Introducción a los Algoritmos - 1er. cuatrimestre 2025

Guía 2: Funciones, listas, recursión e inducción

El objetivo de los siguientes ejercicios es continuar el aprendizaje de la ***programación funcional****, es decir, al desarrollo de programas como funciones. Veremos en estos ejercicios cómo trabajar con listas. Para familiarizarnos con ellas, los primeros ejercicios son de evaluación y tipado. Los siguientes serán sobre definición de funciones recursivas y no recursivas, y finalmente, aprenderemos cómo hacer demostraciones por inducción.*

Listas

Ahora, comenzaremos a complejizar el lenguaje de nuestras ***expresiones*** *agregando* ***listas****. Una lista (o secuencia) es una colección ordenada de valores, que deben ser todos del mismo tipo; por ejemplo,* `[1,2,5]`.

Denotamos a la lista vacía con `[]`*. El operador* `:` *(llamado "pegar") es fundamental (se lo denomina constructor) ya que permite construir listas arbitrarias a partir de la lista vacía.* `:` *toma un elemento x (a izquierda) y una lista* `xs` *y devuelve una lista con primer elemento* `x` *seguido de los elementos de* `xs`*. Por ejemplo* `3:[ ] = [3]`*, y* `1:[2,3] = [1,2,3]`*. Para denotar listas no vacías utilizamos expresiones de la forma* `[x,y,...,z]`*, que son abreviaciones de* `x:(y:...:(z:[ ]))`*. Como el operador* `:` *es asociativo a derecha, es lo mismo escribir* `x:(y:...:(z:[ ]))` *que* `x:y:...:z:[ ]`*. Otros operadores sobre listas son los siguientes:*

- `length`*, longitud, toma una lista* `xs` *y devuelve su cantidad de elementos. Ej:* `length [1,2,0,5] = 4`.
- `!!` *toma una lista* `xs` *(a izquierda) y un natural* `n` *que indica una posición, y devuelve el elemento de la lista que se encuentra en la posición* `n` *(contando a partir de la posición 0). Ej:* `[1,3,3,6,2,3,4,5] !! 4 = 2`*. Este operador, llamado índice, asocia a izquierda, por lo tanto* `xs !! n !! m` *se interpreta como* `(xs !! n) !! m`.
- `take` *(tomar) dada un natural* `n` *y una lista* `xs` *devuelve la sublista con los primeros* `n` *elementos de* `xs`*. Ej:* `take 2 [1,2,3,4,5,6] = [1,2]`.
- `drop` *(tirar) dada un natural* `n` *y una lista* `xs` *devuelve la sublista sin los primeros* `n` *elementos de* `xs`*. Ej:* `drop 2 [1,2,3,4,5,6] = [3,4,5,6]`.
- `++` *toma dos listas* `xs` *(a izquierda) e* `ys` *(a derecha), y devuelve la lista con todos los elementos de* `xs` *seguidos de los elementos de* `ys`*. Ej:* `[1,2,4] ++ [1,0,7] = [1,2,4,1,0,7]`*. Este operador, llamado concatenación, es asociativo por lo que podemos escribir sin ambigüedad expresiones sin paréntesis, como* `xs ++ ys ++ zs`.

Existen además dos funciones fundamentales sobre listas que listamos a continuación.

- `head`*, llamada cabeza, toma una lista xs y devuelve su primer elemento. Ej:* `head [1,2,3] = 1`.
- `tail`*, llamada cola, toma una lista xs y devuelve la sublista que resulta de eliminar el primer elemento. Ej:* `tail [1,2,3] = [2,3]`.

La aplicación de función asocia a izquierda, por lo tanto en general es necesario utilizar paréntesis para que la expresión quede bien tipada. Si se quiere escribir la expresión `tail (tail xs)` *no se pueden eliminar los paréntesis, puesto que* `tail tail xs` *(que se interpreta como* `(tail tail) xs`*) no tiene sentido.*

El objetivo de los siguientes ejercicios es familiarizarse con el tipo de listas y extender el método para justificar el tipado de expresiones, considerando expresiones más complejas que las que veníamos trabajando.

