Introducción a los Algoritmos - 1er. cuatrimestre 2011

Guía 2: Listas, funciones, inducción y recursión

Docentes: Araceli Acosta, Carlos Areces, Mariana Badano, Luciana Benotti,
Javier Blanco, Paula Estrella, Pedro Sánchez Terraf, Mauricio Tellechea,

El objetivo de esta guía es introducir dos temas muy importantes: las definiciones recursivas, y la inducción como sistema de demostración asociado a ellas. Comenzaremos también a implementar funciones más complejas en `haskell`.

En esta práctica, como en el resto de las prácticas de esta materia, los ejercicios marcados como ***(!)*** *son algo mas complejos que el resto; y son modelos de los ejercicios que se tomarán en los parciales. Los ejercicios marcados como ☺ son para las almas inquietas que quieren explorar más alla de los contenidos que veremos en la materia.*

Listas

A partir de esta sección extendemos el lenguaje de nuestras fórmulas con el tipo de las listas, denotado como List. Una lista (o secuencia) es una colección ordenada de valores, que deben ser todos del mismo tipo. Denotamos lista vacía con $[\,]$. *El operador* $\triangleright$ *(llamado "pegar") 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:*

- $\triangleleft$ *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* $[\,] \triangleleft z \ldots \triangleleft y \triangleleft x$.
- $\#$, *llamado cardinal, toma una lista* xs *y devuelve su cantidad de elementos. Ej:* $\#.[1,2,0,5] = 4$
- . *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* $(xs.n).m$.
- $\uparrow$ *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* $(xs \uparrow n) \uparrow m$.
- $\downarrow$ *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* $(xs \downarrow n) \downarrow m$.
- $+\!\!+$ *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* $xs +\!\!+ ys +\!\!+ zs$.

Existen además dos funciones fundamentales sobre listas que listamos a continuación. Notar que como son funciones se hace explícita la aplicación de función con el símbolo (.).

- *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* $x \triangleright (xs \uparrow n)$, *pero la expresión* $tail.xs.n$ *no tiene sentido si no se ponen paréntesis (o bien es* $(tail.xs).m$ *o bien* $tail.(xs.n)$*).*

., #, head y tail	*aplicación de función e 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*

Te recomendamos leer las secciones 7.2, 7.4, 7.5, 7.6, 7.8, 7.10, del libro para profundizar en estos conceptos.
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:

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

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

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

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

   d) $[5,6,7] \uparrow 2$

   e) $[5,6,7] \downarrow 2$

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

   g) $([3,5] \triangleright [[14,16],[1,3]]).1$

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

   i) $(([[1]] +\!\!+ [[2]]) \triangleleft [3]) \uparrow 2$.

   j) $(([[\,]] +\!\!+ [[\,]]) \triangleleft ([\,] +\!\!+ [\,])) \uparrow \#([\,] \triangleright [[\,]])$.

   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`

2. Teniendo en cuenta la definición intuitiva de los operadores de listas de la introducción a esta sección, escribí las reglas de tipado de cada uno de ellos. Por ejemplo, el operador *head* toma una lista y devuelve el primer elemento de ella. La lista puede contener cualquier otro tipo, 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 ($Num, Bool, [Num], \ldots$) 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.):

$$head : [A] \to A \qquad\qquad \frac{head.[A]}{A}$$

3. Decidí si las siguientes expresiones están bien escritas, agregando paréntesis para hacer explícita la precedencia y la asociatividad, y justificando con el árbol del tipado correspondiente. Usá un intérprete de `haskell` para verificar los resultados.

   a) $-45 \triangleright [1,2,3]$.
   b) $[1,5,False]$.
   c) $0 \triangleleft [1,2,3]$.
   d) $([1,2] +\!\!+ [3,4]) \triangleleft 5$.
   e) $([1] +\!\!+ [2]) \triangleleft [3]$.
   f) $head.[[5]]$.
   g) $head.[True, False] +\!\!+ [False]$.
   h) $(\#[True, False]) \triangleright [3,4]$.
   i) $[[0,1],[False, False]]$.
   j) $tail.([[2],[4,5]].0)$.
   k) $([1,2] \uparrow 3) \downarrow 2$.

4. Decidí si es posible asignar tipos a las variables $x, y, z, \ldots$ de forma que las expresiones queden bien tipadas. Justificá con un árbol de tipado y una tabla de tipos de las variables.

   a) $x \triangleright y \triangleright z$.
   b) $x \triangleright (y \triangleleft z)$.
   c) $(x \triangleright y) \triangleright z$.
   d) $(x +\!\!+ y) \triangleright (z +\!\!+ w)$.
   e) $head.xs.n = head.(xs.n)$.
   f) $x \triangleright [x]$.
   g) $x \triangleright [[x]]$.
   h) $x \triangleright [[True]] \triangleright y$.
   i) $xs \uparrow ys.4$.
   j) $xs \uparrow [1, 2].0$.
   k) $xs \downarrow xs.2$.

