# Facultad de Ciencias - UNAM
# Estructuras Discretas 2023-1
# Práctica 2: Introducción a Haskell (Parte 2)

Laura Friedberg Gojman
Saúl Adrián Rojas Reyes
José Manuel Madrigal Ramírez

10/febrero/2023
**Fecha de entrega: 24/febrero/2023**

## Instrucciones

Junto a este PDF, subí un archivo llamado `Practica2.hs`. Descárgalo y ábrelo en tu editor de texto preferido. Abre también una terminal situada en la carpeta donde guardaste el archivo.

La dinámica de trabajo será similar a la que seguimos en la práctica anterior. Continuaremos con la introducción a los conceptos básicos de Haskell. Abordaremos los pares, las listas y su convergencia con funciones de múltiples argumentos que nos permitirán hacer cosas interesantes.

## Ejercicios

### Pares ordenados

1. En Haskell tenemos la posibilidad de construir pares ordenados con valores de distintos tipos. La sintaxis para ello es la siguiente:

   ```
   <identificador> :: ( <tipo1>   , <tipo2>   )
   <identificador> = ( <valor1> , <valor2>   )
   ```

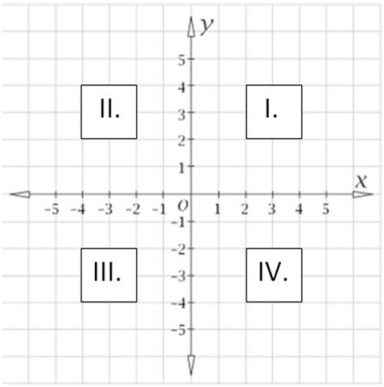
   Por ejemplo, podemos tener un par ordenado de un entero con un carácter:

   ```
   parUno :: ( Int, Char )
   parUno = ( 3, 't' )
   ```

   Decimos que `( Int, Char )` es el tipo de nuestro par.

   *Escribe tres ejemplos de pares compuestos por tipos básicos distintos.*

2. Considera los cuadrantes del plano cartesiano en la siguiente imagen. *Escribe una función que tome un par de números reales y suponiendo que son las coordenadas de un punto, devuelva el cuadrante al que dicho punto pertenecería. Utiliza guardas en tu definición.*




Aquí hay unos ejemplos de su ejecución en la terminal:

```
Main> cuadrante (2,3)
Main> 1
Main> cuadrante (-2,-3)
Main> 3
```

Pista: Comienza escribiendo la firma de tipo, la función va de los pares de números reales a los enteros:

*cuadrante:* $R^2 \to Z$

¿Qué tipo de dato nos sirve para modelar números reales?

Una vez definida la firma de tipo utiliza guardas para especificar la salida de la función según las características de la entrada. Por ejemplo, si ambos datos del par son menores a cero devolvemos `3` para indicar que el los números del par corresponden a las coordenadas de un punto situado en el tercer cuadrante.

3. *Escribe una función llamada* `modulo` *que calcule el módulo de un vector en* $R^2$.
Algunos ejemplos de su ejecución serían:

```
*Main> modulo (4,3)
5.0
*Main> modulo (2, -5)
5.385164807134504
```

Pista: En el módulo Prelude de GHC existe un conjunto de funciones predefinidas que podemos utilizar en nuestros programas. Para calcular la raíz cuadrada de un número tenemos:

```
sqrt::float -> float
```

Puedes probar esta función en la terminal:

```
*Main> sqrt 25
5.0
Main> sqrt 7
8.426149773176359
```

Finalmente puedes utilizarla dentro de la definición de la función `modulo` de este ejercicio.

4. Si queremos escribir una función con más de un valor de entrada, por ejemplo, con tres valores, utilizamos la siguiente sintaxis para especificar su firma de tipo:

   `<identificador>:: <Tipo1> -> <Tipo2> -> <Tipo3> -> <Tipo4>`

   Donde `<Tipo1>` , `<Tipo2>` , y `<Tipo3>` son los tipos de los datos de entrada. Mientras que `<Tipo4>` es el tipo del dato de salida.

   *Escribe una función que tome tres enteros y devuelva su suma. Recuerda comenzar escribiendo la firma de tipo de la función seguida de su definición.*

   Aquí hay un ejemplo de su ejecución en terminal:

   ```
   *Main> suma 2 3 4
   9
   Main> suma 4 31 (-21)
   13
   ```

   Observaciones:

   - Al aplicar una función no utilizamos paréntesis para envolver sus parámetros.
   - Debemos escribir los números negativos entre paréntesis para evitar ambigüedad con la resta.

5. *Escribe una función que calcule la media de cinco números reales dados.*

   Veamos algunos ejemplos de cómo debería ejecutarse:

   ```
   *Main>media 4.3 32.0 21.2 2 3
   12.5
   *Main>media (-3) 42 3.0 10 11
   12.6
   ```

   **Una observación importante:** La *aplicación de funciones* tiene **mayor precedencia** que cualquier operador in-fijo. Si escribimos `foo 3 n + 1 4` el compilador entenderá que queremos aplicar la función foo a 3 y a n para después sumar 1, el 4 quedaría de lado. Es decir, el compilador entendería: `(foo 3 n) + (1 4)`. Si lo que queremos es que n se sume

con 1 antes de que se aplique la función, debemos especificarlo utilizando paréntesis. Es decir, debemos escribir: `foo 3 (n + 1) 4`.

**Listas**

6. Las listas son uno de los tipos de dato más utilizados en Haskell. Las siguientes líneas de código ejemplifican algunas maneras de definirlas:

```
nums, range, range2 :: [Int]
nums = [1, 2, 3, 19]
range = [1..100]
range2 = [2, 4..100]
```

*Transcribe el código, ejecútalo y toma captura de tu trabajo.*
*Responde en un archivo de texto plano las siguientes preguntas observando el código que escribiste:*

- ¿Para qué sirven los dos puntos .. al definir una lista?
- Si quiero escribir una lista de Double's ¿Cómo especifico su tipo sintácticamente?

