Introducción a los Algoritmos 2C 2025

Práctico 4: Listas y Funciones Recursivas

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**" y notado ":" en Haskell) 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]

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 ▷ [ ]).

El operador ▷ es asociativo a derecha, por lo tanto, es lo mismo escribir x ▷ (y ▷ . . . (z ▷ [ ]) que x ▷ y ▷ . . . z ▷ [ ]. Otros operadores sobre listas son los siguientes:

- #, llamado **cardinal**, toma una lista xs y devuelve su cantidad de elementos. Ej:

  #[1, 2, 0, 5] = 4

  En Haskell #xs se escribe `length xs`.

- ! 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.

  En Haskell xs ! n se escribe `xs !! n`.

- ↑ toma una lista xs (a izquierda) y un natural n que indica una cantidad, y devuelve la sublista con los primeros n elementos de xs. Ej:

  [1, 2, 3, 4, 5, 6] ↑ 2 = [1, 2]

Este operador, llamado **tomar**, asocia a izquierda, por lo tanto xs ↑ n ↑ m se interpreta como (xs ↑ n) ↑ m. En Haskell xs ↑ n se escribe: `take n xs`.

- ↓ toma una lista xs (a izquierda) y un natural n que indica una cantidad, y devuelve la sublista sin los primeros n elementos de xs. Ej:

  [1, 2, 3, 4, 5, 6] ↓ 2 = [3, 4, 5, 6]

  Este operador, llamado **tirar**, se comporta igual al anterior, interpretando xs ↓ n ↓ m como (xs ↓ n) ↓ m. En Haskell xs ↓ n se escribe: `drop n xs`.

- ⧺ toma una lista xs (a izquierda) y otra ys, 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 **concatenar**, es asociativo por lo que podemos escribir sin ambigüedad expresiones sin paréntesis, como xs ⧺ ys ⧺ zs.

  En Haskell xs ⧺ ys se escribe `xs ++ ys`.

- ◃ toma una lista xs (a izquierda) y un elemento y devuelve una lista con todos. los elemento de xs seguidos por x como último elemento. Ej:

  [1, 2] ◃ 3 = [1, 2, 3]

  Este operador, llamado "**pegar a izquierda**", es asociativo a izquierda, luego es lo mismo ([ ] ◃ z) . . . ◃ y) ◃ x que [ ] ◃ z . . . ◃ y ◃ x.

  En Haskell xs ◃ x no está definido, pero podríamos utilizar el operador de concatenar para implementarla de la siguiente manera: `xs++[x]`.

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

- head, llamada **cabeza** o **head** en inglés, toma una lista xs y devuelve su primer elemento. Ej:

  head.[1,2,3] = 1.

- tail, llamada **cola** o **tail** en inglés, 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.

A continuación, listamos los niveles de precedencia de estos operadores. Los que están más arriba tienen mayor precedencia. Cuando hay más de un operador en un nivel de precedencia, es necesario poner paréntesis para evitar la ambigüedad. Por ejemplo x ▷ xs ↑ n se interpreta como x ▷ (xs ↑ n).

!, #, head y tail

↑, ↓

▷ , ◁

++

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.

Ejercicio 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) #[5, 6, 7]

b) [5, 3, 57] ! 1

c) [0, 11, 2, 5] ▷ [ ]

d) [5, 6, 7] ↑ 2

e) [5, 6, 7] ↓ 2

f) head.(0 ▷ [1, 2, 3])

Ejercicio 2: 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 (en mayúsculas). Entonces podemos decir que el operador head toma una lista de tipo [A], donde la variable A

representa cualquier tipo (Num, Bool, [Num], . . .) y devuelve un elemento de esa lista, por lo tanto debe ser un elemento de tipo A.

Notación funcional:

$$head : [A] \rightarrow A$$

Notación de árbol:

$$\underline{head.[A]}$$

$$A$$

Ejercicio 3: 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. Si no están bien escritas, explicá por qué.

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. 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.

Ejercicio 4: ¿Qué hacen las siguientes funciones?

