

Funcional 3: Recursividad

Nota previa

Algo de recursividad... La recursividad en funcional consiste en definir una función en términos de sí mismo.

Si queremos calcular la sumatoria de una lista recursivamente vamos a tener un caso base y un caso recursivo definiéndola así:

```
suma [] = 0      -- Caso Base  
suma (x:xs) = x + suma xs  -- Caso Recursivo
```

Ahora sí, los ejercicios

1.1. Armar la función que me devuelve el n-ésimo número de la serie de Fibonacci, que se arma así:

```
fib(1) = 1  
fib(2) = 1  
fib(n) = fib(n-1) + fib(n-2) si n > 2
```

Hacerlo con guardas, y también con pattern matching.

1.2. Armar la función **pertenece/2**, que dados una lista y un número vale True si el número está en la lista.

Ayuda: Acá hay que pensar bien el caso base y variantes del caso recursivo.

1.3. Armar una función **intersección/2** que dadas dos listas me devuelve la lista de los elementos que están en las dos.

1.4. Armar una función **transformadaLoca** que dada una lista de números, devuelva otra a partir de la original tal que

- a los elementos mayores a 19 los elimina
- a los menores a 19 pares les suma 2
- a los menores a 19 impares les suma 1

```
Main> transformadaLoca [8,15,22,9,101,13]  
[10,16,10,14]
```

Ayuda: Vale armar una función auxiliar, se la puede llamar p.ej. sumaLoca.

1.5. Definir recursivamente la función **productList/1**, que me dada una lista me devuelva la productoria de los elementos de la misma.

1.6. Definir recursivamente la función **maximo/1**, recibe una lista de números y devuelve el

máximo número de la lista.

1.7. Definir recursivamente la función **menoresA/2**, recibe un número *n* y una lista de números *l*, y devuelve la sublista de *l* de los menores a *n*. Más fácil con un ejemplo: se espera que:

```
Main> menoresA 20 [23,5,16,38,11,24]
[5,16,11]
```

1.8.1 Armar una función **promedios** que dada una lista de listas me devuelve la lista de los promedios de cada lista-elemento. P.ej.

```
Main> promedios [[8,6],[7,9,4],[6,2,4],[9,6]]
[7,6.67,4,7.5]
```

1.8.2. Armar una función **promediosSinAplazos/1** que dada una lista de listas me devuelve la lista de los promedios de cada lista-elemento, excluyendo los que sean menores a 4 que no se cuentan. P.ej.

```
Main> promedios [[8,6],[6,2,4]]
[7,5]
```

1.9. Definir la función **diferencias/1** recursivamente, que recibe una lista de números devuelve la diferencia, en valor absoluto, de cada uno con el siguiente, excepto el último que no tiene siguiente. P.ej. se espera que

```
Main> diferencias [5,8,3,1,9]
[3,5,2,8].
```

1.10. Definir la función **sinRepetidos/1**, que me devuelve ... exactamente eso. P.ej. se espera que

```
Main> sinRepetidos [1,2,1,3,1,4,2,5,3]
[1,2,3,4,5]
```

Nota: No importa en qué orden los devuelva.

1.11.1 Definir la función **alVesre**, que recibe una lista y devuelve otra con los mismos elementos pero en el orden inverso. P.ej. se espera que

```
Main> alVesre [1,3,5,7,3,4]
[4,3,7,5,3,1].
```

1.11.2 Pensar primero y probar después la respuesta que va a dar Haskell ante estas consultas
alVesre ["la", "tarde", "se", "puso", "linda"]
alVesre ["la tarde se puso linda"]

1.12.1 Definir la función **sinExtremos**, que recibe una lista de números y devuelve lo que resulta

de sacar los números máximo y el mínimo. P.ej. se espera que

```
Main> sinExtremos [38,3,86,341,29,42,35,9]
[38,86,29,42,35,9].
```

Nota: Si la lista tiene menos de dos elementos, sinExtremos debe devolver la lista vacía.

1.12.2. Definir la función **sinPuntas/2**, que generaliza sinExtremos diciendo cuántos quiero sacar de cada punta. P.ej. se espera que

```
Main> sinPuntas 2 [38,3,86,341,29,42,35,9]
[38,29,42,35].
```

¿Cómo se puede definir sinExtremos usando sinPuntas?

1.12.3. Definir la función **sonTodosIguales/1**, que dada una lista devuelve True si todos sus elementos son iguales.

Dar tres definiciones

usando sinRepetidos

usando dispersion (esta serviría sólo para números) (Buscar referencia en guía anterior)

sin usar nada, en forma recursiva.

