

EST-1141

Lenguajes de Programación

Juan Zamora O.

Agosto, 2024.



PONTIFICIA
UNIVERSIDAD
CATÓLICA DE
VALPARAÍSO



Estructura

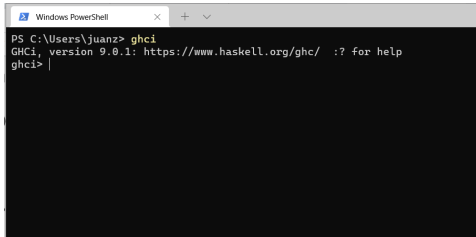
- 1 Introducción a Haskell
- 2 Acerca de la importancia de los Tipos
- 3 Tipos de dato definidos por el Usuario
- 4 Entrada y Salida en Haskell

¿Que es Haskell?

- Lenguaje funcional puro
- Tareas expresadas en términos de funciones
- Una función solo calcula usando los parámetros y retorna un determinado valor
 - No existe efecto lateral (side effect)
- Es perezoso: Postpone la evaluación de una expresión hasta que se necesita el resultado para otros cálculos
 - Opuesto: evaluación impaciente (eager)
- Evaluación perezoso (lazy)
- Permite pensar programas como series de transformaciones sobre datos
- Es estáticamente tipado
 - Tipos de expresiones son resueltos al compilar un programa
 - Varios errores son oportunamente detectados
- Tiene inferencia de tipos

El entorno de Haskell

- Haskell tiene varias implementaciones (Hugs, GHC...)
- Usaremos el *Glasgow Haskell Compile* (GHC)
- GHC tiene 3 componentes:
 - ghc: Compilador que genera código nativo
 - ghci: Interprete interactivo
 - runghc: Programa para ejecutar scripts en Haskell sin tener que compilarlos

A screenshot of a Windows PowerShell terminal window. The title bar at the top says "Windows PowerShell" with standard window controls. The terminal content shows the command prompt "PS C:\Users\juanz>" followed by the user typing "ghci". The system response is "GHCi, version 9.0.1: https://www.haskell.org/ghc/ :? for help". Below this, the prompt "ghci>" is shown with a cursor, indicating the interactive Haskell environment is ready for use.

```
PS C:\Users\juanz> ghci
GHCi, version 9.0.1: https://www.haskell.org/ghc/ :? for help
ghci> |
```

Interacciones básicas con el Interprete

```
1  -- aritmética simple, notación infija y prefija
2  2 + 7
3  (+) 2 7 -- probar sin paréntesis
4  2 * 3
5  2 * (-3) -- probar sin paréntesis
6  (*) 4 8 -- * es una función?
7  succ 5
8  min 9 10
9  max 7 9 -- cómo calcular el max entre mas de 2 números.
   Probar que pasa con max 7 9 11 1
10 (succ 9) + (max 5 4) + 1
```

```
1  -- Lógica booleana, operadores y comparación de valores
2  True && False
3  False || True -- probar reemplazando True con 1
```

```
1  -- Operadores de comparación
2  1 == 1
3  2 < 3
4  4 >= 3.99
5  5 /= 6 -- En Haskell no existe !=
6  not True
```

Precedencia y Asociatividad

- 1 $1 + 4 * 4$
- 2 $5 - 10 / 2$
- 3 $5 - (\mathbf{div} \ 10 \ 2)$
- 4 $5 - \mathbf{div} \ 10 \ 2$
- 5 $5 + 10 + 2$ -- *de izq a der es decir $(5 + 10) + 2$*

Precedence	Operator	Description	Associativity
9 highest	.	Function composition	Right
8	^, ^^, **	Power	Right
7	*, /, 'quot', 'rem', 'div', 'mod'		Left
6	+, -		Left
5	:	Append to list	Right
4	==, /==, <, <==, >==, >	Compare-operators	
3	&&	Logical AND	Right
2		Logical OR	Right
1	>>, >>=		Left
1	<<<		Right
0	\$/, \$!, 'seq'		Right

Listas

- Colecciones homogéneas

```
1 [1,2,3]
2 ["aba","bba","aab"]
3 [1..5]
4 [11,9..1]
5 [10,9..1]
```

