# UT1 Repartido de ejercicio

El siguiente texto incluye ejercicios de programación en Haskell. Corresponde a la unidad temática UT1 *Lenguaje funcional básico y recursividad* del curso de *"Programación Funcional"* dictado por Ignacio Pacheco y Leonardo Val.

La realización de estos ejercicios es opcional, y no forma parte de ninguna forma de la calificación final del curso. Los docentes se reservan el derecho de corregir, modificar y ampliar el siguiente documento según crean conveniente.

### Funciones simples y numéricas

#### `nextMonth` & `previousMonth`

Se define el tipo `Month` como un número entero entre 0 y 11 (inclusive), dónde 0 se entiende como *Enero*, 1 como *Febrero*, 2 como *Marzo*, y así sucesivamente hasta el 11 para *Diciembre*. Se pide definir las funciones `nextMonth` y `previousMonth`, las cuales toman un mes y retornan el siguiente o el anterior respectivamente.

- `nextMonth 1 = 2`
- `previousMonth 1 = 0`
- `nextMonth 11 = 0`
- `previousMonth 0 = 11`

*Variantes*:

1. Agregar un argumento que indique cuantos meses hacia adelante o atrás (respectivamente) se quiere ir. Las funciones como están pedidas serían equivalente a tener este argumento extra en 1.
2. Permitir que el argumento anterior sea negativo.

#### `rockPaperScissors`

Se definen los posibles movimientos del juego *piedra, papel o tijera* (*rock, paper, scissors* en inglés) con un número entero, donde 0 es *piedra*, 1 es *papel* y 2 es *tijeras*. Se pide definir la función `rockPaperScissors`, que toma dos acciones y retorna 1 si gana el primer jugador, -1 si gana el segundo jugador o 0 si es un empate.

- `rockPaperScissors 1 1 = 0`
- `rockPaperScissors 1 2 = -1`
- `rockPaperScissors 1 0 = 1`
- `rockPaperScissors 0 1 = -1`
- `rockPaperScissors 2 2 = 0`

#### UK lengths

El sistema imperial británico de medidas de longitud tiene tres unidades llamadas *pulgadas* (*inches* en inglés), *pies* (*feet*) y *yardas* (*yards*). Definir las funciones de conversión de longitudes medidas en pulgadas y yardas a pies: `fromInchesToFeet` y `fromYardsToFeet`.

- `fromInchestoFeet 12.0 = 1.0`
- `fromYardstoFeet 1.0 = 3.0`
- `fromYardstoFeet 0.0 = 0.0`

#### `inRange`

Definir la función inRange que determina si un número entero está dentro de un rango dado por otros dos números enteros (de manera inclusiva).

- `inRange 5 7 1 = False`
- `inRange 5 7 6 = True`
- `inRange 5 7 8 = False`
- `inRange 5 7 5 = True`
- `inRange 7 5 6 = False`
- `inRange (-1) 0 2 = False`
- `inRange (-1) 1 0 = True`

*Variantes*:

1. Usar números en punto flotante en lugar de enteros.

#### `fromDirChar`

Definir la función `fromDirChar` que toma un caracter y retorna 1 si es >, -1 si es <, y 0 en otro caso.

- `fromDirChar '>' = 1`
- `fromDirChar '<' = -1`
- `fromDirChar 'a' = 0`
- `fromDirChar '9' = 0`
- `fromDirChar '.' = 0`

### Funciones recursivas y de listas

#### `isSorted`

Definir la función `isSorted`, que toma una lista de números enteros, retornando `True` si ésta está ordenada, o `False` sino.

- `isSorted [1,2,3] = True`
- `isSorted [1,3,2] = False`
- `isSorted [2,3] = True`
- `isSorted [3,2,1] = False`
- `isSorted [] = True`

*Variantes*:

1. Agregar un argumento que indique si el orden que se debe controlar es el ascendente o el descendente.

**`fromDirStr`**

Definir la función `fromDirStr`, que acumula todos los movimientos de `fromDirChar` (ver ejercicio anterior) en un `String`.

- `fromDirStr ">>>" = 3`
- `fromDirStr "<<" = -2`
- `fromDirStr "" = 0`
- `fromDirStr "<<>>" = 0`
- `fromDirStr "<a>b>c>" = 2`
- `fromDirStr "abc" = 0`

*Variantes*:

1. Agregar un argumento con una posición inicial. Las funciones originales actuarían como si ese argumento fuese 0.

**`toDirStr`**

Definir la función `toDirStr`, que toma un entero y retorna un `String` que haría que `fromDirStr` (ver ejercicio anterior) retorne ese entero.

- `toDirStr 3 = ">>>"`
- `toDirStr (-2) = "<<"`
- `toDirStr 0 = ""`
- `toDirStr 7 = ">>>>>>>"`
- `toDirStr (-1) = "<"`

**`traceDirStr`**

Definir la función `traceDirStr`, similar a la función `fromDirStr` (ver ejercicio anterior). Retorna una lista de enteros intermedios, en lugar de únicamente el resultado final. Pueden definirse funciones auxiliares.

- `traceDirStr ">>>" = [1,2,3]`
- `traceDirStr "<<" = [-1,-2]`
- `traceDirStr "" = []`
- `traceDirStr "<<>>" = [-1,-2,-1,0]`
- `traceDirStr "<a>b>c>" = [-1,0,1,2]`
- `traceDirStr "abc" = []`

## Funciones inspiradas en la *Generala*

**`isGenerala`**

Definir la función `isGenerala` que decide si los valores de 5 dados forman generala; i.e. son todos los valores iguales. Los 5 valores se especifican en una lista de enteros del 1 al 6. La lista debe tener 5 valores del 1 al 6.

- `isGenerala [1,2,3,4,5] = False`
- `isGenerala [] = False`
- `isGenerala [2,2,2,2,2] = True`
- `isGenerala [4,4,1,4,4] = False`
- `isGenerala [7,7,7,7,7] = False`
- `isGenerala [3,3,3,3] = False`
- `isGenerala [1,1,1,1,1,1] = False`
- `isGenerala [2,2,2,2,2] = True`

**`isPoker`**

Definir la función `isPoker` que decide si los valores de 5 dados forman póker; i.e. hay cuatro valores iguales y uno diferente. Los 5 valores se especifican en una lista de enteros del 1 al 6. La lista debe tener 5 valores del 1 al 6.

- `isPoker [1,2,3,4,5] = False`
- `isPoker [] = False`
- `isPoker [2,2,2,2,5] = True`
- `isPoker [4,4,4,4,4] = False`
- `isPoker [1,7,7,7,7] = False`
- `isPoker [3,3,3,3] = False`
- `isPoker [1,2,1,2,1,1] = False`
- `isPoker [1,2,1,1,1] = True`