Funciones, inducción y recursión

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 el* 0*, o la lista* $[\,]$*). Por otro lado demostrar para un elemento arbitrario del dominio (por ejemplo* $n+1$*, o la lista* $x \triangleright xs$*) que si suponemos que la propiedad es cierta para todos los elementos más chicos que él (por ejemplo* n*, o* 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.*

El objetivo de los siguientes ejercicios es introducirnos en la ***programación funcional****, es decir, al desarrollo de programas como funciones (generalmente recursivas), y a la demostración (por inducción) de propiedades sobre estos programas.*

5. Definí las funciones simples que describimos a continuación, luego implementálas en `haskell` . Por ejemplo:

   Enunciado: *signo* : $Int \to Int$, que dado un entero retorna su signo, de la siguiente forma: retorna 1 si x es positivo, -1 si es negativo y 0 en cualquier otro caso.

   Solución:

   $$\begin{array}{l} sgn : Num \to Num \\ sgn.x \doteq (\ 0 < x \to\ 1 \\ \qquad \Box\, x < 0 \to -1 \\ \qquad \Box\, x = 0 \to\ 0 \\ \qquad) \end{array}$$

   ```
   sgn :: Int -> Int
   sgn x | 0<x   = 1
         | x<0   = -1
         | x==0  = 0
   ```

   En `haskell` los conectivos para las condiciones se pueden escribir así:

$\wedge$	`&&`
$\vee$	`||`
$\neg$	`not`

   a) *entre0y9* : $Int \to Bool$, que dado un entero devuelve *True* si el entero se encuentra entre 0 y 9.
   b) *segundo3* : $(Int, Int, Int) \to Int$, que dada una terna de enteros devuelve su segundo elemento.
   c) *mayor3* : $(Int, Int, Int) \to (Bool, Bool, Bool)$, que dada una una terna de enteros devuelve una terna de valores booleanos que indica si cada uno de los enteros es mayor que 3.
   Por ejemplo: $mayor3.(1, 4, 3) = (False, True, False)$; $mayor3.(5, 1984, 6) = (True, True, True)$
   d) *ordena* : $(Int, Int) \to (Int, Int)$, que dados dos enteros los ordena de menor a mayor.
   e) *rangoPrecio* : $Int \to String$, que dado un número que representa el precio de una computadora, retorne "muy barato" si el precio es menor a 2000, "demasiado caro" si el precio es mayor que 5000, "hay que verlo bien" si el precio está entre 2000 y 5000, y "esto no puede ser!" si el precio es negativo.
   f) *absoluto* : $Int \to Int$, que dado un entero retorne su valor absoluto.
   g) *esMultiplo2* : $Int \to Bool$, que dado un entero n devuelve *True* si n es múltiplo de 2.
   Ayuda: usar `mod`, el operador que devuelve el resto de la división.
   h) *rangoPrecioParametrizado* : $Int \to (Int, Int) \to String$ que dado un número x, que representa el precio de un producto, y un par $(menor, mayor)$ que represente el rango de precios que uno espera encontrar, retorne "muy barato" si x está por debajo del rango, "demasiado caro" si está por arriba del rango, "hay que verlo bien" si el precio está en el rango, y "esto no puede ser!" si x es negativo.

6. Definí recursivamente los operadores básicos de listas: $\#, \,.\,, \triangleleft, \uparrow, \downarrow, +\!\!+$. Para los operadores $\uparrow, \downarrow$ y . deberás hacer recursión en ambos parámetros, en el parámetro lista y en el parámetro numérico.

7. Demostrá por inducción las siguientes propiedades. Para ello necesitarás las definiciones recursivas de los operadores del ejercicio anterior.

   a) $xs +\!\!+ [\,] = xs$ (la lista vacía es el elemento neutro de la concatenación)

   b) $\#xs \geqslant 0$

   c) $xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$ (la concatenación es asociativa)

   d) $(xs +\!\!+ ys) \uparrow \#xs = xs$

   e) $(xs +\!\!+ ys) \downarrow \#xs = ys$

   f) $xs +\!\!+ (y \triangleright ys) = (xs \triangleleft y) +\!\!+ ys$

   g) $xs +\!\!+ (ys \triangleleft y) = (xs +\!\!+ ys) \triangleleft y$

8. **(!)** Definí funciones por recursión y/o composición para cada una de las siguientes descripciones. Luego evaluá manualmente la función para los valores de cada ejemplo justificando cada paso realizado. Programálas en `haskell` y verificá los resultados obtenidos. Por ejemplo:

   Enunciado: $duplica : [Int] \to [Int]$, que dada una lista de enteros duplica cada uno de sus elementos.