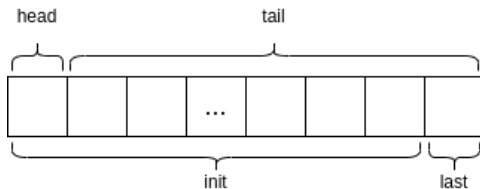

- Operadores sobre listas ++ y :

```
1 [3,1,3] ++ [4,5]
2 [] ++ [False, True] ++ [True]
3 1 : []
4 1 : [2,3] -- Intente cambiar los operandos de orden
```

A continuación se presentan las operaciones más usadas sobre listas, junto con un esquema que muestra las 4 fundamentales:

1	head
2	tail
3	last
4	init
5	length
6	null
7	reverse
8	take
9	drop
10	maximum
11	sum
12	product
13	elem
14	zip

Queda como responsabilidad de l@s estudiantes averiguar cómo se usa y qué hace cada una



Cadenas y caracteres

- Cadenas se definen mediante comillas dobles
- Caracteres se definen mediante comillas simples

```
1 "hola"  
2 putStrLn "Este es un mensaje"  
3 putStrLn ['h','o','l','a'] -- Una cadena es una lista de  
    caracteres
```

```
1 'h' : "ola"  
2 'h' : ['o','l','a']
```

Compilando nuestro primer archivo

- 1 Cree un archivo `hola_curso.hs` en la carpeta que usted elija
- 2 Agregue el contenido:

```
1 main = do
2     putStrLn "Hola_ curso" -- ojo con la tabulación.
3     putStrLn "Adios."
```

- Ejecute desde la consola (dentro del mismo directorio anterior)

`ghc hola_curso.hs`

`./hola_curso`

Actividades

- Ingrese las siguientes expresiones en ghci e indique el tipo asociado (:type XX):

```
1 5 + 8
2 3 * 5 + 8
3 2 + 4
4 (+) 2 4
5 sqrt 16
6 succ 6
7 pred 9
8 sin(pi / 2)
9 truncate pi
10 round 3.7
11 ceiling 3.5
```

- Cree el archivo ciudades.txt con el siguiente contenido:

```
1 Santiago , Chile
2 Buenos Aires , Argentina
3 Montevideo , Uruguay
4 Brasilia , Brasil
5 Quito , Ecuador
```

- contador.hs y otro archivo denominado

```
1  -- archivo contador.hs
2
3  main = interact wordCount
4      where wordCount input = show (length (lines input)) ++
        "\n"
```

- Compile el archivo hs mediante el comando `ghc contador.hs`
- Si resulta todo *OK*, ejecute `./contador < ciudades.txt` ¿Qué está contando el programa?

Acerca de la importancia de los Tipos

- Toda expresión en Haskell tiene un tipo
- Por una parte este tipo *indica* que el valor asociado **comparte** ciertas **propiedades** con otros valores del mismo tipo
- Por ejemplo, podemos sumar números o concatenar listas
- Diremos siempre que una expresión tiene un tipo X p es del tipo X

¿Para qué sirven los tipos en un LP?

- A nivel de hardware un computador procesa bytes
- Un sistema de tipos nos:
 - Entrega **Abstracción**
 - Si sé que un valor de mi programa es de tipo String, *no me interesa o afecta* los detalles de cómo se representa, almacena o recupera ese valor desde el hardware
 - Agrega significado a esos bytes. *Estos bytes corresponden a texto, una reserva de la aerolínea, información personal de un paciente ...*
 - Ayuda a prevenir la mezcla incoherente de tipos
- Un sistema de tipos permite delinear la manera en que pensamos y escribimos código en el Lenguaje
- El sistema de tipos de Haskell permite pensar en un nivel muy abstracto, permite además *escribir* programas concisos y efectivos

El sistema de tipos de Haskell

- Fuerte
- Estático
- Con inferencia automática

Tipado Fuerte

- Sistema garantiza que un programa no puede contener **ciertos tipos de error**
- ¿Que errores?
 - Formar expresiones sin sentido
 - E.g. Usar una función como un número entero o entregarle un String a una función que espera un valor flotante.
 - Convertir un valor de un tipo a otro. Haskell no convierte automáticamente tipos de valores. (Esto no siempre es lo más conveniente en términos de eficiencia)
- Un sistema con tipado fuerte permite atrapar errores difíciles de detectar en el código antes de que causen problemas
- Un sistema con tipado fuerte tratará como válidas una **menor cantidad** de expresiones en comparación con otro sistema más frágil (*weak*)