1. Utilizá las definiciones intuitivas de los operadores de listas para evaluar las siguientes expresiones. Subrayá la subexpresión resuelta en cada paso justificado. Luego usá un intérprete de `haskell` para verificar los resultados. Por ejemplo:

```
    [23,45,6] !! (head [1,2,3,10,34])
= { def. de head }
    [23,45,6] !! 1
= { def. de !! }
    45
```

a) `length [5,6,7]`

b) `[5,3,57] !! 1`

c) `[0,11,2,5]:[ ]`

d) `take 2 [5,6,7]`

e) `drop 2 [5,6,7]`

f) `head (0:[1,2,3])`

g) `([1,2] ++ [3,4]) ++ [(2+3)]`

h) `take 2 (([[1]] ++ [[2]]) ++ [[3]])`

i) `take (length ([ ]:[[ ]])) (([[ ]] ++ [[ ]]) ++ [([ ] ++ [ ])])`

Teniendo en cuenta la definición intuitiva de los operadores de listas de la introducción a esta sección, escribí el tipo de cada uno de ellos. Por ejemplo, el operador `head` *toma una lista y devuelve el primer elemento de ella. La lista puede contener elementos de cualquier tipo (todos del mismo), ya sean números, valores booleanos, otras listas, etc. Para denotar esta situación utilizamos variables "de tipo" (como* `a`, `b` `c`, *... etc). Entonces podemos decir que el operador* `head` *toma una lista de tipo* `[a]`*, donde la variable a representa cualquier tipo (*`Int`, `Bool`, `[Int]`, *...) y devuelve un elemento de esa lista, por lo tanto debe ser un elemento de tipo* `a`*. Esto lo escribimos en* notación funcional *(izq.) o en* notación de árbol *(der.):*

```
head :: [a] -> a
```

```
head [a]
--------
   a
```

2. Decidí si las siguientes expresiones están bien escritas, agregando paréntesis para hacer explícita la precedencia y la asociatividad. Usá un intérprete de `haskell` para verificar los resultados.

a) `-45:[1,2,3]`

b) `([1,2] ++ [3,4]) ++ [5]`

c) `0 ++ [[1,2,3]]`

d) `[ ]:[ ]`

e) `([1] ++ [2]) ++ [[3]]`

f) `[1,5,False ]`

g) `head [[5]]`

h) `head [True,False ] ++ [False ]`

Funciones recursivas

Una ***función recursiva*** *es una función tal que en su definición puede aparecer su propio nombre. Una buena pregunta sería ¿Cómo lograr que no sea una definición circular? La clave está en el principio de inducción: en primer lugar hay que definir la función para el (los) caso(s) más "pequeño(s)", que llamaremos* ***caso base*** *y luego definir el caso general en términos de algo más "chico", que llamaremos* ***caso inductivo****. En el caso base no debe aparecer el nombre de la función que se está definiendo. El caso inductivo es donde aparece el nombre de la función que se está definiendo, y debe garantizarse que el (los) argumento(s) al cual se aplica en la definición es más "chico" (para alguna definición de más chico) que el valor para la que se está definiendo.*

3. Una función de **filter** es aquella que dada una lista devuelve otra lista cuyos elementos son los elementos de la primera que cumplan una determinada condición, en el mismo orden y con las mismas repeticiones (si las hubiere). Por ejemplo: `soloPares :: [Int] -> [Int]` devuelve aquellos elementos de la lista que son pares.

   Definí recursivamente las siguientes funciones filter.

   a) `soloPares :: [Int] -> [Int]`, que dada una lista de enteros `xs` devuelve una lista sólo con los números pares contenidos en `xs`, en el mismo orden y con las mismas repeticiones (si las hubiera).
   Por ejemplo: `soloPares [3,0,-2,12] = [0,-2, 12]`

   b) `mayoresQue10 :: [Int] -> [Int]`, que dada una lista de enteros `xs` devuelve una lista sólo con los números mayores que 10 contenidos en `xs`,
   Por ejemplo: `mayoresQue10 [3,0,-2, 12] = [12]`

c) `mayoresQue :: Int-> [Int] -> [Int]`, que dado un entero `n` y una lista de enteros `xs` devuelve una lista sólo con los números mayores que `n` contenidos en `xs`,
Por ejemplo: `mayoresQue 2 [3,0,-2, 12] = [3,12]`

Preguntas:

a) ¿Se te ocurre algún otro ejemplo de una función de este tipo?

