

# **Instituto Tecnológico Superior de Chicontepec**

**Ingeniería en Sistemas Computacionales**

**Nombre:**

**Camelia Bautista Hernández**

**Docente:**

**Ing. Efrén Flores Cruz**

**Asignatura:**

**Programación Lógica y Funcional**

**Unidad 2**

**Modelo de programación funcional**

**Carpeta de evidencias**

**8° SEMESTRE**

## CONTENIDO

INTRODUCCION .....	3
RESULTADO.....	4
FUNCIONES DEFINIDAS EN HASKELL.....	4
FUNCIONES DE LISTAS .....	4
TUPLAS.....	9
FUNCIONES DE DUPLAS .....	11
COMANDO ZIP .....	12
COMANDO: t .....	13
CONVERSORES SHOW Y READ .....	14
COMBINAR FUNCIONES DE LISTAS INTENCIONALES.....	15
FUNCIONES DE LISTAS INFINITAS.....	15
Clycle .....	16
Funciones definidas succ, min y max.....	16
CONCLUSION.....	18

## INTRODUCCION

En la presente carpeta nos muestra las practicas realizadas en el software de haskell es un lenguaje de programación estandarizado multi-propósito, funcionalmente puro, con evaluación no estricta y memorizada, y fuerte tipificación estática. Las características más interesantes de Haskell incluyen el soporte para tipos de datos y funciones recursivas, listas, tuplas, guardas y encaje de patrones. La combinación de las mismas puede resultar en algunas funciones casi triviales cuya versión en lenguajes imperativos pueden llegar a resultar extremadamente tediosas de programar.

Para poder realizar los siguientes ejercicios en haskell utilizamos los operadores lógicos, operadores de comparación y así como también en cada operación nos define las diferentes funciones en haskell utilizamos las siguientes funciones dependiendo de lo que se pide en la instrucción a continuación nos muestra la lista de funciones definidas en haskell que se utilizaron:

- Funciones: succ, min, max.
- Combinaciones de funciones: listas.
- creación de listas,
- concatenar listas
- Rangos de listas.
- listas intencionales
- listas intencionales dobles
- combinación funciones de listas intencionales
- funciones de listas infinitas
- tuplas: crear tuplas, funciones de tuplas , comando :t .
- conversores show y read.

## RESULTADO

### FUNCIONES DEFINIDAS EN HASKELL

#### FUNCIONES DE LISTAS

##### Listas intencionales

##### Let listas

En la siguiente imagen nos muestra el siguiente ejercicio **let list=** permite crear una lista, asociarlo y poder usar esa lista

```
Prelude> let list= [x | x<-[1..100], x `mod` 2==0]
Prelude> list
[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]
Prelude> let list= [x | x<-[1..100], x `mod` 2==1]
Prelude> list
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,75,77,79,81,83,85,87,89,91,93,95,97,99]
Prelude> |
```

Una lista intencional es una lista donde se filtran los elementos de la lista que queremos que se muestren o no, según las condiciones que buscamos. A continuación, nos muestra los siguientes ejercicios realizados.

1. lista que muestra los números impares del 1 al 20

•2. lista que muestra los números pares del 1 al 20, multiplicados por 10

```
Prelude> let lista = [ x | x <- [1..20] , x `mod` 2 == 1 ]
Prelude> lista
[1,3,5,7,9,11,13,15,17,19]
Prelude> let lista = [ x*10 | x <- [1..20] , x `mod` 2 == 0 ]
Prelude> lista
[20,40,60,80,100,120,140,160,180,200]
Prelude> |
```

## RANGOS DE LISTAS

En el siguiente ejercicio nos muestra que, si quisiéramos tener una lista de los números pares hasta el 100, utilizamos rangos. Haskell cuenta con un sistema inteligente, al cual solo tenemos que indicarle qué queremos generar.