1.13.1 Implementar la función **esPar/1**, que dado un número natural (entero ≥ 0) indique si es par o no, usando únicamente sumas y restas (o sea, no vale hacer mod, ni usar modr).

Hacerlo de dos formas:

confiando en una función auxiliar **esParDesde/2** de forma tal que quede

esPar n = esParDesde n 0

sin necesitar función auxiliar ni parámetro adicional.

1.13.2 Extender alguna de estas implementaciones para que cubra también los enteros negativos.

1.14.1 Implementar la función **divr/2**, que es la misma función que div (parte entera de la división, p.ej. div 14 3 es 4), de forma tal que se usen sólo sumas y restas.

1.14.2. Idem para **mod**: implementar modr/2.

1.14.3. Definir la función **devolverParDivMod/2** que devuelve el par, el primer elemento de la tupla el divr/2 y el segundo el modr/2.

Por. Ej:

```
Main> devolverParDivMod 14 3
(4,2)
```

1.15.1 Definir la función **primo/1** que indica si un número es primo o no.

1.15.2. Definir la función **siguientePrimo/1** que devuelve el primer número primo mayor (estricto) al parámetro.

P.ej.

```
Main> siguientePrimo 3
5
Main> siguientePrimo 4  también devuelve 5
```

1.15.3. (difícil) Definir la función **factoresPrimos/1** que devuelve la lista de factores primos de un número, con su multiplicidad y en orden ascendente. P.ej.

```
Main> factoresPrimos 720
[2,2,2,2,3,3,5].
```

Consejo: puede venir bien una función auxiliar.

1.16.1. Definir la función **mcd/2**, que recibe dos números y devuelve su máximo común divisor. Se puede usar el algoritmo de Euclides, que se describe a continuación:

```
r resto de a entre b (dar a r el valor del resto de a por b)
si r es 0, entonces el mcd entre a y b es b
si r es distinto de 0 entonces el mcd entre a y b es el mismo que el mcd entre b y r.
```

P.ej. si $a = 12$ y $b = 4$, el mcd es 4, porque el resto de dividir a por b es 0.

Si $a = 30$ y $b = 9$, el mcd entre a y b es el mismo que hay entre 9 y 3, o sea entre b y el resto de dividir a por b.

Más info en http://es.wikipedia.org/wiki/Algoritmo_de_Euclides.

1.16.2. Definir la función **mcm'/2** (sí, vale poner apóstrofe, es como decir "mcm prima") que devuelva el mínimo común múltiplo entre dos números calculándolo así:

```
si el mayor divide al menor, ya está, el mcm es el mayor.
si no pruebo con el mayor * 2.
si no pruebo con el mayor * 3.
... y así, en algún momento va a parar (¿por qué?).
```

P.ej. quiero calcular el mcm entre 10 y 6

10 no divide a 6

20 no divide a 6

30 sí divide a 6 ¡voilà! el mcm entre 10 y 6 es 30.

1.16.3. (muy difícil) Definir la función **mcm''/2** (léase "mcm segunda") que devuelva el mínimo común múltiplo entre dos números calculándolo como el producto de los factores primos a la máxima potencia.

Consejo: Usar el **factoresPrimos** que fue definido en el ejercicio anterior y definir una función que a partir de las dos listas devuelva la lista con cada factor a su máxima potencia.

1.17.1. Definir la función **obtenerPalabras/1**, dada una lista de Strings devuelve la sublista de aquellos que representen una sola palabra (i.e. que no tengan espacios). Ej si uso la frase de Saint Exupéry.

```
Main> obtenerPalabras ["Lo", "esencial", "es", "invisible", "a los ojos"]
["Lo", "esencial", "es", "invisible"]
```

1.17.2. Definir la función **palabrasLargas/1**, recibe una lista de Strings, y devuelve la sublista de aquellos que representen una sola palabra (i.e. que no tengan espacios) y que tengan al menos 7 letras.

```
Main> palabrasLargas["Lo", "esencial", "es", "invisible", "a los ojos"]  
["esencial", "invisible"]
```

1.17.3. Definir la función **longitudesDeNombres/1**, recibe una lista de Strings, y devuelve la lista de las longitudes de aquellas que representan nombres, i.e., que empiecen con una mayúscula.

