Encaso de ser verdadero un THEN seguido de se acción, y encaso de ser falso un ELSE seguido de se acción.
- Esto seria : `nombreFunción parámetros = if evaluación
then accion1
else accion2`

ESTRUCTURA IF

- Ejemplo en Haskell:
- Enviaremos 2 parámetros de entradas x y, con IF evaluar si x es mayor a y, si esto es verdadero entonces enviara un mensaje “si es mayor”, y si el resultado es falso, enviara “no es mayor.”

A screenshot of a text editor window titled "Haskell: Bloc de notas". The window has a green title bar and a menu bar with options: Archivo, Edición, Formato, Ver, and Ayuda. The main text area contains the following Haskell code:

```
comparacion x y = if x > y  
                  then "si es mayor"  
                  else "no es mayor"
```

The code is displayed in a monospaced font with syntax highlighting: keywords like 'if', 'then', and 'else' are in blue, and string literals are in red. The window also features a scrollbar on the right side.

Imagen 5: Ejemplo de if en Haskell.

ESTRUCTURA IF

- Abrimos nuestro archivo .hs

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

- Probamos nuestro código.

```
*Main> comparacion 3 5  
"no es mayor"  
*Main> comparacion 20 7  
"si es mayor"
```

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.
- Las listas son muy utilizadas ya que son muy practicas al querer trabajar con un conjunto de datos que sean del mismo tipo.
- Por ejemplo, podemos crear listas de entero pero esta solo contara con elementos enteros no podemos mezclarlo con caracteres, ni una lista de caracteres con entero.

LISTAS

- La manera que se declara una lista es la siguientes:
`[elemento1,elemento2,elemento3,elemento4...elementon]`
- Cabe destacar que la dimensión de la lista puede ser la que nosotros queramos, es decir podemos hacer una lista con n elementos.
- Un ejemplo de una lista seria:
`[4,5,9,25,60]` <- todos los elementos de la lista son de mismo tipo en este caso enteros.

LISTAS

```
Prelude> [3, 'a', 4, 5]
```

```
<interactive>:11:2: error:
```

```
    No instance for (Num Char) arising from the literal '3'
```

```
    In the expression: 3
```

```
    In the expression: [3, 'a', 4, 5]
```

```
    In an equation for 'it': it = [3, 'a', 4, ....]
```

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

Lista de listas

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

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

Lista de enteros

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

```
Prelude> [True,False,True,False]
```

Lista de booleanos

```
[True,False,True,False]
```

```
Prelude> ['a','b','c','d']
```

Lista de caracteres

```
"abcd" ?
```

LISTAS

- Como vimos en el ejemplo anterior forzosamente nuestra lista debe de contar con elementos del mismo tipo, ya que si queremos hacer lo contrario Haskell nos mostrara un mensaje de error.
- Pero si miramos detenidamente sucede algo muy raro con las listas de tipo carácter.

```
Prelude> ['a','b','c','d']  
"abcd"
```

- Nos muestra la lista como un valor cadena, pero, ¿Porqué sucede?

LISTAS

- Respondiendo la pregunta anterior, es muy sencillo esto sucede porque Haskell cuenta con el tipo de dato cadena, en su defecto una lista de caracteres la reconoce como una cadena.
- Pero esto no quiere decir que sea un elemento cadena, por el contrario este elemento se comporta como n elementos y no uno solo, y lo podemos corroborar aplicando el ejemplo anterior:

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

LISTAS

- Haskell cuenta con otra manera de declarar listas pero esta solo es posible si la lista cuenta con una secuencia, es decir si queremos que nuestra lista valla de 1 hasta 10.
- Esto lo podemos lograr con los símbolos (..)
- El ejemplo en Haskell seria de la siguiente manera:

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

LISTAS

- Una manera de crear variables para las listas es de la siguiente manera:

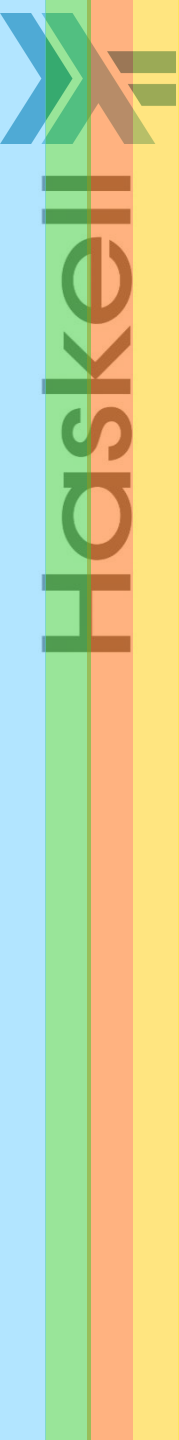
Let nombreVariable = [elemento1,elemento2...elementon]

- Donde:
 - let es la función que nos ayuda a guardar el parámetro siguiente como el nombre de la variable.
 - nombreVariable el nombre que deseamos para asignarle el valor lista.
 - [elemento1,elemento2,elementon] representa los n elementos de la lista.

LISTAS

- Ejemplo en Haskell:

```
Prelude> let x=[1,3,2]
Prelude> x
[1,3,2]
Prelude> let flotantes=[1.3,2.4,3.5] Prelude> flotantes
[1.3,2.4,3.5]
Prelude> let cadena=['h','o','l','a']
Prelude> cadena
"hola"
Prelude> let booleanos = [False,True]
Prelude> booleanos
[False,True]
```



CONCATENACION DE LISTAS

- La concatenación es la unión de los elementos de 2 o mas listas dentro de una sola lista.
- En Haskell esta es una tarea muy sencilla ya que tiene un operador predeterminado (++).
- La concatenación de las listas, solo es posible entre lista, no entre listas y otro tipo de datos como enteros o caracteres aunque los elementos de esta lista sean del mismo tipo de dato.