Tipado Estático

- Esto significa que el compilador conoce el tipo de cada valor y expresión al compilar el código (Antes generar el ejecutable)
- Esto genera programas que difícilmente se caeran por errores triviales
- Por ejemplo, al intentar usar la expresión `True && "False"`, el compilador infiere los tipos y detecta que **no** es posible emparejarlos

1

```
True && "False"
```

- Otros lenguajes (como Python) usan el denominado *Duck Typing*
 - El tipo asociado a un objeto no es tan importante como la compatibilidad con la operación que se está intentando realizar
 - Los tipos no se verifican sino que se verifica la existencia de métodos o atributos que permitan la evaluación de una expresión

```
In [6]: def calcular(a,b,c):  
...:     return (a + b) * c  
...:  
  
In [7]: calcular(1,2,3)  
Out[7]: 9  
  
In [8]: calcular([1,2,3],[4,5,6], 2)  
Out[8]: [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]  
  
In [9]: calcular('pera',' y naranja ', 3)  
Out[9]: 'pera y naranja pera y naranja pera y naranja '
```

Tipado con inferencia automática

- Tipos son deducidos al compilar un código
- Haskell permite también la especificación de el tipo asociado a un valor

Acerca del sistema de tipos de Haskell

- Tiene una combinación potente entre estático y con inferencia automática
 - Permite escribir código conciso, seguro y cuyo código tiene gran poder expresivo
- Gracias a este, el compilador es capaz de indicar fallas en el código en una etapa temprana de desarrollo

Tipos básicos

- Char: Caracter **Unicode**
- Bool: Valor binario (lógico)
- Int: Valor entero con signo de tamaño fijo (32 bits generalmente)
- Integer: Valor entero de tamaño no acotado (no se usa tan a menudo como tipo Int)
- Double: Valor con punto flotante (existe el tipo Float, pero se recomienda uso de Double)

Para especificar el tipo de un determinado valor se utilizan ::

```
1 :type 32323 -- Haskell infiere el tipo  
2 :type 3 :: Int -- se denomina firma de tipo
```

Tuplas

- Colección de tamaño fijo
- Se delimita mediante (...)
- Puede contener valores de distintos tipos
- ("978-1784394707", "Learning Haskell for Data Analysis", 2015)
- El tipo asociado a una tupla está determinado por número, ubicación y tipo de sus miembros

```
1 :type (False, 'a')  
2 :type ('a', False)
```

Listas

- El tipo `list` es polimórfico, debido a que puede contener valores de cualquier otro tipo

```
1 :type [1,2,3]
2 :type [True, True, False]
3 :type tail [1, 6, 11]
4 :type head [1, 6, 11]
5 :type [[True], [False, False, True]]
```

Listas por comprensión

- elementos se describen a través de propiedades que tienen en común
- Ej. Números pares entre 0 y 20

```
1 [x | x <- [0..20], (x `mod` 2 == 0) ]
```

[0,2,4,6,8,10,12,14,16,18,20]

- Ej. Números entre 1 y 5 más 3

```
1 [x+3 | x <- [1,2,3,4,5]]
```

[4,5,6,7,8]

Clases de tipos

Categorías que agrupan tipos según un comportamiento y soporte de operaciones definidos

Eq: Cualquier tipo que soporte verificación de igualdad ($=$ y \neq) entre dos valores (casi todas)

Ord: Cualquier tipo que soporte comparación ($<$, \leq , $>$, \geq) entre dos valores

Show: Cualquier tipo que pueda ser presentado como String. Es posible verificar que un determinado tipo es parte de esta clase porque sus valores soportan el uso de la función show.

Read: Hace lo opuesto. Es decir, toma un String y retorna un tipo (debe ser miembro de Read). Se puede verificar con la función read. Ejemplo: read "8.2" + 3.8.

Enum: Tipos cuyos valores están secuencialmente ordenados. Los tipos en esta clase pueden ser usados en rangos de listas, ya que soportan el uso de las funciones succ y pred. Ejemplo: succ 'a'.