1.17.4. Definir la función **esVocalosa/1**, recibe un String y devuelve True si representa una palabra (ya definido) y si tiene más vocales que no-vocales. Vale considerar solamente palabras en minúscula.

```
Main> esVocalosa "sabio"  
True
```

1.18.1 Implementar una función **cifrasBinarias/1** que devuelva la cantidad de cifras binarias que necesito para escribir un número. P.ej.

```
Main> cifrasBinarias 15  
4  
Main> cifrasBinarias 16  
5
```

Pensarlo en forma recursiva.

1.18.2. Extender la implementación a cualquier base (binario = base 2), armando la función **cifrasBase/2** que recibe la base y el número, de forma tal que la función **cifrasBinarias/1** se pueda redefinir así:

`cifrasBinarias n = cifrasBase 2 n`

1.18.3. Definir la función **convertirABase/2**, que dado un número decimal y una base, devuelva el número convertido a la base.

```
Main> convertirABase 77 2  
1001101
```

```
Main> convertirABase 159 8  
237
```

Nota: Utilizar la función **cifrasBase/2**.

1.18.4. Definir la función, **convertirADecimal/2**, que dado un número y la base n en el que esta convertirlo a base 10 (a decimal).

```
Main> convertirADecimal 1001101 2  
77
```

1.19. Implementar la función **nroCapicua/1** que indica si un número es o no capicúa.

1.20. Volviendo al ejercicio de las lluvias de un determinada mes. De donde conocemos la siguiente información:

lluviasEnero = [0,2,5,1,34,2,0,21,0,0,0,5,9,18,4,0]

1.20.1. (difícil) Definir la función **diasLluviosos/1**, que devuelve los números de día en los que se registraron lluvias. P.ej. se espera que la consulta

```
Main> diasLluviosos lluviasEnero  
[2,3,4,5,6,8,12,13,14,15].
```

Ayuda: hay que aparear lo que llovió cada día con el número de día correspondiente, la lista de los días puede obtenerse así:

```
[1..length lluvias]
```

1.21. Se representa la información sobre ingresos y egresos de una persona en cada mes de un año mediante dos listas, de 12 elementos cada una. P.ej., si entre enero y junio gané 100, y entre julio y diciembre gané 120, mi lista de ingresos es

```
[100,100,100,100,100,100,120,120,120,120,120,120]
```

y si empecé en 100 y fui aumentando de a 20 por mes, llegando a 220, queda

```
[100,120..220]
```

(jugar un poco con esta notación)

Definir las funciones:

1.21.1. resultados, que dadas las listas de ingresos y egresos devuelve la lista de los resultados de cada mes.

Nota: Utilizar Listas por Compresión.

1.21.2. resultado, que dadas las listas de ingresos y egresos y un mes, devuelve el resultado del mes.

Nota: Utilizar la función resultados.

1.22. Definir la función **estaOrdenada/1**, que recibe una lista de números e indica si está ordenada de menor a mayor o no.

Ayuda: Esta les conviene pensarla en forma recursiva. Sin recursividad explícita sale ... pero no es para nada fácil, en las dos formas que nos salieron usamos fold (en una foldl y en otra foldr), en una también necesitamos zip.

1.23. Imaginemos un tablero con la idea p.ej. del juego de la oca, casilleros numerados desde el 1 hasta (supongamos) el 1000.

Tengo una ficha que está en un casillero, decimos que este es el casillero inicial.

Quiero saber cuándo va a caer en un casillero múltiplo de m (meta) haciendo saltos de a s casilleros (s por salto).

P.ej. si inicial = 3, meta = 6, salto = 7, entonces el resultado buscado es 24, porque la ficha partiendo del casillero 3 y saltando de a 7 va a hacer este recorrido: 3-10-17-24, en el 24 para porque es múltiplo de 6.

Implementar una función `dondePara/3` que recibe los valores de inicial, meta y salto, y devuelve el resultado buscado.

Suponer que si la ficha llega al casillero 1000 o se pasa, entonces el resultado es 1000. P.ej. si inicial = 3, meta = 6, salto = 2, entonces nunca va a llegar a un casillero como los que quiero, porque va a ir siempre por casilleros impares. En este caso que la función devuelva 1000

1.24. (difícil) Nota previa:

En este ejercicio no usar representación de un número como String, hacer lo que se pide usando sólo operaciones sobre números enteros.