CONCATENACION DE LISTAS

- Ejemplo en Haskell:

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

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

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

```
<interactive>:17:1: error:
```

```
Non type-variable argument in the constraint: Num [a]
```

```
(Use FlexibleContexts to permit this)
```

```
When checking the inferred type
```

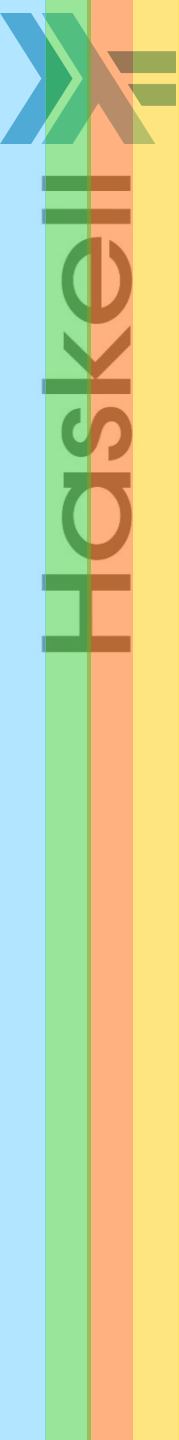
```
it :: forall a. (Num [a], Num a) => [a]
```

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

```
"hola mundo "
```

Mensaje de error al querer concatenar una lista con un elemento entero.

Otro ejemplo para corroborar que listas de caracteres actúan como cadenas

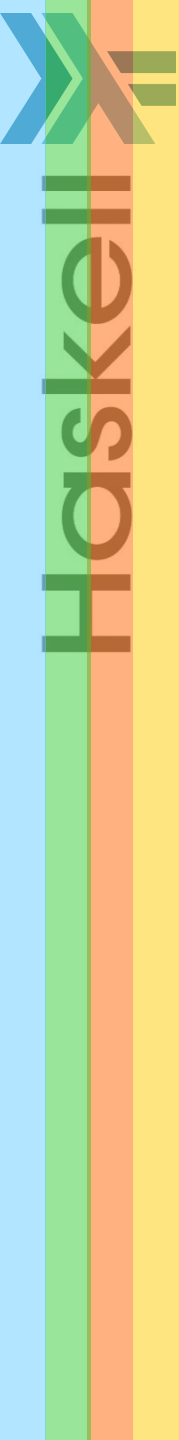


CONCATENACION DE LISTAS

- Pero, ¿Cómo hacemos para unir una lista con un solo elemento?, muy fácil podemos poner ese único elemento dentro de una cadena, de esta forma:

```
Prelude> [1,2]++[3]
[1,2,3]
Prelude> ['h','o','l']++['a']
"hola"
```

- Recordando siempre que los elementos de las listas deben de ser del mismo tipo de dato.



CONCATENACION DE LISTAS

- Pero otra opción que Haskell permite para resolver este problema sería usando el símbolo don puntos (:).
- Este carácter permite la unión de un elemento con una lista, siempre y cuando estos sean del mismo tipo de dato.
- Pero ese elemento solo puede unirse al inicio o a la cabeza de la lista.

CONCATENACION DE LISTAS

- A continuación se demuestra esto en Haskell.

```
Prelude> 'h':"ola"  
"hola"  
Prelude> 1:[2,3,4]  
[1,2,3,4]  
Prelude> 'm':['u','n','d','o']  
"mundo"  
Prelude> False:[True,False,False]  
[False,True,False,False]  
Prelude> 2.3:[1.0,2.3,5.6]  
[2.3,1.0,2.3,5.6]
```

CONCATENACION DE LISTAS

- Errores en la concatenación.

```
Prelude> 'a':[2,3,4]
```

```
<interactive>:10:6: error:
```

```
    No instance for (Num Char) arising from the literal '2'
```

```
    In the expression: 2
```

```
    In the second argument of '(:)', namely '[2, 3, 4]'
```

```
    In the expression: 'a' : [2, 3, 4]
```

Error al querer elementos de diferente tipo de dato.

```
Prelude> ['m','u','n','d']: 'o'
```

```
<interactive>:12:19: error:
```

```
    Couldn't match expected type '[[Char]]' with actual type 'Char'
```

```
    In the second argument of '(:)', namely 'o'
```

```
    In the expression: ['m', 'u', 'n', 'd'] : 'o'
```

```
    In an equation for 'it': it = ['m', 'u', 'n', ....] : 'o'
```

Error querer concatenar el elemento al final de la lista.

INDICE DE LISTAS

- Una manera de crear variables para las listas es de la siguiente manera:

Let nombreVariable = [elemento1,elemento2...elementon]

- Donde:
 - let es la función que nos ayuda a guardar el parámetro siguiente como el nombre de la variable.
 - nombreVariable el nombre que deseamos para asignarle el valor lista.
 - [elemento1,elemento2,elementon] representa los n elementos de la lista.

INDICE DE LISTAS

- Un índice de listas nos devuelve el valor de un elemento que se encuentra en cierta posición de la lista.
- Por ejemplo, si tenemos una lista con estos valores [6,3,1]. Y de esta lista queremos mostrar el elemento de la posición 2 que se encuentra en dentro de esta misma lista.

```
let x = [ 6 , 3 , 1 ]
```

- Entonces, el valor que queremos mostrar es 3.

INDICE DE LISTAS

- ¿Cómo haremos esto?, muy fácil Haskell cuenta con una función que nos permite realizar esta sencilla tarea, y se denota de la siguiente manera (!!).
- Si nos basamos en el ejemplo anterior:

```
let x = [ 6 , 3 , 1 ]
```

INDICE DE LISTAS

- Y queremos mostrar el elemento en la posición 2 sería así:


`x !! 1`

- Donde :
 - x es igual a la lista
 - !! Es la función índice de lista.
 - 1 es la posición.

