Matemática y Lógica - 2019

Guía de estudio 2: Listas y Funciones Recursivas

Docente: Araceli Acosta

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* $\triangleright$ *(llamado "pegar" y notado* `:` *en Haskell) es fundamental (se lo denomina constructor) ya que permite construir listas arbitrarias a partir de la lista vacía.* $\triangleright$ *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* $3 \triangleright [\,] = [3]$, *y* $1 \triangleright [2, 3] = [1, 2, 3]$. *Para denotar listas no vacías utilizamos expresiones de la forma* $[x, y, \ldots, z]$, *que son abreviaciones de* $x \triangleright (y \triangleright \ldots \triangleright (z \triangleright [\,]))$. *Como el operador* $\triangleright$ *es asociativo a derecha, es lo mismo escribir* $x \triangleright (y \triangleright \ldots \triangleright (z \triangleright [\,]))$ *que* $x \triangleright y \triangleright \ldots \triangleright z \triangleright [\,]$. *Otros operadores sobre listas son los siguientes:*

- $\#$ *(cardinal) toma una lista* xs *y devuelve su cantidad de elementos. Ej:* $\#[1, 2, 0, 5] = 4$. *En Haskell* $\#xs$ *se escribe:* `length xs`.
- $!$ *(indice) 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`.
- $\uparrow$ *(tomar) 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:* $[1, 2, 3, 4, 5, 6] \uparrow 2 = [1, 2]$. *Este operador, llamado tomar, asocia a izquierda, por lo tanto* $xs \uparrow n \uparrow m$ *se interpreta como* $(xs \uparrow n) \uparrow m$. *En Haskell* $xs \uparrow n$ *se escribe:* `take n xs`.
- $\downarrow$ *(tirar) 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:* $[1, 2, 3, 4, 5, 6] \downarrow 2 = [3, 4, 5, 6]$. *Este operador, llamado tirar, se comporta igual al anterior, interpretando* $xs \downarrow n \downarrow m$ *como* $(xs \downarrow n) \downarrow m$. *En Haskell* $xs \downarrow n$ *se escribe:* `drop n xs`.
- $+\!\!+$ *(concatenar) 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 concatenación, es asociativo por lo que podemos escribir sin ambigüedad expresiones sin paréntesis, como* $xs +\!\!+ ys +\!\!+ zs$. *En Haskell* $xs +\!\!+ ys$ *se escribe:* `xs ++ ys`.
- $\triangleleft$ *(pegar por derecha) 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:* $[1, 2] \triangleleft 3 = [1, 2, 3]$. *Este operador, llamado "pegar a izquierda", es asociativo a izquierda, luego es lo mismo* $([\,] \triangleleft z) \ldots \triangleleft y) \triangleleft x$ *que* $[\,] \triangleleft z \ldots \triangleleft y \triangleleft x$. *En Haskell* $xs \triangleleft x$ *se escribe:* `xs++[x]`.

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.*

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 $x \triangleright xs \uparrow n$ *se interpreta como* $x \triangleright (xs \uparrow n)$.

., #, head y tail	*índice, cardinal, head y tail*
$\uparrow, \downarrow$	*tomar y tirar elementos de una lista*
$\triangleright, \triangleleft$	*pegar a derecha y pegar a izquierda*
$+\!\!+$	*concatenar dos listas*

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:

$$\begin{aligned} & \underline{[23,45,6].(head\ [1,2,3,10,34])} \\ = & \{ \text{ def. de } head \} \\ & \underline{[23,45,6]!1} \\ = & \{ \text{ def. de } ! \} \\ & 45 \end{aligned}$$

En `haskell` los distintos operadores se pueden escribir así:

$head.xs$	`head xs`
$tail.xs$	`tail xs`
$x \triangleright xs$	`x:xs`
$xs \triangleleft x$	`xs ++ [x]`
$xs \uparrow n$	`take n xs`
$xs \downarrow n$	`drop n xs`
$xs +\!\!+ ys$	`xs ++ ys`
$\#xs$	`length xs`
$xs.n$	`xs !! n`

   a) $\#[5,6,7]$

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

   c) $[0,11,2,5] \triangleright [\,]$

   d) $head.(0 \triangleright [1,2,3])$

   e) $([1,2] +\!\!+ [3,4])$.

   f) $(2 = 3) \triangleright [True, False]$

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.*

2. ¿Qué hacen las siguientes funciones?

   Ayuda: Evaluá las funciones para algunos valores.

   a)

$$\begin{aligned} contar &: [Int] \to Int \\ contar.[\,] &\doteq 0 \\ contar.(x \triangleright xs) &\doteq 1 + contar.xs \end{aligned}$$

   b)

$$\begin{aligned} f &: [Int] \to Bool \\ f.[\,] &\doteq False \\ f.(x \triangleright xs) &\doteq x = 5 \vee f.xs \end{aligned}$$

3. Defina recursivamente las siguientes funciones de Naturales en Naturales

   a) la función *sumar*, que dado un Natural n devuelve la suma de todos los naturales menores o iguales a n.

   b) la función *factorial*, que dado un Natural n devuelve el factorial de n.