   Por ejemplo $duplica.[2, 5, 12] = [4, 10, 24]$

   Solución:

$$
\begin{aligned}
duplica &: [Int] \to [Int] \\
duplica.[\,] &\doteq [\,] \\
duplica.(x \triangleright xs) &\doteq (x * 2) \triangleright duplica.xs
\end{aligned}
$$

```
duplica :: [Int] -> [Int]
duplica [] = []
duplica (x:xs) = (x*2): duplica xs
```

$$
\begin{aligned}
& \underline{duplica.[2, 5, 12]} \\
= & \{ \text{ def. de } duplica \text{ caso } (x \triangleright xs) \ \} \\
& \underline{(2 * 2)} \triangleright duplica.[5, 12] \\
= & \{ \text{ aritmética } \} \\
& 4 \triangleright \underline{duplica.[5, 12]} \\
= & \{ \text{ def. de } duplica \text{ caso } (x \triangleright xs) \ \} \\
& 4 \triangleright \underline{(5 * 2)} \triangleright duplica.[12] \\
= & \{ \text{ aritmética } \} \\
& 4 \triangleright 10 \triangleright \underline{duplica.[12]} \\
= & \{ \text{ def. de } duplica \text{ caso } (x \triangleright xs) \ \} \\
& 4 \triangleright 10 \triangleright \underline{(12 * 2)} \triangleright duplica.[\,] \\
= & \{ \text{ aritmética } \} \\
& 4 \triangleright 10 \triangleright 24 \triangleright \underline{duplica.[\,]} \\
= & \{ \text{ def. de } duplica \text{ caso } [\,] \ \} \\
& 4 \triangleright 10 \triangleright 24 \triangleright [\,] \\
= & \{ \text{ notación } \} \\
& [4, 10, 24]
\end{aligned}
$$

```
Main> duplica [2,5,12]
[4,10,24]
```

   a) $multiplica : Int \to [Int] \to [Int]$, que dado un número n y una lista, multiplica cada uno de los elementos por n.

   Por ejemplo: $multiplica.3.[3, 0, -2] = [9, 0, -6]$

   ¿Qué relación existe con la función anterior?

   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 'guardar' el mayor.

   c) $esMultiploLista : Int \to [Int] \to [Bool]$, que dado un entero n y una lista de enteros xs devuelve una lista de booleanos que indica si n es múltiplo de cada uno de los elementos de xs.

   Por ejemplo: $esMultiploLista.6.[2, 3, 5] = [True, True, False]$

d) $sum : [Num] \to Num$, que toma una lista de números y devuelve la suma de ellos.
Por ejemplo: $sum.[1,2,3] = 6$

e) $prod : [Num] \to Num$, que toma una lista de números y devuelve el producto de ellos.
Por ejemplo: $prod.[1,2,3,4] = 24$

f) $promedio : [Num] \to Num$, que calcula el valor promedio de los valores de una lista de números reales.
Por ejemplo: $promedio.[6,7,9,4,10] = 7,2$

g) $sumaPares : [(Num, Num)] \to Num$, que dada una lista de pares de números, devuelve la sumatoria de todos los números de todos los pares.
Por ejemplo: $sumaPares.[(1,2)(7,8)(11,0)] = 29$

h) $todos0y1 : [Int] \to Bool$, que dada una lista devuelve *True* si ésta consiste sólo de 0s y 1s.
Por ejemplo: $todos0y1.[1,0,1,2,0,1] = False$, $todos0y1.[1,0,1,0,0,1] = True$

i) $todosMenores10 : [Int] \to Bool$, que dada una lista devuelve *True* si ésta consiste sólo de números menores que 10.
Por ejemplo: $todosMenores10.[2,-3,-5,4,,-11] = True$, $todosMenores10.[2,-4,3,11] = False$

j) $hay0 : [Int] \to Bool$, que dada una lista decide si existe algún 0 en ella.
Por ejemplo: $hay0.[1,2] = False$, $hay0.[1,4,0,5] = True$.

k) $soloPares : [Int] \to [Int]$, 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: $soloPares.[0,1,2,3,4,5,2,3,4] = [0,2,4,2,4]$

l) $quitar0s : [Int] \to [Int]$ que dada una lista de enteros devuelve una la lista pero quitando todos los ceros.
Por ejemplo $quitar0s.[2,0,3,4] = [2,3,4]$

m) $ultimo : [A] \to A$, que devuelve el último elemento de una lista.
Por ejemplo: $ultimo.[10,5,3,1] = 1$

n) $inicio : [A] \to A$, que devuelve todos los elementos de la lista menos el último.
Por ejemplo: $ultimo.[10,5,3,1] = [10,5,3]$