INDICE DE LISTAS

- Cabe mencionar que la posición de la lista comienza en cero, por eso a pesar de que el elemento en el segundo lugar de la lista, el valor de la posición dentro de la lista es 1.

```
let x = [ 6, 3 , 1 ]
```



Posición	0	1	2
	6	3	1

INDICE DE LISTAS

- Ejemplo en Haskell:

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

```
Prelude> lista !!0
```

```
[1,2]
```

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

```
2
```

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

```
Prelude> enteros!!0
```

```
1
```

```
Prelude> let caracteres=['h','o','l','a']
```

```
Prelude> caracteres!!3
```

```
'a'
```

```
Prelude> let flotantes=[1.3,2.4,3.5]
```

```
Prelude> flotantes!!2
```

```
3.5
```

INDICE DE LISTAS

- Una lista se considera un tipo de dato, ya que una lista puede contener elementos tipo listas.
- Y estas se manejan de la misma manera que cualquier otra lista que cuenta con elementos de otro tipo de dato.

```
Prelude> [[1,2,3],[4,5]]
[[1,2,3],[4,5]]
Prelude> [['a'..'d'],['e','f','g']]
["abcd","efg"]
```

INDICE DE LISTAS

- Pero al querer manipular los índices de listas, puede haber ciertas variaciones.
- Por ejemplo, si al ejecutar la función de índice de lista en la posición cero, esta mostrar la primera lista completa:

```
Prelude> let x = [['a'..'d'],['e','f','g']]
Prelude> x !! 0
"abcd"
```

INDICE DE LISTAS

- Pero que pasa si queremos ver el primer elemento de la primera lista, tendríamos que usar doble función índice de listas de esta manera:

```
Prelude> let x = [['a'..'d'],['e','f','g']]
```

```
Prelude> x !! 0 !! 0  
'a'
```

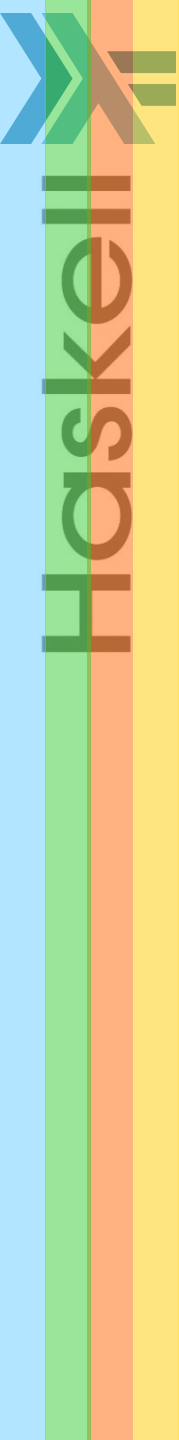
Primera lista,
primer elemento.

```
Prelude> x !! 0 !! 1  
'b'
```

Primera lista,
segundo elemento.

```
Prelude> x !! 1 !! 2  
'g'
```

Segunda lista,
tercer elemento.



FUNCIONES DE LISTAS

- Las listas también cuentan 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, para después hacer que estos elementos se comporten de cierta manera.
- A continuación repasaremos algunas de las funciones mas importantes.

FUNCION LENGTH

- Esta función nos devuelve la longitud o tamaño de la lista. Por ejemplo si tenemos la siguiente lista:

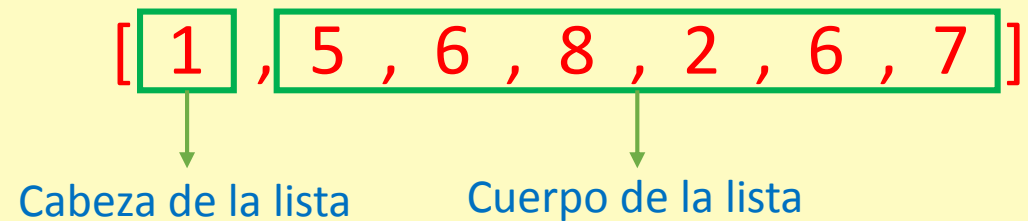
[1,5,6,8,2,5,7]

- La longitud de nuestra lista deberá ser de 7.

```
Prelude> length [1,5,6,8,2,5,7]  
7
```

FUNCION HEAD

- Todas las lista se divide de la siguiente manera:



- Donde:
- La cabeza de la lista es el primer elemento que se encuentra dentro de la lista.
- El cuerpo de la lista con todos los elementos a excepción de la cabeza.

FUNCION HEAD

- La función HEAD no ayuda si queremos devolver el primer elemento, o en su defecto la cabeza de la lista
- Por ejemplo si tenemos:

[1,2,3,4,5,6]

- Entonces 1 deberá ser la cabeza de la lista:

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

FUNCION TAIL

- Por el contrario, la función TAIL nos devolverá el cuerpo de la lista o todos los elementos de la lista a excepción de primero.
- Aplicando el ejemplo anterior esto seria así:

```
Prelude> tail [1,2,3,4,5,6]
[2,3,4,5,6]
Prelude> tail ['h','o','l','a']
"ola"
Prelude> tail [False,True]
[True]
Prelude> tail [2.0,3.0,4.0,5.0,6.0,7.0,8.0]
[3.0,4.0,5.0,6.0,7.0,8.0]
```

FUNCION LAST

- La función LAST, nos devuelve lo contrario de la función HEAD, es decir el ultimo elemento de la lista

[1 , 5 , 6 , 8 , 2 , 6 , 7]

Init de la lista

Last elemento de la lista

- Donde:
- El init de la lista son todos los elementos de nuestra lista a excepción de ultimo.
- Last de la lista, es el ultimo elemento de la lista.