Num: Tipos cuyos valores pueden ser usados como números.



Construcción de Funciones en Haskell

Todas las funciones en Haskell siguen 3 reglas. Son estas reglas las que obligan a que una función en Haskell se comporte como una función matemática:

- Deben tomar un argumento
- Deben retornar/generar un valor
- Cada vez que una función es invocada con el mismo argumento, **debe** retornar el mismo valor

Funciones en el interprete vs archivos independientes

- ghci no es muy cómodo para escribir funciones, debido a que acepta solamente un conjunto restringido de instrucciones del lenguaje
- Deberemos crear archivos de código, los cuales podrán ser posteriormente *compilados* con ghc o cargados directamente desde ghci
- Comencemos con una función que sume dos números. Creemos el archivo calcula_suma.hs con el siguiente contenido:

```
1  -- función sumar que toma dos números y calcula la suma de
   estos
2  sumar a b = a + b
```

- Luego, carguemos el archivo dentro de ghci y ejecutamos la instrucción:
ghci> :load calcula_suma.hs

Algunos aspectos importantes respecto del Ejemplo

- Haskell no tiene una instrucción de retorno
- Una función es una sola expresión y no una *secuencia* de instrucciones
- El **valor de esta expresión es el resultado de la función**
- = representa significado o definición
- En Haskell las variables son solamente un medio para denominar expresiones
 - Una vez asociada a una expresión, su valor no cambia y representa siempre la misma expresión
 - No se asocian con una ubicación en memoria principal
- Construya un archivo denominado `ejemplo_variable_haskell.hs` con el siguiente contenido:

```
1 x = 10
2 x = 11
```

- Intente cargarlo en `ghci` o compilarlo con `ghc`

Estructura condicional

- Haskell tiene una expresión para `if`
- Su sintáxis puede ser mostrada mediante el siguiente ejemplo:
 - Construya un archivo denominado `ejemplo.hs` con el contenido:

```
1  -- archivo de ejemplo:
2  mayor a b = if a >= b
3              then a  -- la tabulación es muy importante
4              else b
```

- Cargue el archivo desde ghci y ejecute luego: mayor 23 32

- La estructura if tiene 3 componentes:
 - Expresión de tipo Bool que se denomina *predicado*
 - Palabra clave then seguida de *otra expresión*★. Esta expresión será usada como el valor resultante del if cuando el predicado es evaluado como True.
 - Palabra clave else seguida de *otra expresión**. Esta expresión será usada como el valor resultante del if cuando el predicado es evaluado como False.
- Las expresiones que acompañan a then y else (★ y *) se denominan ramas y **deben** tener el **mismo tipo**
- La clausula else es obligatoria

Actividades

- Construya una función que indique si un número dado es par o no lo es.
- Construya una función que tome una lista como parámetro y entregue la cantidad de elementos en ella

Funciones Anónimas

- Se les llama funciones λ
- Son funciones sin un nombre, es decir se usa su propia expresión de definición para invocarlas
- Solamente toman un valor y retornan un valor
- **No** pueden **usarse sin entregarle** un **valor** para su(s) parámetro(s)
- Ejemplo: La función suma puede escribirse como una función lambda:
 $\lambda x y \rightarrow x + y$
 - Deben entregarsele valores a sus parámetros, de otra forma esta expresión genera un error

Polimorfismo en Haskell

- Una función polimórfica es aquella cuya definición permite aplicarla sobre argumentos de distintos tipos
- Por ejemplo: `take (take 3 "lenguaje")`
 - Funciona correctamente sin importar el tipo contenido en la lista
 - Al mirar la firma de esta función, encontramos una variable de tipo `a`

```
1 Prelude> :type take  
2 take :: Int -> [a] -> [a]
```

- Esta firma puede leerse como: “take toma una lista con elementos del tipo `a` y retorna un valor del mismo tipo `a`”
- Si una función tiene una variable de tipo en su firma, entonces la función es polimórfica

Comentario respecto a las variables de tipo

- Las variables de tipo siempre comienzan con letras minúsculas
- No se pueden confundir con las variables normales debido a que la especificaciones de tipos y de las funciones van separadas