4. Una función de **map** es aquella que dada una lista devuelve otra lista cuyos elementos son los que se obtienen de aplicar una función a cada elemento de la primera en el mismo orden y con las mismas repeticiones (si las hubiere). Por ejemplo: `duplica :: [Int] -> [Int]` devuelve cada elemento de la lista multiplicado por 2.

   Definí recursivamente las siguientes funciones de map.

   a) `sumar1 :: [Int] -> [Int]`, que dada una lista de enteros le suma uno a cada uno de sus elementos.
   Por ejemplo: `sumar1 [3,0,-2] = [4,1,-1]`

   b) `duplica :: [Int] -> [Int]`, que dada una lista de enteros duplica cada uno de sus elementos.
   Por ejemplo: `duplica [3,0,-2] = [6,0,-4]`

   c) `multiplica :: Int-> [Int] -> [Int]`, que dado un número `n` y una lista, multiplica cada uno de los elementos por `n`.
   Por ejemplo: `multiplica 3 [3,0,-2] = [9,0,-6]`

   Preguntas:

   a) ¿Se te ocurre algún otro ejemplo de una función de este tipo?

5. Una función de **fold** es aquella que dada una lista devuelve un valor resultante de combinar los elementos de la lista. Por ejemplo: `sum: [Int] -> Int` devuelve la sumatoria de los elementos de la lista.

   Definí recursivamente las siguientes funciones de tipo fold.

   a) `todosMenores10 :: [Int] -> Bool`, que dada una lista devuelve `True` si ésta consiste sólo de números menores que 10.
   Por ejemplo: `todosMenores10 [1,3,9] = True`

   b) `hay0 :: [Int] -> Bool`, que dada una lista decide si existe algún 0 en ella.
   Por ejemplo: `hay0 [1,0,3] = True`

   c) `sum :: [Int] -> Int`, que dada una lista devuelve la suma de todos sus elementos.
   Por ejemplo: `suma [1,2,3] = 6`

   Preguntas:

   a) ¿Se te ocurre algún otro ejemplo de una función de este tipo?

Más funciones recursivas

6. Una función de tipo **zip** es aquella que dadas dos listas devuelve una lista de pares donde el primer elemento de cada par se corresponde con la primera lista, y el segundo elemento de cada par se corresponde con la segunda lista. Por ejemplo: `repartir :: [String] -> [String] -> [(String,String)]` donde los elementos de la primera lista son nombres de personas y los de la segunda lista son cartas españolas es una función que devuelve una lista de pares que le asigna a cada persona una carta.

   Ej: `repartir ["Juan", "Maria"] ["1 de Copa", "3 de Oro", "7 de Espada", "2 de Basto"] = [("Juan","1 de Copa"), ("Maria","3 de Oro")]`

   Defina la función recursivamente.

7. Una función de tipo **unzip** es aquella que dada una lista de tuplas devuelve una lista de alguna de las proyecciones de la tupla. Por ejemplo, si tenemos una lista de ternas donde el primer elemento representa el nombre de un alumno, el segundo el apellido, y el tercero la edad, la función que devuelve la lista de todos los apellidos de los alumnos en una de tipo unzip.

   Definir la función `apellidos :: [(String, String, Int)] -> [String]`.

   Ej: `apellidos [("Juan","Dominguez",22), ("Maria","Gutierrez",19), ("Damian","Perez",43)]`
   `["Dominguez","Gutierrez","Perez"]`

   Definí la función recursivamente.

8. Definí recursivamente los operadores básicos de listas: `length`, `!!`, `take`, `drop`, `++`. Para los operadores `take` y `drop` deberás hacer recursión en ambos parámetros, en el parámetro lista y en el parámetro numérico.