FUNCION LAST

- Ejemplo de la función LAST en Haskell:

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

```
6
```

```
Prelude> last ['h','o','l','a']
```

```
'a'
```

```
Prelude> last [False,True]
```

```
True
```

```
Prelude> last [2.0,3.0,4.0,5.0,6.0,7.0,8.0]
```

```
8.0
```

```
Prelude> last [[1,2,3],[4..7],[8,9,0]]
```

```
[8,9,0]
```

FUNCION INIT

- La función INIT, trabaja al revés que la función TAIL, muestra todos los elementos de la lista a excepción del ultimo numero.
- Aplicando el ejemplo anterior:

```
Prelude> init [1,2,3,4,5,6]
[1,2,3,4,5]
Prelude> init ['h','o','l','a']
"hol"
Prelude> init [False,True]
[False]
Prelude> init [2.0,3.0,4.0,5.0,6.0,7.0,8.0]
[2.0,3.0,4.0,5.0,6.0,7.0]
Prelude> init [[1,2,3],[4..7],[8,9,0]]
[[1,2,3],[4,5,6,7]]
```


FUNCION REVERSE

- Otra función de Haskell es reverse, esta función nos devuelve nuestra lista de forma reversa.
- Si tenemos la lista: `[1,5,6,8,2,6,7]`.
- Aplicando la función reverse, la lista se mostraría de la siguiente manera: `[7,6,2,8,6,5,1]`.

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

FUNCION TAKE

- Como su nombre lo dice, la función TAKE toma un numero de elementos de la lista.
- Esta función funciona de la siguiente manera, **TAKE númeroElementos elementosLista**.
- Cabe mencionar, que TAKE toma desde el primer elemento hasta el numero de posiciones indicados:

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

FUNCION DROP

- La función DROP, al contrario de TAKE esta quita de la lista un numero de elementos.
- Esta función actúa de la siguiente manera, **DROP númeroElementos elementosLista.**

```
Prelude> drop 4 [1,5,6,8,2,6,7]
[2,6,7]
Prelude> drop 2 [1,5,6,8,2,6,7]
[6,8,2,6,7]
Prelude> drop 1 ['h','o','l','a']
"ola"
Prelude> drop 6 [2.0,3.0,4.0,5.0,6.0,7.0,8.0]
[8.0]
```

FUNCION MINIMUM

- La función MINIMUM no devuelve el elemento con el valor mínimo de la lista.
- Por ejemplo de un lista, **[1,2,3,4]** el valor mínimo será **1**.

```
Prelude> minimum [1,2,3,4]
1
Prelude> minimum ['h','o','l','a']
'a'
Prelude> minimum [1.3,1.2,1.1]
1.1
Prelude> minimum [False,True]
False
```

FUNCION MAXIMUM

- La función MAXIMUM no devuelve el elemento con el valor mayor de la lista.
- Por ejemplo de un lista, **[1,2,3,4]** el valor mayor será **4**.

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

```
4
```

```
Prelude> maximum ['h','o','l','a']
```

```
'o'
```

```
Prelude> maximum [1.3,1.2,1.1]
```

```
1.3
```

```
Prelude> maximum [False,True]
```

```
True
```

FUNCION SUM

- La función SUM, suma todos los elementos dentro de la lista. De la siguiente manera:
- Si tenemos una lista [1,2,3,4], el resultado de la función SUM seria **10**.

```
Prelude> sum [1..4]
```

```
10
```

```
Prelude> 1+2+3+4
```

```
10
```

FUNCION PRODUCT

- La función `PRODUCT`, multiplica todos los elementos dentro de la lista. De la siguiente manera:
- Si tenemos una lista `[1,2,3,4]`, el resultado de la función `SUM` seria **10**.

```
Prelude> product [1..7]
```

```
5040
```

```
Prelude> 1*2*3*4*5*6*7
```

```
5040
```

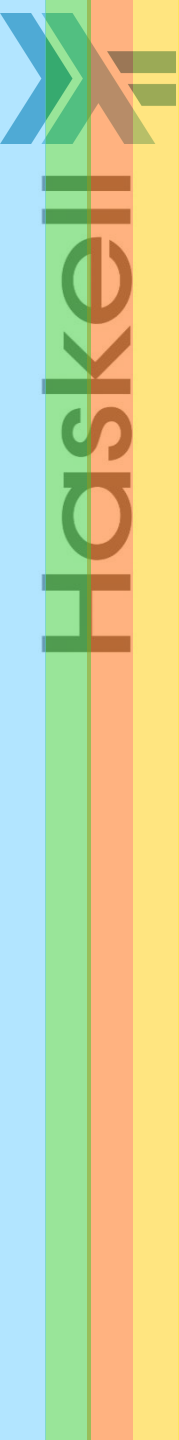
FUNCION `ELEMENT`

- La función `ELEMENT`, nos devuelve un valor booleano (verdadero o falso), con `ELEMENT` se pregunta si el elemento descrito existe dentro de la lista.
- Si el elemento existe nos devolverá un valor True, en caso contrario el valor será False.
- Su notación es la siguiente: **elementoBuscar `element` lista.**

FUNCION `ELEMENT`

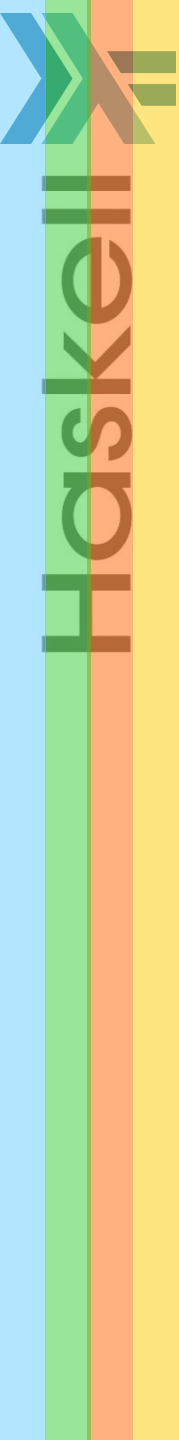
- Donde:
 - elementoBuscar es el elemento que se busca dentro de la lista.
 - `element` es la función de búsqueda del elemento.
 - Lista son los elementos de la lista.

```
Prelude> 8 `elem` [0,9,8,7,6,5]
True
Prelude> 10 `elem` [0,9,8,7,6,5]
False
Prelude> 'a' `elem` "hola"
True
Prelude> 'z' `elem` "hola"
False
```