Definiendo tipos de funciones

- Anteriormente hemos visto cómo escribir funciones indicando únicamente variables para cada argumento
- Hemos dejado a Haskell la tarea de inferir los tipos automáticamente
- También es posible especificar esta información al definir la función:

```
1  -- multiplica dos números
2  f1 :: Int -> Int -> Int
3  f1 a b = a * b
```

- Pruebe usar la función anterior con la expresión `f1 3 3.0`
- Escriba nuevamente la función como:

```
1  -- multiplica dos números de cualquier tipo
2  -- => es una restricción de tipo
3  f1 :: Num a => a -> a -> a
4  f1 a b = a * b
```

Definiendo con Patrones

- Se identifican distintos *moldes* para los argumentos o datos de entrada
- La función se define mediante distintos cuerpos (varias definiciones independientes, una por cada *molde* o escenario de uso)
- Por ejemplo, consideremos la función adivina que indica si el número entregado como argumento coincide con el número secreto:

```
1 adivina :: (Integral a) => a -> String  
2 adivina 7 = "¡Encontró el número secreto!"  
3 adivina x = "¡Disculpa, pero no adivinaste!"
```

- Al invocar a la función los patrones se revisan desde arriba hacia abajo, y cuando se produce el calce, se utiliza ese cuerpo.
- Definamos la función factorial de un número de manera recursiva como el producto entre el número y el factorial de su predecesor:

```
1 factorial :: (Integral a) => a -> a
2 factorial 0 = 1
3 factorial n = n * factorial (n - 1)
```

- **Siempre** debe incluirse un *patrón atrapa-todo* que aborda cualquier caso no cubierto por los patrones anteriores

Patrones sobre listas

- Un patrón como `x:xs` asociará la cabeza de la lista a `x` y el resto de ella a `xs`

```
1 head' :: [a] -> a
2 head' [] = error "¡No es posible realizar esta operación en una lista vacía!"
3 head' (x:_) = x
```

```
1 head' [7,9,2,10]
```

- Es posible aplicar la misma idea a más de un elemento del tope de una lista

```
1 primeros3 :: [a] -> [a]
2 primeros3 [] = error "¡No es posible realizar esta operación en una lista vacía!"
3 primeros3 (x:y:z:_) = [x,y,z]
```

```
1 primeros3 [7,9,2,10]
```

[7,9,2]

- Consideremos ahora el siguiente ejemplo:

```
1 sumaR :: (Num a) => [a] -> a
2 sumaR [] = 0
3 sumaR (x:xs) = x + sumaR xs
```

```
1 sumaR [5,2,1]
```

- Otro recurso interesante es @ para poder definir un patrón sin perder la referencia al objeto completo:

```
1 primeraLetra :: String -> String
2 primeraLetra "" = "¡String_vacía!"
3 primeraLetra palabra@(x:xs) = "La_primera_letra_de_" ++
    palabra ++ "_es_" ++ [x]
```

```
1 primeraLetra "Juan"
```

"La primera letra de Juan es J"

Escoltas en funciones

- Siguen siempre a una función con sus parámetros
- Son expresiones booleans junto con predicados ... similares a las condiciones.
- Admiten multiples condiciones ... bastante parecidos a if , elif ... else en Python
- Se usan *pipes* | para indicarlos

```
1  imc :: (RealFloat a) => a -> a -> String -- estado según
    indice de masa corporal
2  imc peso estatura
3      | peso / estatura ^ 2 <= 18.5 = "¡Muy_bajo!"
4      | peso / estatura ^ 2 <= 25.0 = "¡Se_supone_que_es_
    normal!"
5      | peso / estatura ^ 2 <= 30.0 = "¡Mejor_no_digo_que_estás_
    un_poco_pasado!"
6      | otherwise      = "¡Felicitaciones!_sacaste_el_premio_
    gordo."
```



```
1 putStrLn (imc 84 1.77)
```

¡Mejor no digo que estás un poco pasado!

¿Como implementaría la función max entre dos números usando escoltas?