9. (i) Definí funciones por recursión para cada una de las siguientes descripciones. (ii) Evaluá los ejemplos manualmente (iii) Identificá si las funciones son de algún tipo ya conocido (filter, map, fold). (iv) Programálas en `haskell` y verificá los resultados obtenidos.

   a) `maximo :: [Int] -> Int`, que calcula el máximo elemento de una lista de enteros.
   Por ejemplo: `maximo [2,5,1,7,3] = 7`
   Ayuda: Ir tomando de a dos elementos de la lista y 'quedarse' con el mayor.

   b) `sumaPares :: [(Int, Int)] -> Int`, que dada una lista de pares de números, devuelve la sumatoria de todos los números de todos los pares.
   Por ejemplo: `sumaPares [(1,2)(7,8)(11,0)] = 29`

   c) `todos0y1 :: [Int] -> Bool`, que dada una lista devuelve `True` si ésta consiste sólo de 0s y 1s.
   Por ejemplo: `todos0y1 [1,0,1,2,0,1] = False`, `todos0y1 [1,0,1,0,0,1] = True`

   d) `quitar0s :: [Int] -> [Int]` que dada una lista de enteros devuelve la lista pero quitando todos los ceros.
   Por ejemplo `quitar0s [2,0,3,4] = [2,3,4]`

   e) `ultimo :: [a] -> a`, que devuelve el último elemento de una lista.
   Por ejemplo: `ultimo [10,5,3,1] = 1`

   f) `repetir :: Int-> Int-> [Int]`, que dado un número `n` mayor o igual a 0 y un número `k` arbitrario construye una lista donde `k` aparece repetido `n` veces.
   Por ejemplo: `repetir 3 6 = [6,6,6]`

   g) `concat :: [[a]] -> [a]` que toma una lista de listas y devuelve la concatenación de todas ellas.
   Por ejemplo: `concat [[1,4],[],[2]] = [1,4,2]`

   h) `rev :: [a] -> [a]` que toma una lista y devuelve una lista con los mismos elementos pero en orden inverso.
   Por ejemplo: `rev [1,2,3] = [3,2,1]`

Inducción

Una técnica poderosa para demostrar propiedades sobre un dominio inductivo, como son los naturales o las listas, es usar el ***principio de inducción****. La idea que rige este principio consiste en demostrar dos cosas. Por un lado verificar que la propiedad se satisface para los elementos "más chicos" del dominio (por ejemplo, la lista* `[ ]`*). Por otro lado demostrar para un elemento arbitrario del dominio (por ejemplo, la lista* `x:xs`*) que si suponemos que la propiedad es cierta para todos los elementos más chicos que él (por ejemplo* xs*), entonces la propiedad también es satisfecha por ese elemento. Dado que todo elemento de un dominio inductivo puede ser "construido" a partir de elementos más simples, este procedimiento demuestra que la propiedad es satisfecha por todos los elementos del dominio, y por lo tanto es válida.*

10. Considerando las definiciones de los ejercicios anteriores demostrá por inducción sobre `xs` las siguientes propiedades:

a) `sum (sumar1 xs)` = `sum xs + (length xs)`

b) `sum (duplica xs)` = `2 * sum xs`

c) `length (duplica xs)` = `length xs`

11. Demostrá por inducción las siguientes propiedades. **Ayuda:** Recordá la definición de cada uno de los operadores implicados en cada expresión.

    a) `xs ++ [ ]` = `xs` (la lista vacía es el elemento neutro de la concatenación)

    b) `length xs` $\geqslant 0$

12. Considerando la función `quitarCeros :: [Int] -> [Int]` definida de la siguiente manera

```
quitarCeros [] = []
quitarCeros (x:xs)
  | x /= 0 = x:quitarCeros xs
  | x == 0 = quitarCeros xs
```

    demostrá que

```
sum (quitarCeros xs) = sum xs
```

    Ayuda: Tené en cuenta que como la función `quitarCeros` se define por casos, el caso inductivo también deberá dividirse en dos casos.

13. Considerando la función `soloPares :: [Int] -> [Int]` definida de la siguiente manera

```
soloPares [] = []
soloPares (x:xs)
  | mod x 2 == 0 = x:(soloPares xs)
  | mod x 2 /= 0 = soloPares xs
```

    demostrá que

```
soloPares (xs ++ ys) = (soloPares xs) ++ (soloPares ys)
```

    Ayuda: Tené en cuenta que como la función `soloPares` se define por casos, el caso inductivo también deberá dividirse en dos casos.