1.24.1. Implementar la función **`primerCifra/1`** que devuelve la primer cifra de un número entero positivo. P.ej. se espera que

```
Main> primerCifra 416285
4.
```

1.24.2. Implementar la función **`sinPrimerCifra/1`** que recibe un número y lo devuelve sin su primer cifra. P.ej. se espera que

```
Main> sinPrimerCifra 416285
16285.
```

Ayuda: usar el operador de exponenciación $^$ (probar con la consulta 10^3).

1.25. Representamos matrices como una lista de listas, donde cada lista es una fila. P.ej. a la matriz

```
1 2 3
2 3 4
3 4 5
```

la representaremos así

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

Definir las siguientes funciones

`filas mat / columnas mat / dimension mat`: indican la cantidad de filas y de columnas y la dimensión (o sea el par (cant filas, cant columnas)) de una matriz.

`esCuadrada mat`: funcion booleana que indica si una matriz es cuadrada.

`valor mat i j`: devuelve el valor en la posición (i,j) de la matriz

fila mat i: devuelve una lista con la fila i-ésima de la matriz
 columna mat j: devuelve una lista con la columna j-ésima de la matriz
 diagonalPrincipal mat: devuelve una lista con los valores de la diagonal principal de la matriz.
 parteSuperior mat / parteInferior mat: devuelven listas con los valores arriba / abajo de la diagonal principal
 esIdentidad mat: función booleana que indica si una matriz es identidad.

P.ej.

```

filas [[1,2,3],[2,3,4]]      -- devuelve 2
columnas [[1,2,3],[2,3,4]]   -- devuelve 3
dimension [[1,2,3],[2,3,4]]  -- devuelve (2,3)
esCuadrada [[1,2,3],[2,3,4]] -- devuelve False
valor [[1,2,3],[7,3,4]] 2 1  -- devuelve 7
fila [[1,2,3],[7,3,4]] 2    -- devuelve [7,3,4]
columna [[1,2,9],[7,3,4]] 3  -- devuelve [9,4]
diagonalPrincipal [[1,2,3],[4,5,6],[7,8,9]] -- devuelve [1,5,9]
parteSuperior [[1,2,3],[4,5,6],[7,8,9]]    -- devuelve [2,3,6]
parteInferior [[1,2,3],[4,5,6],[7,8,9]]     -- devuelve [4,7,8]
  
```

En el caso de parteSuperior y parteInferior no importa el orden.

Ayuda: en varios casos conviene pensar en la definición y aplicar listas por comprensión.

1.26.1 Triángulo de Pascal (o de Tartaglia)

Definir la función filaPascal, que dado un número n devuelve la lista con la n-ésima fila del triángulo de Pascal. P.ej.

```

Main> filaPascal 5
[1,4,6,4,1]
  
```

Hacerlo recursivo, el caso base es

filaPascal 1 = [1]

y para el caso recursivo pensar qué hay que hacer para obtener la fila n+1 a partir de la fila n: es la suma del 1ro + el 2do ... etc ... (usar una función auxiliar) con un uno a algún costado (elegir cuál).

1.26.2. Definir la función trianguloPascal, que dado un n devuelve la lista de las filas del triángulo hasta la n-ésima inclusive. P.ej.

```

trianguloPascal 5 -- devuelve [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
  
```

Hacerlo con un map. Ayuda: ¿sobre qué lista se hace el map?

1.27.1. Ordenamiento

Definir la función **ordenar**/1, que recibe una lista y devuelve otra con los mismos elementos de la anterior pero ordenados de menor a mayor. Usar esta idea: para ordenar una lista tomo el mínimo, lo pongo al principio, después agarro el mínimo del resto... y así hasta que ordené todos.

Conviene definir una función auxiliar **sacarUno**/2, que recibe un elemento y una lista, y devuelve la lista sin la primer aparición de ese elemento.

1.27.2. (difícil) Definir la función **quickSort**/1, que recibe una lista y devuelve otra con los mismos elementos de la anterior pero ordenados de menor a mayor. Usar la idea de quickSort: tomo el primer elemento, llamémoslo x

separo el resto en los menores a x y los mayores a x
la lista ordenada será: resultado de ordenar los menores más x más resultado de ordenar los mayores.

1.27.3. (muy difícil) Definir la función **quickSortSegun/2**, que recibe una función y una lista, y devuelve una lista con los mismos elementos de la original ordenados de menor a mayor, donde el criterio de comparación es la función.

P.ej. se espera que

```
Main> quickSortSegun ultimaCifra [15,28,33,41,56,74]  
[41,33,74,15,56,28].
```