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

d) la función *repetir*, que dado un dos naturales n y m suma n veces el número m

4. Investigar cómo es la función *fibonacci*.

5. Utilizando la multiplicación, definir la función *potencia*, que dado dos naturales b y p devuelve b^p.

6. Una función de **fold** es aquella que dada una lista devuelve un valor resultante de combinar los elementos de la lista. Por ejemplo: $sum : [Int] \to Int$ devuelve la sumatoria de los elementos de la lista.

   Definí recursivamente las siguientes funciones fold.

   a) $sum : [Int] \to Int$, que dada una lista de enteros devuelve la suma de todos sus elementos.

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

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

   d) $todosMenores10 : [Int] \to Bool$, que dada una lista devuelve *True* si ésta consiste sólo de números menores que 10.

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

7. 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: $duplica : [Int] \to [Int]$ devuelve cada elemento de la lista multiplicado por 2.

   Definí recursivamente las siguientes funciones de map.

   a) $duplica : [Int] \to [Int]$, que dada una lista de enteros duplica cada uno de sus elementos.
   Por ejemplo: $duplica.[3, 0, -2] = [6, 0, -4]$

   b) $sumar1 : [Int] \to [Int]$, que dada una lista de enteros xs le suma 1 (uno) a cada uno de sus elementos,
   Por ejemplo: $sumar1.[3, 0, -2, 12] = [4, 1, -1, 13]$

   c) $sonMayoresQue10 : [Int] \to [Bool]$, que dada una lista de enteros xs devuelve una lista de *Bool* que establece si cada elemento de la lista xs es o no mayor que 10,
   Por ejemplo: $sonMayoresQue10.[3, 0, -2, 12] = [False, False, False, True]$

   d) $cuadruplica : [Int] \to [Int]$, dada na lista de enteros devuelve otra cuyos elementos son el cuádruple de la primera.
   Por ejemplo: $triplica.[2, 5, 1, 7, 3] = [6, 15, 3, 21, 9]$

8. 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: $soloPares : [Int] \to [Int]$ devuelve aquellos elementos de la lista que son pares.

   Definí recursivamente las siguientes funciones filter.

   a) $soloPares : [Int] \to [Int]$, 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: $soloPares.[3, 0, -2, 12] = [0, -2]$

   b) $mayoresQue10 : [Int] \to [Int]$, 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) $sacarCeros : [Int] \to [Int]$, que dado una lista de enteros xs devuelve otra lista sin los 0's,
   Por ejemplo: $sacarCeros.[3, 0, -2, 12] = [3, -2, 12]$

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).

a) *triplica* : $[Int] \to [Int]$, dada na lista de enteros devuelve otra cuyos elementos son el triple de la primera.

Por ejemplo: *triplica*.$[2, 5, 1, 7, 3] = [6, 15, 3, 21, 9]$

b) *maximo* : $[Int] \to Int$, 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. Para el caso de de la lista vacía $maximo.[] = -\infty$

c) *todos0y1* : $[Int] \to Bool$, 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* : $[Int] \to [Int]$ 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* : $[A] \to A$, que devuelve el último elemento de una lista.

Por ejemplo: *ultimo* $[10, 5, 3, 1] = 1$

Ayuda: El caso base es para una lista con un sólo elemento, ya que la lista vacía no tiene último elemento. Es decir, el caso base es: $ultimo.[x] = x$

10. (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* : $[A] \to [A] \to Bool$, 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* : $[(String, Int, Int, Int)] \to [(String, Int)]$, 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* : $[(Int, Int)] \to [(Int, Int)]$, 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* $\to$ *String*, pone espacios entre cada letra de una palabra.
Por ejemplo: *expandir*.`"hola"` = `"h o l a"` (¡sin espacio al final!).

11. **Películas**

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

$(\langle$Nombre de la película$\rangle, \langle$Año de estreno$\rangle, \langle$Duración de la película$\rangle, \langle$Nombre del director$\rangle)$

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

a) Definí la función *verTodas* : $[(String, Int, Int, String)] \to Int$ que dada una lista de películas devuelva el tiempo que tardaría en verlas a todas.

b) Definí la función *estrenos* : $[(String, Int, Int, String)] \to [String]$ que dada una lista de películas devuelva el listado de películas que estrenaron en 2016.

c) Definí la función *filmografia* : $[(String, Int, Int, String)] \to String \to [String]$ 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* : $[(String, Int, Int, String)] \to String \to Int$ que dada una lista de películas y el nombre de una película, devuelva la duración de esa película.