**let lista= [2, 4 .. 100]**

```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> let lista= [2, 4 .. 100]
Prelude> let 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]
Prelude> |
```

**let listas:** En los siguientes ejercicios ingresamos los números (1,2),(3,4),(5,6)unas vez ingresada solo nos muestra los números de acuerdo a cada dato o numero ingresado.

```
WinGHCi
File Edit Actions Tools Help
[Icons]
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude>
Prelude>
Prelude> let listas =[(1,2),(3,4),(5,6)]
Prelude> listas
[(1,2),(3,4),(5,6)]

Prelude> let lista = [(1, "dos"),(2,"uno")]
Prelude> lista
[(1,"dos"),(2,"uno")]
Prelude> |

Prelude> let lista = [23,24,25]
Prelude> lista
[23,24,25]
Prelude> |
```

## LISTAS INTENCIONALES DOBLES

Para combinar listas intencionales seguimos la estructura de una lista intencional simple, sólo que agregaremos la siguiente lista separándola de la primera con una coma. Crear una lista intencional doble que elija de una primera lista (del 1 al 20) los números menores que 10 y de una segunda lista (del 1 al 100) los múltiplos de 10, y devuelva la suma de las dos listas.

```
Prelude>
Prelude> let lista = [ x+y | x<-[1..20], y<-[1..100], x<10, y `mod` 10 == 0 ]
Prelude> lista
[11,21,31,41,51,61,71,81,91,101,12,22,32,42,52,62,72,82,92,102,13,23,33,43,53,63,73,83,93,103,14,24,34,44,54,64,74,84,94,104,15,25,35,45,55,65,75,85,95,105,16,26,36,46,56,66,76,86,96,106,17,27,37,47,57,67,77,87,97,107,18,28,38,48,58,68,78,88,98,108,19,29,39,49,59,69,79,89,99,109]
```

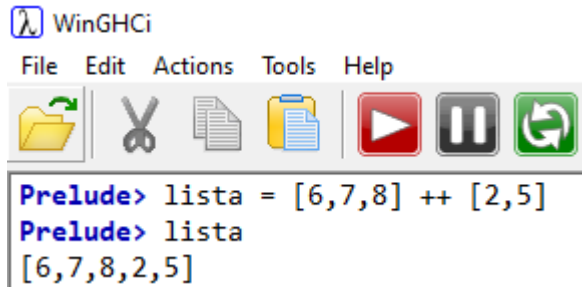
## Listas

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 en lista como podemos observar en los siguientes ejercicios solo nos muestra en listas de acuerdo a los datos ingresados.

```
Prelude> lista = [1,2,3,4,5]
Prelude> lista
[1,2,3,4,5]
Prelude> lista1 = ['a','b','c']
Prelude> lista1
"abc"
Prelude> |
```

Creación de listas y concatenar las listas ++ Haskell tiene listas creadas por defecto ejemplo lista de números y letras

- No se pueden mezclar datos en las listas de Haskell
- Para concatenar cadenas ++



```
WinGHCi
File Edit Actions Tools Help
[Paste] [Cut] [Copy] [Print] [Run] [Pause] [Refresh]
Prelude> lista = [6,7,8] ++ [2,5]
Prelude> lista
[6,7,8,2,5]
```

- “ho” ++ “la” (una lista de caracteres ['h', 'o'] ++ ['l', 'a'])
- Un string es una lista donde cada uno de los elementos de la lista, en realidad es un carácter individual letra por letra

```
Prelude> lista2 = ['h' , 'o'] ++ ['l', 'a']
Prelude> lista2
"hola"
Prelude> |
```

Usamos el operador: concatenar un elemento al inicio de la lista la cual al ejecutar los siguientes datos este operados nos ayuda a juntar los datos números o letras, así como nos muestra en la siguiente imagen

```
Prelude> lista3 = 45: [35, 25, 15]
Prelude> lista3
[45,35,25,15]
Prelude> |
```