Usando Where dentro de las funciones

- En ocasiones se repiten expresiones múltiples veces
- Por ejemplo, en el caso de `imc` con la expresión `peso / estatura ^ 2`
- La redundancia en este caso no es algo deseable.. dificulta interpretabilidad y mantenimiento del código
- **Existe** una manera de asociar estas expresiones a un solo nombre mediante la instrucción `where`

```
1  imc :: (RealFloat a) => a -> a -> String -- estado según
    índice de masa corporal
2  imc peso estatura
3      | valor <= flaco = "¡Muy_bajo!"
4      | valor <= medio = "¡Se_supone_que_es_normal!"
5      | valor <= pasado = "¡Mejor_no_digo_que_estás_un_poco_
    pasado!"
6      | otherwise      = "¡Felicitaciones!_sacaste_el_premio_
    gordo."
7      where valor = peso / estatura ^ 2
8              flaco = 18.5
9              medio = 25.0
10             pasado = 30.0
```

```
1 | putStrLn (imc 84 1.77)
```

```
1 | `¡Mejor no digo que estás un poco pasado!`
```

Si decidimos cambiar la manera de calcular el índice, solo modificamos el código en un solo lugar.

Funciones de Alto Nivel

- Nos referimos a funciones cuyos argumentos son funciones y su retorno también
- Es importante encerrar entre () el primer parámetro, debido a que \rightarrow es asociativo por derecha

```
1  aplicar2veces :: (a -> a) -> a -> a
2  aplicar2veces f x = f (f x)
```

```
1  aplicar2veces (\p -> 2 * p) 5
2  aplicar2veces (+ 3) 10
3  aplicar2veces (++) "waka" "waka"
```

20

16

"waka waka waka"

Algunas funciones nativas

- zip: Combina los elementos de dos listas en una lista de tuplas
- zipWith: Combinar dos listas usando una función binaria
- map: Toma una función y una lista para luego aplicar la función a cada elemento
- filter : Toma un predicado (función que retorna True o False) y una lista, entregando finalmente solo aquellos elementos que satisfacen el predicado

```
1 zip [1,2,3] [4,5,6]
2 zipWith (+) [1,2,3] [10,20,30]
3 zipWith replicate [1,2,3] [10,20,30]
4 map (replicate 2) [1..5]
5 filter even [1..10]
```

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

[11,22,33]

[[10],[20,20],[30,30,30]]

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

[2,4,6,8,10]

Construcción de un programa ejecutable

```
-- ejemplo.hs Este es mi primer código ejecutable de ejemplo  
main = do  
    print "¡Hola, estoy corriendo!"
```

Línea con comentario
descriptivo acerca del contenido del archivo

Comienzo de
función principal

Contenido de la función principal

Introducción

- Cada LP tiene sus tipos de dato básicos
- Frecuentemente es posible resolver un gran número de problemas con estos tipos (Int, Float o String por ejemplo)
- A veces, se adaptan los tipos nativos de manera un tanto *forzada*
- Ejemplo: Se desea representar un círculo mediante su radio y ubicación en el plano 2D.
 - Una alternativa es usar tuplas como (0.501, 1.208, 4)
 - No resulta muy intuitiva y puede generar confusión respecto a lo que hay en cada componente
 - Se puede hacer bastante incómodo acceder a cada atributo usando calce de patrones
- Se pueden hacer sinónimos de tipos ya existente

```
1 type NombrePila = String  
2 type Apellido = String  
3 type Edad = Int
```

```
1 -- función que entrega un String con el nombre, apellido y  
  edad de una persona  
2 strInfoPersona :: NombrePila -> Apellido -> Apellido -> Edad  
  -> String  
3 strInfoPersona n a1 a2 e = a1 ++ "␣" ++ a2 ++ ",␣" ++ n ++ " /␣"  
  " ++ (show e) ++ "␣años"
```

```
1 | strInfoPersona "juan" "zamora" "osorio" 37
```

"zamora osorio, juan/ 37 annos"

- No solo se pueden renombrar tipos de a uno

```
1 | type TipoPersona = (NombrePila, Apellido, Apellido, Edad) --  
    importante la mayúscula en el nombre del tipo
```

```
1 obtenerPrimerApellido :: TipoPersona -> Apellido
2 obtenerPrimerApellido (_, a, _, _) = a
```

```
1 obtenerPrimerApellido ("juan", "zamora", "osorio", 37)
```

"zamora"

- Se pueden definir otros **nuevos**
 - En otro lenguaje se hubieran usado los tipos nativos para definir uno nuevo