RANGOS

- Los rangos de una lista nos sirve para dar un secuencia a los elementos dentro de la lista sin necesidad de indicar manualmente cada uno de los elementos de esta lista.
- Por ejemplo, si queremos que nuestra lista valla de los números de 2 en 2, hasta cierto numero dado.
- Existe un representación mas corta que Haskell reconoce y que hace mas sencillo el manejo de lista sobre todo si estas cuentan con un numero grande elementos.



RANGOS

- Los rangos de una lista nos sirve para dar un secuencia a los elementos dentro de la lista sin necesidad de indicar manualmente cada uno de los elementos de esta lista.
- Existe un representación mas corta que Haskell reconoce y que hace mas sencillo el manejo de lista sobre todo si estas cuentan con un numero grande elementos.
- Por ejemplo, si queremos que nuestra lista valla de los números de 2 en 2, hasta cierto numero dado como el 100.

RANGOS

- Entonces la forma de implementar este método de rango seria el siguiente:

`[elemento1,elemrnto2..elementon]`

- Donde:
 - elemento1 es el primer elemento con el cual queremos que inicie nuestra lista.
 - elemento2 es el segundo elemento, y además este sigue la secuencia.
 - *elementon* el ultimo elemento de la lista.

RANGOS

- Entonces, como mencionamos anteriormente si queremos una lista que valla de dos en dos hasta cien, esto se representaría así:

`let x = [2,4..100]`

- Ejemplo en Haskell:

```
Prelude> let x = [2,4..100]
```

```
Prelude> x
```

```
[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]
```

RANGOS

- Cabe mencionar que este método cuenta con una desventaja, ya que las secuencias solo pueden ir de la suma de un numero o resta de ese mismo numero.
- Por consiguiente, si queremos crear una lista con una secuencia que valla de la multiplicación de un numero, Haskell no será capaz de reconocerlo:

```

Prelude> [1,3..30]
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29]
Prelude> let x = [1,3,9..50]
<interactive>:11:15: error: parse error on input ‘..’

```

Haskell no reconoce el patrón al querer multiplicar por 3
 Error ya que no reconoce más de 2 parámetros en las secuencias.

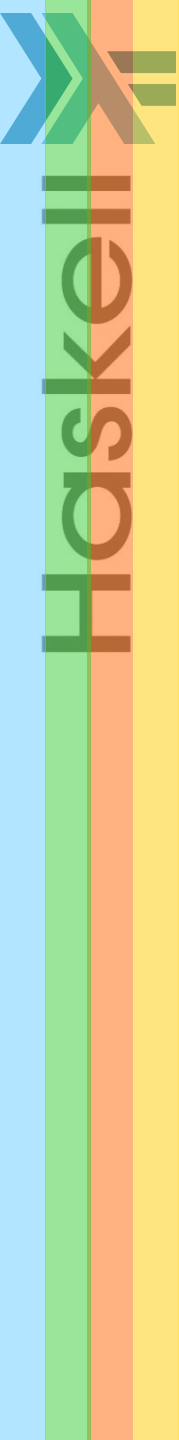
RANGOS

- Pero aun así este método es muy practica ya que hace rápido, fácil, y comprensible las secuencias de las listas.

Prelude> [1,6..36]	←	Suma 5 a cada elemento.
[1,6,11,16,21,26,31,36]		
Prelude> ['a','c'..'g']	←	Salta una letra.
"aceg"		
Prelude> [20,16..0]	←	Resta 4 a cada elemento.
[20,16,12,8,4,0]		
Prelude> [(-3),(-1)..3]	←	Suma 2 a cada elemento.
[-3,-1,1,3]		

FUNCIONES DE LISTAS INFINITAS

- Las listas también pueden tomar formatos infinitos, es decir estas nunca terminan.
- Generalmente las listas infinitas son creadas a través de ciclos, los cuales nos permiten la repetición de elementos dentro de ella.
- Pero, para no tener el problema de que a la hora de compilar y correr nuestro código Haskell y este jamás termine, existen funciones que nos ayuda a manipular y sacarle provecho a este tipo de listas.



FUNCIONES DE LISTAS INFINITAS

- Las listas también pueden tomar formatos infinitos, es decir estas nunca terminan.
- Generalmente las listas infinitas son creadas a través de ciclos, los cuales nos permiten la repetición de elementos dentro de ella.
- La ventaja de usar estos ciclos es que no tienes que escribir todo el código, lo que ahorra tiempo, procesos y deja el código más claro y facilita su modificación en el futuro.

REPEAT

- Este ciclo nos permite repetir infinitamente un elemento determinado.
- Su notación es: **repeat elemento**.
- Donde:
 - Repeat es la función que nos permite repetir el elemento.
 - elemento es el parámetro a repetir infinitamente.

REPEAT

- Ejemplo en Haskell:

CYCLE

- Cycle es un bucle que nos permite repetir una serie de elementos infinitamente.
- Su notación es: `cycle [elemento1,elemento2..elementon]`.
- Donde:
 - Repeat es la función que nos permite repetir los elementos.
 - elemento1, elemento2, elementon los parámetros a repetir infinitamente.