## Hola mundo

```
Prelude> lista4 = 'H' : "ola mundo"
Prelude> lista4
"Hola mundo"
Prelude> |
```

Para hacer referencia a un elemento de la lista se usa el operador y nos muestra el resultado de la lista [23,24,25].

```
Prelude> lista !!0 = 23
```

```
Prelude> lista  
[23,24,25]  
Prelude> |
```

En haskell se pueden incluir listas dentro de una lista, es decir

**lista = [[1,2],[3,4]]**


```
Prelude> lista1= [[1,2], [3,4]]  
Prelude> lista1  
[[1,2],[3,4]]  
Prelude> |
```

**lista !!0**

```
Prelude> lista !!0  
23  
Prelude> |
```



En esta imagen nos muestra el siguiente ejercicio **Let nombre** permite crear una lista, asociarlo y poder usar esa lista la cual ingresamos los nombres y en el siguiente ejercicio nos muestra que podemos ingresar edades , así como, también agregamos en las lista la combinación de un número y una letra como por ejemplo [ (1,"A") ]

 WinGHCi

File Edit Actions Tools Help



```
Prelude> let nombre = ["juan","pedro","olivio"]
Prelude> nombre
["juan","pedro","olivio"]
Prelude> let edad = [15,18,25]
Prelude> edad
[15,18,25]
```

```
Prelude> let lis2 =[(1,"A"),(2,"B")]
Prelude> lis2
[(1,"A"),(2,"B")]
```

## TUPLAS

Es la colección de datos que, a diferencia de las listas, permite combinar diferentes tipos de datos.

**Let Tupla = ("UNO",2)**

```
Prelude> let tupla=("UNO",2)
Prelude> tupla
("UNO",2)
Prelude> let tupla=("UNO",2)
Prelude> fst tupla
"UNO"
```

Listas de Duplas deben contener el mismo tamaño y la misma estructura

```
WinGHCi
File Edit Actions Tools Help
[Icons: Folder, Scissors, Document, Document with Arrow, Play, Pause, Refresh, Erase, Settings, Wrench, Question Mark]
Prelude> let tupla = (1, "dos")
Prelude> tupla
(1,"dos")
Prelude> let tripla = (1, "Pedro", 7461134094)
```

## Let tripla

Se pueden escribir con diferentes tipos de datos como por ejemplo en este ejercicio podemos ingresar los diferentes de datos el id, nombre, un número de teléfono.

```
Prelude> let tripla = (1,"Pedro", 7461134094)
Prelude> tripla
(1,"Pedro",7461134094)
Prelude> Let lista= [(1, "dos"),(2, "uno")]
```

Let eje ingresamos los datos para poder calculas el eje de una figura que se desea y cómo podemos darnos cuenta nos da los resultados

```
Prelude> let eje=[(a,b,c)| c<-[1..10], b<-[1..c], a<-[1..b], a^2+b^2==c^2, a+b+c==24]
Prelude> eje
[(6,8,10)]
Prelude> |
```