14. Considerando la función `sum :: [Int] -> Int` que toma una lista de números y devuelve la suma de ellos, definí `sum` y demostrá que:

```
sum (xs ++ ys) = sum xs + sum ys
```

15. Considerando la función `repetir :: Int-> Int-> [Int]`, que construye una lista de un mismo número repetido cierta cantidad de veces, definida recursivamente como:

```
repetir 0 x = [ ]
repetir n+1 x = x:(repetir n x)
```

    demostrá que `length (repetir n x)` = n.

    Aclaración: el pattern matching `n+1` no funciona en `haskell` (al menos en `ghci`) por defecto, aunque puede habilitarse escribiendo {−#`LANGUAGENPlusKPatterns`#−} en el código. Esta notación es conveniente para hacer la demostración por inducción. Por otro lado, vale la pena notar que si bien el tipo de `n` es `Int`, la propiedad que vamos a demostrar vale solo para `n>0`.

16. Considerando la función `concat : [[a]] -> [a]` que toma una lista de listas y devuelve la concatenación de todas ellas, definida recursivamente como:

```
concat [ ] = [ ]
concat (xs :xss) = xs ++ (concat xss)
```

demostrá que

```
concat (xss ++ yss) = (concat xss) ++ (concat yss)
```

17. Considerando la función `rev :: [a] -> [a]` que toma una lista y devuelve una lista con los mismos elementos pero en orden inverso, definida recursivamente como:

```
rev [ ] = [ ]
rev (x :xs) = rev xs ++ [x]
```

demostrá que `rev (xs ++ ys) = (rev ys) ++ rev xs`

18. Demostrá por inducción las siguientes propiedades. **Ayuda:** Recordá la definición de cada uno de los operadores implicados en cada expresión.

    a) `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs` (la concatenación es asociativa)

    b) `take (length xs) (xs ++ ys) = xs`

    c) `drop (length xs) (xs ++ ys) = ys`

    d) `xs ++ (y:ys) = (xs ++ [y]) ++ ys`

19. Considerando las definiciones dadas en cada caso. Demuestre por inducción sobre `n` las siguientes propiedades:

    a) `f n = 2 * n`, donde

    ```
    f 0 = 0
    f (n+1) = 2 + (f n)
    ```

    b) `g n = n`, donde

    ```
    g 0 = 0
    g (n+1) = 1 + (g n)
    ```

    c) `fn = g n`, donde

    ```
    g 0 = 0
    g (n+1) = n + 1 + (g n)
    ```

    ```
    fn = n*(n+1) /2
    ```

Problemas más complejos

Los siguientes problemas son un poco más complejos que los que se vieron, especialmente porque se parecen más a los problemas a los que nos enfrentamos en la vida real. Se resuelven desarrollando programas funcionales; es decir, se pueden plantear como la búsqueda de un resultado a partir de ciertos datos (argumentos).

Para ello, será necesario en primer lugar descubrir los tipos de los argumentos y del resultado que necesitamos. Luego combinaremos la mayoría de las técnicas estudiadas hasta ahora: ***modularización*** *(dividir un problema complejo en varias tareas intermedias),* ***análisis por casos****, e identificar qué clase de funciones de lista (cuando corresponda) son las que necesitamos:* ***map****,* ***filter*** *o bien* ***fold****.*

20. (i) Definí funciones por recursión para cada una de las siguientes descripciones. (ii) Evaluá los ejemplos manualmente (iii) Identificá si las funciones son de algún tipo ya conocido (filter, map, fold). (iv) Programálas en `haskell` y verificá los resultados obtenidos.

a) `listasIguales :: [a] -> [a] -> Bool`, que determina si dos listas son iguales, es decir, contienen los mismos elementos en las mismas posiciones respectivamente.

Por ejemplo:
```
listasIguales [1,2,3] [1,2,3] = True,
listasIguales [1,2,3,4] [1,3,2,4] = False
```

b) `mejorNota :: [(String,Int,Int,Int)] -> [(String,Int)]`, que selecciona la nota más alta de cada alumno.

Por ejemplo:
```
mejorNota [("Matias",7,7,8),("Juan",10,6,9),("Lucas",2,10,10)] =
   [("Matias",8),("Juan",10),("Lucas",10)]
```

c) `incPrim :: [(Int, Int)] -> [(Int,Int)]`, que dada una lista de pares de enteros, le suma 1 al primer número de cada par.