FUNCIONES DE LISTAS INFINITAS

- Pero, se puede usar una función para no tener el problema a la hora de compilar y correr nuestro código Haskell y este jamás termine.
- Esta función fue mencionada anteriormente, nos ayuda a manipular describiendo las veces que queremos que se repitan los elementos, y sacarle provecho a este tipo de listas.
- Esta función es take, como recordamos con take tomamos un numero de elementos de la lista.

FUNCIONES DE LISTAS INFINITAS

- Su forma es de la siguiente manera:

`take númeroElementos (ciclo elemento/s)`

- Donde:
 - Take es la función que toma un numero de elementos.
 - númeroElementos el numero de elementos a tomar.
 - Ciclo son los bucles repeat o cycle.
 - Elemento/s es el/los elemento/s de las lista a repetir.

FUNCIONES DE LISTAS INFINITAS

- Ejemplo en Haskell:

```
Prelude> take 20 (cycle [1,2,3])  
[1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1,2]  
Prelude> take 10 (repeat 6)  
[6,6,6,6,6,6,6,6,6,6]  
Prelude> take 5 (cycle ['a','b'])  
"ababa"  
Prelude> take 10 (repeat 108)  
[108,108,108,108,108,108,108,108,108,108]
```


LISTAS INTENCIONALES

- Las listas intencionales, sirven para ser un conjunto de elementos pero con bajo condiciones más complejas que un lista normal.
- Con este concepto, pueden ser aplicadas muchas transformaciones a lista, y recorrerlas de manera muy breve y concisa
- Estas listas cuentan con una estructura particular ya que se subdivide en 3 partes.

LISTAS INTENCIONALES

- La estructura de la lista, como cualquier otra lista comienza y termina con corchetes [].

`let x = [listaMostrar | filtro , condición]`

- Donde:
 - listaMostrar es el resultado de la lista que se va a mostrar.
 - filtro es la lista filtrada.
 - condición es la son los parámetros bajo los cuales se trabajara la lista (opcional: solo si el problema lo requiere).

LISTAS INTENCIONALES

- Por ejemplo; si queremos una lista que muestre los números impares del 1 al 50, y al final multiplicarlos por 10, la sintaxis seria de esta manera:

```
let x = [ x*10 | x <- [1..50] , x `mod` 2 == 1 ]
```

Lista a mostrar: elementos
multiplicados por 10.

Condición: números
impares.

Filtro: números del
1 al 50.

LISTAS INTENCIONALES

- Ejemplo en Haskell:

```
Prelude> let x = [ x*10 | x <- [1..50] , x `mod` 2==1 ]
```

```
Prelude> x
```

```
[10,30,50,70,90,110,130,150,170,190,210,230,250,270,290,310,330,350,370,390,410,430,450,470,490]
```

LISTAS INTENCIONALES

- Otro ejemplo, queremos los 30 primeros numero de un lista con números a los que se le suma 5.

```
Prelude> let x = [ x | x <- take 30 ( [5,10..] ) ]
```

```
Prelude> x
```

```
[5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100,105,110,115,120,125,130,135,140,145,150]
```

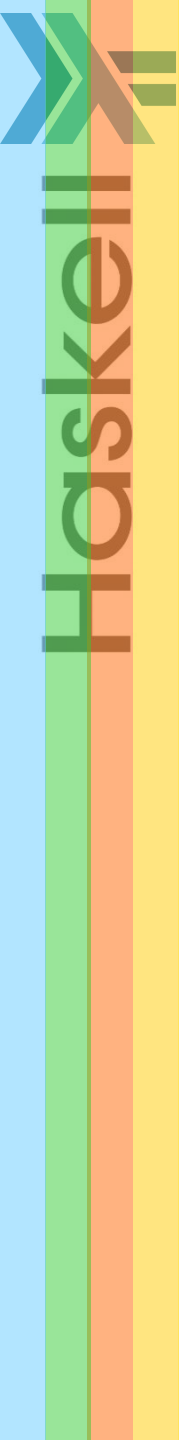
let x = [x | x <- take 30 ([5 , 10 ..])]

Lista a mostrar:
lista ya filtrada.

Filtro: los primero 30 numero
de una lista con números que
se les suma 5

LISTAS INTENCIONALES DOBLES

- Las listas intencionales dobles existen cuando existe una combinación entre estas listas.
- Esto quiere decir que el producto de una lista intencional es utilizado como elemento dentro de otra lista intencional para obtener un resultado final.
- Estas listas intencionales dobles se utilizan para cálculos mas largos, donde deben de existir resultados intermedios, y que necesitan condiciones más estrictas que las listas intencionales nos facilitan.



LISTAS INTENCIONALES DOBLES

- Por ejemplo, si necesitamos conocer el numero de vocales dentro de una cadena.
- El primer paso seria reconocer las vocales de esta cadena, y a continuación hacer una suma del numero de las vocales.
- Si utilizarnos algunos principios de Haskell que no fueran listas intencionales dobles el código de este problema seria enorme.

LISTAS INTENCIONALES DOBLES

- Ejemplo en Haskell.

Imagen 6: Ejemplo listas intencionales dobles en Haskell.

```
Sin título: Bloc de notas
Archivo  Edición  Formato  Ver  Ayuda
mostrarVocales cadena = [vocales | vocales <- cadena, vocales `elem` "aeiou"]
```

Nombre de la
función.

Parámetro.

Lista a mostrar.

Filtro.

Condición.

LISTAS INTENCIONALES DOBLES

- Se buscan que los elementos de la cadena pertenezcas a una lista con los elementos a, e, i, o, u.

```
Prelude> :l Haskell.hs
[1 of 1] Compiling Main                ( Haskell.hs, interpreted )
Ok, modules loaded: Main.
*Main> mostrarVocales "rafael"
"aae"
```

LISTAS INTENCIONALES DOBLES

- Ejemplo en Haskell.

```
Haskell: Bloc de notas
Archivo Edición Formato Ver Ayuda
mostrarVocales cadena = [vocales | vocales <- cadena, vocales `elem` "aeiou"]
sumarVocales suma = sum [1 | x <- (mostrarVocales suma)]
```

Nombre de la
función.