1 **data** Computador = Escritorio | Portatil -- *el tipo Computador tendrá 2 constructores de datos posibles*

- En este caso:
 - Computador corresponde al *constructor de tipo*
 - Este constructor también puede tomar argumentos
 - Escritorio y Portatil son constructores de datos. Son usados para crear instancias concretas del tipo
- Al separar los constructores de dato con | se indica que el tipo Computador puede ser una instancia de Escritorio o Portatil

Ejercicio

- Defina el tipo Persona e incluya el tipo de sangre en su información personal
 - Se compone de 2 elementos: tipo ABO (A, B, AB, O) y valor Rhesus (Rh + o Rh -)
- Escriba una función que tome dos instancias del tipo sanguineo y retorne True cuando es posible la transfusión entre ambos.
 - A puede donar a A y AB
 - B puede donar a B y AB
 - AB puede donar solo a AB
 - O puede donar a cualquiera ““

```
1 type TipoABO = String
2 type Rh = String
3
4 -- En este caso no podemos colocar data TipoSangre = TipoABO
   | Rh , ya que necesitamos ambos
5 data TipoSangre = TipoSangre TipoABO Rh -- este constructor
   de datos con argumentos
```

Primero definimos una función que nos permita obtener el tipo ABO de un dato TipoSangre

```
1 obtenerABO :: TipoSangre -> TipoABO
2 obtenerABO (TipoSangre x _) = x
```

ahora podemos crear un dato de este tipo con el constructor de datos y también usar la función

```
1 pac1 :: TipoSangre
2 pac1 = TipoSangre "A" "POS"
3
4 pac2 :: TipoSangre
5 pac2 = TipoSangre "O" "NEG"
6
7 pac3 :: TipoSangre
8 pac3 = TipoSangre "AB" "POS"
```



```
1 obtenerABO (TipoSangre "AB" "POS") -- uso del constructor de  
  datos  
2 obtenerABO pac1  
3 obtenerABO pac2  
4 obtenerABO pac3
```

"AB"

"A"

"O"

"AB"

Ahora definamos la función siguiendo las condiciones de transfusión planteadas en el enunciado

```
1 puedeDonar :: TipoSangre -> TipoSangre -> Bool
2 puedeDonar paci pacj
3     | aboi == "A" && (aboj == "A" || aboj == "AB") = True
4     | aboi == "B" && (aboj == "B" || aboj == "AB") = True
5     | aboi == "AB" && aboj == "AB" = True
6     | aboi == "O" = True
7     | otherwise = False
8     where aboi = obtenerABO paci
9           aboj = obtenerABO pacj
```

```
1 puedeDonar pac1 pac2
2 puedeDonar pac2 pac1
3 puedeDonar pac2 pac3
4 puedeDonar pac1 pac3
5 puedeDonar pac3 pac1
```

False

True

True

True

False

Solución alternativa sin usar Strings para TipoABO Y RH

```
1 data TipoABO = A | B | AB | O
2 data TipoRh = Pos | Neg
3
4 -- En este caso no podemos colocar data TipoSangre = TipoABO
   | Rh , ya que necesitamos ambos
5 data TipoSangre = TipoSangre TipoABO TipoRh -- este
   constructor de datos con argumentos
```

Para poder mostrar datos con estos tipos necesitamos una función por tipo:

```
1  mostrarABO :: TipoABO -> String
2  mostrarABO A = "A"
3  mostrarABO B = "B"
4  mostrarABO AB = "AB"
5  mostrarABO O = "O"
6
7  mostrarRh :: TipoRh -> String
8  mostrarRh Pos = "+"
9  mostrarRh Neg = "-"
10
11 mostrarTipoSangre :: TipoSangre -> String
12 mostrarTipoSangre (TipoSangre abo rh) = mostrarABO abo ++
    mostrarRh rh
```

Probemos estas funciones

```

1 pac1 :: TipoSangre
2 pac1 = TipoSangre A Pos
3
4 pac2 :: TipoSangre
5 pac2 = TipoSangre O Neg
6
7 pac3 :: TipoSangre
8 pac3 = TipoSangre AB Pos
9
10 mostrarTipoSangre pac1
11 mostrarTipoSangre pac2
12 mostrarTipoSangre pac3