Por ejemplo:
```
incPrim [(20,5),(50,9)] = [(21,5),(51,9)]
incPrim [(4,11),(3,0)] = [(5,11),(4,0)].
```

d) `expandir :: String-> String`, pone espacios entre cada letra de una palabra.

Por ejemplo: `expandir "hola" = "h o l a"` (¡sin espacio al final!).

21. **Películas**

Contamos con una base de datos de películas representada con una lista de tuplas. Cada tupla contiene la siguiente información:

(⟨Nombre de la película⟩, ⟨Año de estreno⟩, ⟨Duración de la película⟩, ⟨Nombre del director⟩)

Observamos entonces que el tipo de la tupla que representa cada película es `(String, Int, Int, String)`.

a) Definí la función `verTodas :: [(String, Int, Int, String)] -> Int` que dada una lista de películas devuelva el tiempo que tardaría en verlas a todas.

b) Definí la función `estrenos :: [(String, Int, Int, String)] -> [String]` que dada una lista de películas devuelva el listado de películas que estrenaron en 2023.

c) Definí la función `filmografia :: [(String, Int, Int, String)] -> String-> [String]` que dada una lista de películas y el nombre de un director, devuelva el listado de películas de ese director.

d) Definí la función `duracion :: [(String, Int, Int, String)] -> String-> Int` que dada una lista de películas y el nombre de una película, devuelva la duración de esa película.

22. **Ventas de PCs**

Una empresa de venta de computadoras está desarrollando un sistema para llevar registro de ventas. Para ello cuenta con la siguiente información:

Lista de los vendedores de la empresa

```
vendedores = ["Martin", "Diego", "Claudio", "José"]
```

Lista de ventas de la forma (fecha, nombre vendedor, lista de componentes de la máquina. La fecha es una tupla de la forma (día, mes, año) y los componentes son Strings.

```
ventas = [((1,2,2006), "Martin", ["Monitor GPRS 3000", "Motherboard ASUS 1500"]),
          ((1,2,2006), "Diego", ["Monitor ASC 543", "Motherboard Pindorcho"]),
          ((10,2,2006), "Martin", ["Monitor ASC 543", "Motherboard ASUS 1200"]),
          ((12,2,2006), "Diego", ["Monitor GPRS 3000", "Motherboard ASUS 1200"]),
          ((4,3,2006), "Diego", ["Monitor GPRS 3000", "Motherboard ASUS 1500"])]
```

Lista de precios de los componentes, de la forma (nombre componente, precio).

```
precios = [("Monitor GPRS 3000", 200), ("Motherboard ASUS 1500", 120), ("Monitor ASC 543",
250), ("Motherboard ASUS 1200", 100), ("Motherboard Pindorcho", 30)]
```

Se pide desarrollar las siguientes funciones

a) `precioMaquina`, recibe una lista de componentes, devuelve el precio de la máquina que se puede armar con esos componentes, que es la suma de los precios de cada componente incluido.

Por ejemplo: `precioMaquina ["Monitor GPRS 3000", "Motherboard ASUS 1500"] = 320`

($200 del monitor más $120 del motherboard)

b) `cantVentasComponente`, recibe un componente y devuelve la cantidad de veces que fue vendido, o sea que formó parte de una máquina que se vendió.

Por ejemplo: `cantVentasComponente "Monitor ASC 543" = 2`

La lista de ventas no se pasa por parámetro, se asume que está identificada por `ventas`.

c) `vendedorDelMes`, se le pasa un par que representa (mes,año) y devuelve el nombre del vendedor que más vendió en plata en el mes. O sea, no cantidad de ventas, sino importe total de las ventas. El importe de una venta es el que indica la función `precioMaquina`.

Por ejemplo: `vendedorDelMes (2,2006) = "Martin"`

(Vendió por $670, una máquina de $320 y otra de $350)

d) Obtener las ventas de un mes, de forma que:

`ventasMes (2,2006) = 1050`

e) Obtener las ventas totales realizadas por un vendedor sin límite de fecha, de forma que:

`ventasVendedor "Diego" = 900`

f) `huboVentas` que indica si hubo ventas en un mes y año determinados.

Por ejemplo: `huboVentas (1, 2006) = False`