Función.

Filtro.

Parámetros.

Lista a mostrar.

Imagen 7: Ejemplo listas intencionales
dobles en Haskell.

LISTAS INTENCIONALES DOBLES

- Por cada elemento que perteneció a la lista de vocales se sumara 1.

```
Prelude> :r
[1 of 1] Compiling Main          ( Haskell.hs, interpreted )
Ok, modules loaded: Main.
*Main> sumarVocales "pepe pecas pica papas"
8
*Main> sumarVocales "tres tristes tigres"
5
*Main> sumarVocales "la manzana del arbol"
7
*Main> sumarVocales ['h','o','l','a',' ','m','u','n','d','o']
4
```

TUPLAS VS LISTAS

- Una tupla, es una colección de datos con una longitud de n elementos.
- Pero cual es la diferencia entre una tupla y una lista? Bueno la característica general entre estas es que la tupla es conjunto heterogéneo de elementos y la listas un conjunto homogéneo.
- Es decir, los elementos de la tupla pueden variar según el tipo de dato a diferencia de las listas.

TUPLAS VS LISTAS

- La sintaxis de una tupla es la siguiente:

(elemento1, elemento2...elementon)

- Donde:

- *()* identifican que el elemento será un tipo tupla.
- *elemento1, elemento2* son los elementos dentro de esta tupla.
- *elementon* elemento final.

TUPLAS VS LISTAS

- Diferencia entre la mezcla de tipo de datos.

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

```
Prelude> lista
```

```
[1,2,3]
```

```
Prelude> let lista=[1,2,'a']
```

Error en lista por
diferencia de tipo de
datos.

```
<interactive>:7:12: error:
```

```
    No instance for (Num Char) arising from the literal '1'
```

```
    In the expression: 1
```

```
      In the expression: [1, 2, 'a']
```

```
      In an equation for 'lista': lista = [1, 2, 'a']
```

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

```
Prelude> tupla
```

```
(1,2,'a')
```

TUPLAS VS LISTAS

- Otra diferencia entre las tuplas y las listas, es como se manejan dentro de una listas. En el caso de listas serian listas de listas y de la tuplas serian listas de tuplas.
- En las listas de listas las longitudes estas pueden variar:

```
Prelude> let listasListas = [[1,2],[3,4],[5,6,7]]
Prelude> listasListas
[[1,2], [3,4], [5,6,7]]
```

 Longitud 2
  Longitud 2
  Longitud 3

TUPLAS VS LISTAS

- Pero a diferencia de las listas de listas, en las listas de tuplas no puede variar la longitud de las tuplas:

```
Prelude> let listasTuplas = [(1,2),(3,4),(5,6,7)]
```

—————→ Error en tupla por el tamaño de longitud.

```
<interactive>:7:33: error:
  Couldn't match expected type '(t, t1)'
    with actual type '(Integer, Integer, Integer)'
In the expression: (5, 6, 7)
In the expression: [(1, 2), (3, 4), (5, 6, 7)]
In an equation for 'listasTuplas':
  listasTuplas = [(1, 2), (3, 4), (5, 6, 7)]
Relevant bindings include
  listasTuplas :: [(t, t1)] (bound at <interactive>:7:5)
```


TUPLAS VS LISTAS

- Cabe mencionar que una tupla con dos elementos es llamada duplas:

```
Prelude> (1, 'a')  
(1, 'a')
```

- Una tupla de tres elementos es una tripla:

```
Prelude> (1,2, 'b')  
(1,2, 'b')
```

- Una tupla con cuatro elementos es una cuatrupla.

```
Prelude> (1,2, 'a', 'b')  
(1,2, 'a', 'b')
```

FUNCIONES DE TUPLAS

- Anteriormente, se había mencionado cual era la manera de conocer el índice de un elemento de una lista.
- Pero, ¿Cómo saber la forma de conocer la posición de un elemento dentro de una tupla?

FUNCIONES DE TUPLAS

- Veamos que sucede si aplicamos la misma función de las listas en un tupla:

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

```
Prelude> lista !! 1
```

```
2
```

```
Prelude> dupla = (1,'a')
```

```
Prelude> dupla !! 0
```

Error en la función
índice para una tupla.

```
<interactive>:12:1: error:
```

```
    Couldn't match expected type '[a]'
```

```
            with actual type '(Integer, Char)'
```

```
In the first argument of '(!!)', namely 'dupla'
```

```
  In the expression: dupla !! 0
```

```
  In an equation for 'it': it = dupla !! 0
```

```
Relevant bindings include it :: a (bound at <interactive>:12:1)
```

FUNCIONES DE TUPLAS

- Para conocer la posición de los elementos dentro de una dupla es de una manera mas compleja en las listas, ya que existe una función para cada posición.
- Función para primera posición: **fst** (firs por la palabra 'primero' en ingles), primer elemento de la dupla.

```
Prelude> dupla = (1, 'a')  
(1, 'a')  
Prelude> fst dupla  
1
```

FUNCIONES DE TUPLAS

- Función para devolver la segunda posición : **snd** (second por la palabra en ingles 'segundo'), elemento dos de la dupla.

```
Prelude> dupla  
(1, 'a')  
Prelude> snd dupla  
'a'
```

LISTA DE DUPLAS ZIP

- Existe una función que permita la unión entre datos de dos listas, estas unión puede hacerse independientemente del tipo de datos de los elementos de cada lista.
- Entonces, la función zip convierte dos listas en una lista de duplas. Donde, se une el primer elemento con el primer elemento de la segunda lista y así sucesivamente.
- Esta función, es muy útil cuando se necesita colecciones de datos y estos datos comparten un mismo fin.