```

”A+”

”O—”

nuestra función puedeDonar ahora será definida usando patrones

```
1 puedeDonar :: TipoSangre -> TipoSangre -> Bool
2 puedeDonar (TipoSangre O _) _ = True
3 puedeDonar _ (TipoSangre AB _) = True
4 puedeDonar (TipoSangre A _) (TipoSangre A _) = True
5 puedeDonar (TipoSangre B _) (TipoSangre B _) = True
6 puedeDonar _ _ = False
```

```
1 puedeDonar pac1 pac2
2 puedeDonar pac2 pac1
3 puedeDonar pac2 pac3
4 puedeDonar pac1 pac3
5 puedeDonar pac3 pac1
```

False

True

True

True

False

Alternativa ultra-cómoda para definir tipos

- Sintáxis de registro usando {}

```
1 data Estudiante = Estudiante {  
2     primerNombre :: String ,  
3     apellidoPat  :: String ,  
4     apellidoMat  :: String ,  
5     edad        :: Int ,  
6     estatura    :: Float ,  
7     carrera     :: String  
8 } deriving (Show)
```

Usemos el modo interactivo para crear un objeto de este tipo

```
1 let e1 = Estudiante "Juan" "Zamora" "Osorio" 19 1.75  
   "Ingenieria_Estadistica"
```

- Haskell automáticamente genera métodos de acceso a los campos

```
1 edad e1  
2 estatura e1
```

19

1.75

Porqué definir tipos propios?

- Mejora la legibilidad del código
- Permite que el compilador verifique que se usa el tipo adecuado y no otro que accidentalmente calce, pero que genere un error posterior.

Introducción

- Transversal a cualquier lenguajes son los requerimientos de entrada y salida
- **Entrada:** Solicitar ingreso externo de datos (desde teclado o archivos en el disco)
- **Salida:** Desplegar datos en algún medio (pantalla o archivos en el disco)
- Al depender de el ingreso de datos se viola la Transparencia referencial
 - Haskell resuelve esto con el tipo IO
 - Cualquier función que use IO será marcada como proveniente de IO

Revisemos el ejemplo:

```
1 funcionQueSuma1 val1 val2 = (val1 + val2 + 1)
2
3 funcionQueSumaX val1 val2 = do  -- do permite secuenciar
   instrucciones de IO
4   putStrLn "Ingresar un número:"
5   xln <- getLine
6   let x = read xln
7   return ((val1 + val2 + x))
```

- Será posible ejecutar `ghci> (funcionQueSuma1 4 5)+ (funcionQueSuma1 2 5)`
- Pero no se podrá `ghci> (funcionQueSumaX 4 5)+ (funcionQueSumaX 4 5)`

Las funciones `putStrLn` y `getLine`

- Ambas retornan lo que se denomina como acciones de entrada y salida (*IO actions*)
 - Esto quiere decir que son *funciones* que violan alguno de los 3 principios: todas toman un valor, todas retornan un valor y mismo argumento => mismo resultado.

```
1 :t putStrLn
2 :t getLine
```

- Esto quiere decir que no entregan valor alguno (`putStrLn`) y que no reciben argumento (`getLine`)
- El valor generado en cualquier función marcada como IO no puede ser usado fuera de ella

El bloque do

- Permite especificar secuencias de instrucciones con entrada y salida.
- por ejemplo

```
1 do {putStrLn "Linea_1"; putStrLn "Linea_2"; putStrLn "Linea_3"}
```

Linea 1

Linea 2

Linea 3

- Con ellos podemos *escapar* de la prisión de las funciones marcadas como IO
- Cuando el resultado de una función marcada como IO se usa en una asignación, empleamos <-
- Para crear variables que reciben valores de funciones no marcadas como IO se usa =

Revisemos este ejemplo

```
1  digaHola :: String -> String
2  digaHola n = "Hola" ++ " " ++ n ++ "!"
3
4  main :: IO ()
5  main = do
6      putStrLn "Indiqueme su nombre por favor:"
7      nom <- getLine -- recordar que no recibe argumentos
8      let mensaje = digaHola nom -- dentro del bloque `do` se
9                          -- usa como si fuera un String común y corriente.
10     putStrLn mensaje
```

- >> runghc archivo_io.hs