7. Para guardar palabras, oraciones y otras secuencias de caracteres útiles dentro de una variable, tenemos que especificar su tipo como `String`:

```
<identificador> ::  String
```

Se dice que el tipo `String` es azúcar sintáctica para el tipo `[Char]`. Esto quiere decir que ambos tipos son equivalentes pero `String` es más fácil de escribir. *Es más dulce con nosotros.*

```
*Main>cadenaUno = ['h','o','l','i']
*Main>cadenaDos = "holi"
*Main>cadenaUno == cadenaDos
True
```

['h','o','l','i'] equivale a "holi".

*Escribe cuatro cadenas (listas de caracteres) utilizando la sintaxis de los corchetes `[]` para dos de ellas y la sintaxis de las comillas para las otras dos.*

8. Podemos construir listas de otros tipos de datos además de `Char`, siempre que todos los elementos de nuestra lista sean del mismo tipo. Para construir listas utilizamos el operador *cons*, que consiste en dos puntos ":", veamos:

```
lista = 1:[]
```

Esto es exactamente lo mismo que escribir:

```
lista = [1]
```

*Escribe tres listas de mas de tres elementos utilizando el operador* `cons` *y el tipo de dato que gustes.*

9. También se pueden escribir listas sin el operador ":"pero esto es sólo azúcar sintáctica. El operador : estará ahí aunque no aparezca de forma explícita. Veamos un ejemplo:

   `miLista :: [Int] miLista = [1,2,3,4,5]`

   Lo anterior podría haberse definido de manera equivalente pero con el operador ":".

   `miLista :: [Int] miLista = 1:2:3:4:5:[]`

   Tengamos siempre presente que estamos trabajando con listas y no con arreglos. Observa cómo toda lista esta construida sobre la lista vacía [].

   *Escribe la versión de las listas definidas en el ejercicio anterior pero ahora sin utilizar el operador cons (los dos puntitos ":"). Utiliza identificadores distintos para que no haya error al compilar.*

10. La *recursión* es una manera de especificar procesos. Estamos familiarizados con formas *iterativas* de especificar procesos utilizando estructuras de control como *for* o *while*. La forma recursiva de hacer las cosas prescinde de esas estructuras y en su lugar aplica la función que estamos definiendo de forma autoreferencial con una previa modificación a los datos de entrada. La siguiente función es un ejemplo de recursión. ¿Puedes describir su funcionamiento?

```
1 secuencia :: Integer -> [Integer]
2 secuencia 1 = [1]
3 secuencia n = n : secuencia(n - 1)
```

   *Transcribe la función anterior y ejecútala con tres valores distintos, guarda una captura de pantalla con tus ejecuciones. Además escribe en el archivo de texto plano el procedimiento que sigue la función si se aplica al 5.*

11. Ahora escribamos una función que regrese la longitud de una lista dada :

```
1 longitudLista :: [Integer] -> Integer
2 longitudLista []     = 0
3 longitudLista (x:xs) = 1 + longitudLista xs
```

   Hagamos algunas obresrvaciones:

   - La primer línea de código es la firma de tipo de la función. Nos dice de qué tipo es su argumento y de qué tipo será su salida. En este caso el argumento será una lista de enteros y la salida un entero.
   - La segunda línea es un **caso base**. Aquí definimos lo que sucederá con los datos más sencillos que pudiéramos recibir como entrada. En este caso es la lista vacía.

- La tercer línea es el **caso recursivo**, en el que llamamos a la función misma pero con un dato de entrada más sencillo.
- `(x:xs)` es un patrón sintáctico. Va entre paréntesis para que el interprete lo tome como una sola entidad. Dentro de los paréntesis especificamos la estructura del dato de entrada. Como en esta ocasión hablamos de listas, para especificar su estructura utilizamos al operador `cons` junto a x, que representa la cabeza de la lista y `xs` que representa la cola de la lista.

*Transcribe la función anterior y ejecútala con tres listas distintas de longitud 0, 3 y 100. No olvides tomar captura de pantalla a tu trabajo.*

12. *Escribe una función que reciba una lista de enteros y los sume de dos en dos.* Por ejemplo:

```
*Main>sumaCadaDos[]
[]
*Main>sumaCadaDos [5]
[5]
*Main>sumaCadaDos [1,2,3,4]
[3, 7]
*Main>sumaCadaDos [1,2,3,4,5,6,7]
[3,7,11,7]
```

Pista: Podemos tener patrones anidados ¿Cómo acceder a los dos primeros elementos de una lista? Basta con escribir un patrón en el que tengamos la cabeza de la lista y la cabeza de la cola de la lista: `(x:(y:xs))` o bien `(x:y:xs)`

## Entrega

La entrega es **individual** y consistirá en un archivo comprimido que contenga:

- El archivo `Practica1.hs` con el código indicado en los ejercicios.
- Las capturas de pantalla indicadas en los ejercicios.
- Un archivo readMe.txt que incluya tu nombre, las respuestas a los ejercicios que lo requieran, así como algún comentario o sugerencia sobre la práctica.

Sube el archivo comprimido a la plataforma Google Classroom utilizando el siguiente formato para el nombre:

*Practica02_ Apellido1-Apellido2-Nombre(s).zip*

Por ejemplo, si Ada Lovelace Turing entregase su práctica le pondría como nombre a su archivo comprimido:

*Practica02_ Lovelace-Turing-Ada.zip*

Cualquier duda que tengas no dudes en enviarme un correo. 😄