LISTA DE DUPLAS ZIP

- Por ejemplo, se necesitan los nombre y numero de control, de los alumnos de ingeniera en sistemas del instituto tecnológico de Mexicali.
- Para esto se necesitaría una lista con nombres y otra con los números de control y al final unir las dentro de una lista de duplas.

nombre = ["Rafael", "Cristal", "Agustín", "Daniel"]
noControl=[12490466,12490889,12490468,12490469]

LISTA DE DUPLAS ZIP

- La sintaxis de esta función sería:

`zip nombre noControl`

- Donde:
 - zip es la función de unión entre listas.
 - nombre es la lista con elementos de nombre de los alumnos.
 - noControl es la lista con los números de control de los alumnos.

LISTA DE DUPLAS ZIP

- Ejemplo, en Haskell:

```
Prelude> nombre = ["Rafael","Cristal","Aguntin","Daniel"]
Prelude> nombre
["Rafael","Cristal","Aguntin","Daniel"]
Prelude> noControl=[12490466,12490889,12490468,12490469]
Prelude> noControl
[12490466,12490889,12490468,12490469]
Prelude> zip nombre noControl
[("Rafael",12490466),("Cristal",12490889),("Aguntin",12490468),("Daniel",12490469)]
```

- Pero, ¿Qué pasa si una de las listas cuenta con una longitud mas larga que la otra lista?

LISTA DE DUPLAS ZIP

- Los que sucede es que Haskell realiza un corte a la lista mas larga de manera que las dos listas cuenten con la misma longitud.
- Ejemplo, en Haskell:

```
Prelude> nombre = ["Rafael","Cristal","Aguntin","Daniel","Pablo","Juan"]
Prelude> noControl=[12490466,12490889,12490468,12490469]
Prelude> zip nombre noControl
[("Rafael",12490466),("Cristal",12490889),("Aguntin",12490468),("Daniel",12490469)]
```

COMANDO :t

- El comando :t, permite conocer el tipo de datos de un elemento y dentro de una función conocer a que tipos de datos perteneces los elementos de esta función, y el formato de esta.
- Por ejemplo:

```
Prelude> :t 6
```

```
6 :: Num t => t
```

6 pertenece a un tipo de dato numérico.

```
Prelude> :t True
```

```
True :: Bool
```

True pertenece a un tipo de dato booleano.

```
Prelude> :t "abc"
```

```
"abc" :: [Char]
```

"abc" pertenece a un tipo lista de caracteres.

```
Prelude> :t 'a'
```

```
'a' :: Char
```

'a' pertenece a un tipo de dato carácter.

COMANDO :t

- Pero, como se menciono anteriormente el comando :t con una función no solo nos devuelve el tipo de dato de sus elementos, también el tipo de formato de esta función.
- Función head, que devuelve la cabeza de una lista.

```
Prelude> :t head  
head :: [a] -> a
```

La función head, pertenece a una lista que nos devuelve un solo elemento de la lista.

- Función snd, nos devuelve el segundo elemento de una dupla.

```
Prelude> :t snd  
snd :: (a, b) -> b
```

La función snd, pertenece a una dupla con elementos a y b, esta nos devuelve el elemento b.

COVERSORES SHOW Y READ

- A veces, en nuestro código tenemos datos que nos sirve para la realización de alguna función o para obtener algún resultado.
- Pero, ¿Qué pasa que los cálculos que necesitamos realizar no son permitidos por Haskell ?, por la diferencia entre tipos de datos.
- Existen funciones que nos hacen más fácil la utilización de cálculos entre diferentes tipos de datos, ya que convierten estos datos de su tipo de dato original en otro tipo de dato alterado por la función.

COVERSOR SHOW

- Una de estas funciones conversores es `show`, que convierte cualquier tipo de dato de un elemento, en un tipo de dato cadena.
- Por ejemplo, si necesitamos que el numero de control de un alumno en vez de comportarse como entero se comporte como cadena seria de la siguiente manera:

```
Prelude> noControl = 12490889  
Prelude> show noControl  
"12490889"
```

- Esto puede hacerse independientemente de tipo de dato de entrada.

COVERSOR READ

- La función `read`, permite hacer lo contrario que `show` al convertir una cadena, en otro tipo de dato definido por un segundo elemento.

```
Prelude> read "2" + 3  -----> Convierte una cadena en un entero.  
5  
Prelude> read "True" && False -----> Convierte una cadena en un booleano.  
False  
Prelude> read "[1]" ++ [2,3] -----> Convierte una cadena en una lista de  
[1,2,3] enteros.  
Prelude> read "7.6" * 2.5 -----> Convierte una cadena en un decimal.  
19.0
```

CONCLUSION

- Los lenguajes funcionales ofrecen formas más fáciles de hacer las cosas, los programas funcionales tienden a ser más cortos.
- Los código al tener una estructura más corta facilita la verificación de la corrección de una función.
- Más aun, Haskell cuenta con valores inmutables, esto lleva a tener menos errores. De esta manera, los programas en Haskell son más fáciles de escribir, más robustos y más fáciles de mantener.

REFERENCIAS

- José Javier Villena (2014). *Tutoriales Haskell*. Recuperado de https://www.youtube.com/watch?v=YAKTlImnS-g&list=PLraIUviMMM3fbHLdBJDmBwcNBZd_1Y_hC&index=1.
- José labra G. (1998). *Introducción al lenguaje Haskell*. Recuperado de: <http://ldc.usb.ve/~astorga/haskell0.pdf>.
- Haskell wiki (2003). *Porque Haskell importa*. Recuperado de: https://wiki.haskell.org/Es/Por_que_Haskell_importa.
- *¡Aprende Haskell por el bien de todos!*. Recuperado de: <http://aprendehaskell.es/main.html>.