Let eje = en el siguiente ejercicio nos muestra que de acuerdo a los números que se ingrese, así como nos muestra en la siguiente imagen los números del 1 al 10 .

```
Prelude> let eje = [(a,b,c) | a<-[1..10], b<-[1..10], c<-[1..10]]
Prelude> eje
[(1,1,1),(1,1,2),(1,1,3),(1,1,4),(1,1,5),(1,1,6),(1,1,7),(1,1,8),(1,1,9),(1,1,10),
(1,2,1),(1,2,2),(1,2,3),(1,2,4),(1,2,5),(1,2,6),(1,2,7),(1,2,8),(1,2,9),(1,2,10),
(1,3,1),(1,3,2),(1,3,3),(1,3,4),(1,3,5),(1,3,6),(1,3,7),(1,3,8),(1,3,9),(1,3,10),
(1,4,1),(1,4,2),(1,4,3),(1,4,4),(1,4,5),(1,4,6),(1,4,7),(1,4,8),(1,4,9),(1,4,10),
(1,5,1),(1,5,2),(1,5,3),(1,5,4),(1,5,5),(1,5,6),(1,5,7),(1,5,8),(1,5,9),(1,5,10),
(1,6,1),(1,6,2),(1,6,3),(1,6,4),(1,6,5),(1,6,6),(1,6,7),(1,6,8),(1,6,9),(1,6,10),
(1,7,1),(1,7,2),(1,7,3),(1,7,4),(1,7,5),(1,7,6),(1,7,7),(1,7,8),(1,7,9),(1,7,10),
(1,8,1),(1,8,2),(1,8,3),(1,8,4),(1,8,5),(1,8,6),(1,8,7),(1,8,8),(1,8,9),(1,8,10),
(1,9,1),(1,9,2),(1,9,3),(1,9,4),(1,9,5),(1,9,6),(1,9,7),(1,9,8),(1,9,9),(1,9,10),
(1,10,1),(1,10,2),(1,10,3),(1,10,4),(1,10,5),(1,10,6),(1,10,7),(1,10,8),(1,10,9),
(1,10,10),(2,1,1),(2,1,2),(2,1,3),(2,1,4),(2,1,5),(2,1,6),(2,1,7),(2,1,8),
(2,1,9),(2,1,10),(2,2,1),(2,2,2),(2,2,3),(2,2,4),(2,2,5),(2,2,6),(2,2,7),(2,2,8),
(2,2,9),(2,2,10),(2,3,1),(2,3,2),(2,3,3),(2,3,4),(2,3,5),(2,3,6),(2,3,7),(2,3,8),
(2,3,9),(2,3,10),(2,4,1),(2,4,2),(2,4,3),(2,4,4),(2,4,5),(2,4,6),(2,4,7),(2,4,8),
(2,4,9),(2,4,10),(2,5,1),(2,5,2),(2,5,3),(2,5,4),(2,5,5),(2,5,6),(2,5,7),(2,5,8),
(2,5,9),(2,5,10),(2,6,1),(2,6,2),(2,6,3),(2,6,4),(2,6,5),(2,6,6),(2,6,7),(2,6,8),
(2,6,9),(2,6,10),(2,7,1),(2,7,2),(2,7,3),(2,7,4),(2,7,5),(2,7,6),(2,7,7),(2,7,8),
(2,7,9),(2,7,10),(2,8,1),(2,8,2),(2,8,3),(2,8,4),(2,8,5),(2,8,6),(2,8,7),(2,8,8),
(2,8,9),(2,8,10),(2,9,1),(2,9,2),(2,9,3),(2,9,4),(2,9,5),(2,9,6),(2,9,7),(2,9,8),
(2,9,9),(2,9,10),(2,10,1),(2,10,2),(2,10,3),(2,10,4),(2,10,5),(2,10,6),(2,10,7),
(2,10,8),(2,10,9),(2,10,10)]
```

## FUNCIONES DE DUPLAS

### Snd Tupla

snd= muestra el segundo elemento de la dupla

```
Prelude> snd tupla
2
Prelude> snd (1,5)
5
Prelude> |
```

### Let dupla

letdupla = (1, "dos")

```
Prelude> letdupla = (1,"dos")
Prelude> letdupla
(1,"dos")
Prelude> |
```

## Fst

**fst**= devuelve el primer elemento de la dupla, así como nos muestra en la siguiente imagen ingresados dos datos 1, 2 y el resultado es 1 el primer dato.

```
Prelude> fst (1,2)  
1
```

## COMANDO ZIP

Zip: combinar diferentes tipos de listas devolviendo una lista de duplas

- Creamos la lista nombres.
- Creamos la lista números donde el número.
- Con el comando Zip nos muestra en una dupla el nombre y con el número la edad .