Ayuda: Evaluá las funciones para algunos valores.

a) contar: [Int] → Int
contar.[ ] ≐ 0
contar.(x ▹ xs) ≐ 1 + contar.xs

b) f: [Int] → Bool
f.[ ] ≐ False
f.(x ▹ xs) ≐ x = 5 ∨ f.xs

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

Definí recursivamente las siguientes funciones *fold*.

a) sumar: [Int] → Int, que dada una lista de enteros devuelve la suma de todos sus elementos.

b) prod: [Int] → Int, que dada una lista de enteros devuelve el producto de todos sus elementos.

c) card: [Int] → Int, que dada una lista devuelve la cantidad de elementos de la lista.

d) todosMenores10: [Int] → Bool, que dada una lista devuelve True si ésta consiste sólo de números menores que 10 y False en caso contrario.

e) hay0: [Int] → Bool, que dada una lista decide si existe algún 0 en ella.

Ejercicio 6: Una función *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 *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]

Ejercicio 7: Una función *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]

Ejercicio 8: Definí recursivamente los operadores básicos de listas: #, ! , ◁, ↑, ↓, ♯ . Para los operadores ↑, ↓ y !, deberás hacer recursión en ambos parámetros, en el parámetro lista y en el parámetro numérico.

Funciones recursivas de sobre los números naturales

Ejercicio 9: ¿Qué hacen las siguientes funciones?

Ayuda: Evaluá las funciones para algunos valores.

a) h: Nat → Nat
h.0 ≐ 0
h.(n+1) ≐ 2 + h.n

b) g: Nat → Bool
g.0 ≐ True
g.1 ≐ False
g.(n+2) ≐ g.n

Ejercicio 10: Definí recursivamente las siguientes funciones de Naturales en Naturales

a) la función acumular: Nat → Nat, que dado un Natural n devuelve la suma de todos los naturales menores o iguales a n.

b) la función factorial: Nat → Nat, que dado un Natural n devuelve el factorial de n.

c) la función sumaCuadrados: Nat → Nat, que dado un Natural n devuelve la suma de todos los naturales menores o iguales a n elevados al cuadrado.

d) la función repetir: Nat → Nat → Nat, que dado dos naturales n y m, suma n veces el número m.

Ejercicio 11: Investigar cómo es la función fibonacci.

Ejercicio 12: Utilizando la multiplicación, definir la función potencia, que dado dos naturales b y p devuelve b^p .

Más funciones recursivas

Ejercicio 13: 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”].[“1deCopa”, “3deOro”, “7deEspada”, “2deBasto”] =

[(“Juan”, “1deCopa”), (“Maria”, “3deOro”)]

Defina la función recursivamente.

Ejercicio 14: 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”, ”Rojas”, 18)] =

[”Dominguez””Gutierrez”, ”Rojas”]

Defina la función recursivamente.

Ejercicio 15: Definí funciones por recursión para cada una de las siguientes descripciones. Luego, implementarlas en Haskell.

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. ¡Cuidado con el caso base!

b) sumaPares : [(Num, Num)] → Num, 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 : Num → Num → [Num], 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]

i) Definir la función noEstá: Char -> [Char] -> Bool que determina que una letra está ausente en una palabra.
Por ejemplo:
noEstá . ‘o’. [f,a,m,a,f] = True
noEstá . ‘f’. [f,a,m,a,f] = False

j) Definí la función sumaPond : [(Int, Int)] -> Int que dada una lista de pares de enteros suma cada segundo elemento multiplicado por el primero.
Por ejemplo:
sumaPond.[(3, 7), (0, 100), (2, 4)] = 29
sumaPond.[(1, 3), (2, 9)] = 1*3 + 2*9 = 3 + 18 = 21

k) Definí la función sinA : [String] -> [String] que una lista de palabras devuelve las que no tienen letra ‘a’. (recordar que String = [Char])

Por ejemplo: sinA.[”ocelote”, “murcielago”, “proboscidio”] = [”ocelote”, “proboscidio”]

Problemas

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.

Ejercicio 16: Definí funciones por recursión para cada una de las siguientes descripciones. Luego, implementarlas en Haskell.

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!).

Ejercicio 17: 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 2020.

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.

Ejercicios de finales

Ejercicio 18: Definir la función recursiva dobles : [Num] → [(Num, Num)], que dada una lista de números retorna la lista resultante de armar un par con cada uno de ellos y luego ese mismo número multiplicado por dos. Ejemplo:

dobles.[11, 7, 21] = [(11, 22), (7, 14), (21, 42)]

Ejercicio 19: Definir la función recursiva losOrozco : [Char] → Bool que dada una lista de caracteres (una String) xs retorna True si la única vocal que aparece en xs es la ‘o’, es decir, si no aparece ninguna de las otras vocales, y False si aparece alguna otra. Ejemplos:

(i) losOrozco.“famaf” = False

(ii) losOrozco.“hoy” = True