Zip nombre, edad

```
Prelude> zip nombre edad  
[("juan",15),("pedro",18),("olivio",25)]  
Prelude> zip [1..] nombre  
[(1,"juan"),(2,"pedro"),(3,"olivio")]  
Prelude> |
```

Ingresamos con el comando **:m + Data.List** ingresamos al directorio y una vez ingresada agregamos lo siguiente . **Map Data.Set** utilizamos el comando intersperse como ejemplo ingresamos HOLA MUNDO y como resultado nos muestra las letras con las silabas uno por uno separados .

```
Prelude> :m + Data.List
Prelude Data.List> :m + Data.List Data.Map Data.set
```

```
Prelude Data.List Data.Map Data.Set> intersperse '.'"HOLAMUNDO"
"H.O.L.A.M.U.N.D.O"
Prelude Data.List Data.Map Data.Set> |
```

## COMANDO: t

El comando: **t** se utiliza para ver qué tipo de dato tiene un valor o una función

**:t head**

**:t fst**

```
Prelude> :t head
head :: [a] -> a
Prelude> :t fst
fst :: (a, b) -> a
Prelude> |
```

El tipo de dato que se utiliza el tipo de dato si ingresamos la letra A nos muestra el tipo de dato que utiliza y si agregamos un numero nos devuelve el dato en decimal y si realizamos de forma decimal nos devuelve el dato en factorial.



```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> :t "A"
"A" :: [Char]
Prelude> :t 5
5 :: Num p => p
Prelude> :t 2.0
2.0 :: Fractional p => p
```

## CONVERSORES SHOW Y READ

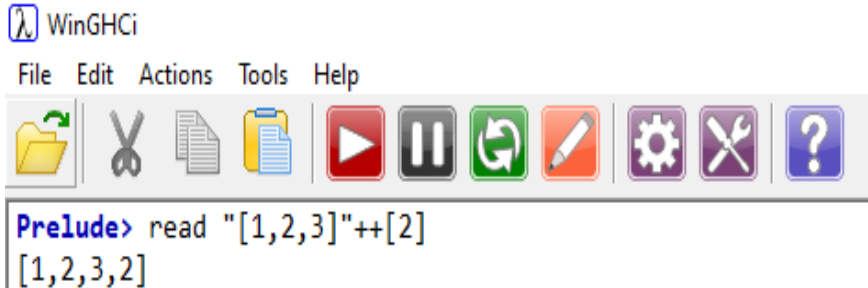
**show** nos ayuda a convertir cualquier lista a cadena de texto



```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> show 5
"5"
Prelude> show True
"True"
```

**Read** permite leer cualquier tipo de datos al tipo del segundo elemento que le pasamos como podemos observar el resultado el 10 lee la suma de dos números.

```
Prelude> read "5" +5
10
```



```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> read "[1,2,3]" ++ [2]
[1,2,3,2]
```









## Min

min: devuelve el valor mínimo de dos argumentos, devuelve el número más pequeño de dos números introducidos, solo soporta dos parámetros

```
Prelude> min 3 89
3
Prelude> min 90 78
78
```

## Max

Max: devuelve el valor máximo de dos argumentos, devuelve el valor más grande de dos parámetros

```
Prelude> max 56 78.5
78.5
Prelude> max 10 12
12
```

## Combinaciones de funciones

En los siguientes ejercicios es la combinación de las funciones de succ, min y max para verificar el resultado de acuerdo a las funciones utilizadas.

```
Prelude> max 4 (succ 10)
11
Prelude> succ (max 8 3)
9
Prelude> succ (max 8 (min 24 9.5))
10.5
Prelude>
```

## CONCLUSION

En las siguientes practicas realizadas es importante conocer las funciones y operadores para poder realizar los ejercicios en el programa haskell, así como, los lenguajes funcionales son más intuitivos y ofrecen más formas y formas más fáciles de hacer las cosas, los programas funcionales tienden a ser más cortos (usualmente entre 2 y 10 veces). La semántica está más cercana al problema que en una versión imperativa, lo que facilita la verificación de la corrección de una función. Más aun, Haskell no permite efectos laterales, 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.