ñ) $pares : [A] \to [A] \to [(A,A)]$, que toma dos listas y devuelve una lista de pares, tal que el n-ésimo elemento es el par de los n-ésimos elementos de cada una de las listas.
Por ejemplo: $pares.[0,1,2,3,4].[10,20,30] = [(0,10),(1,20),(2,30)]$

o) $listasIguales : [A] \to [A] \to Bool$, que determina si dos listas son iguales, es decir, contienen los mismos elementos en las mismas posiciones respectivamente.
Por ejemplo: $listasIguales.[1,2,3].[1,2,3] = True$, $listasIguales.[1,2,3,4].[1,3,2,4] = False$

9. Considerando la función $sum : [Num] \to Num$ del ejercicio 8, demostrá que:

$$sum.(xs +\!\!+ ys) = sum.xs + sum.ys$$

10. Considerando la función $repetir : Nat \to Num \to [Num]$, que construye una lista de un mismo número repetido cierta cantidad de veces, definida recursivamente como:

$$\begin{aligned} repetir.0.x &\doteq [\,] \\ repetir.(n+1).x &\doteq x \triangleright repetir.n.x \end{aligned}$$

demostrá que $\#repetir.n.x = n$.

11. **(!)** Considerando las funciones *sum* y *duplica* del ejercicio 8, demostrá que:

$$sum.(duplica.xs) = 2 * sum.xs$$

12. **(!)** Considerando la función $concat : [[A]] \to [A]$ que toma una lista de listas y devuelve la concatenación de todas ellas, definida recursivamente como:

$$\begin{array}{rcl} concat.[\,] & \doteq & [\,] \\ concat.(xs \triangleright xss) & \doteq & xs +\!\!+ concat.xss \end{array}$$

demostrá que $concat.(xss +\!\!+ yss) = concat.xss +\!\!+ concat.yss$

13. **(!)** Considerando la función $rev : [A] \to [A]$ que toma una lista y devuelve una lista con los mismos elementos pero en orden inverso, definida recursivamente como:

$$\begin{array}{rcl} rev.[\,] & \doteq & [\,] \\ rev.(x \triangleright xs) & \doteq & rev.xs \triangleleft x \end{array}$$

demostrá que $rev.(xs +\!\!+ ys) = rev.ys +\!\!+ rev.xs$

14. Determiná el tipo de cada una de las funciones definidas a continuación, asignando un tipo apropiado a cada variables. Por ejemplo, consideremos la función *más* definida como:

$$más.xs.ys \doteq xs.0 + ys.0$$

Podemos construir el árbol de tipado para el *cuerpo* de la función (lo que está a la derecha de $\doteq$):

$$\dfrac{\dfrac{\dfrac{xs}{[Num].Nat}\,.\,\dfrac{0}{Nat}}{Num} + \dfrac{\dfrac{ys}{[Num].Nat}\,.\,\dfrac{0}{Nat}}{Num}}{Num} + \dfrac{}{}$$

Luego las variables xs y ys son de tipo $[Num]$ y el resultado es de tipo Num. Notar además que todas las variables del cuerpo están listadas como parámetros de la función. Por lo tanto, el tipo de la función *más* es:

$$más : [Num] \to [Num] \to Num$$

Podes verificar los resultados programando en `haskell` las funciones y consultando en el interprete. Por ejemplo, la función *más* la podemos definir como

```
más xs ys = (xs !! 0) + (ys !! 0)
```

Una vez cargada la definición en el intérprete, invocando el comando `:t mas` obtenemos:

```
Main> :t más
más :: Num a => [a] -> [a] -> a
```

   a) $f.xs.n.m \doteq xs.n.m$
   b) $g.x.y \doteq x \triangleright y$
   c) $h.x \doteq [head.x] +\!\!+ [head.x]$
   d) $k.x.xs.y \doteq [x, y, xs.x + y]$
   e) $duplica'.xs \doteq xs +\!\!+ xs$
   f) $cuadruplica.xs \doteq duplica.(duplica.xs)$
   g) $cabeza.xs \doteq (head.xs = 0)$

15. **(!)** ¿Está bien escrita esta expresión? Decidí si es posible dar un tipo a la función f definida como:

$$f.i.j.N.xs.ys \doteq tail.((i \triangleright xs) \triangleright j \triangleright ys) \triangleleft (ys.N.i \triangleright j)$$

En caso afirmativo, da el tipo de la función justificando con un árbol de tipado y una tabla de tipo de las variables.

16. **(!)** Considerando las funciones $bin2dec : [Int] \to Int$ y $repetirUnos : Int \to [Int]$ definidas recursivamente como:

$$\begin{array}{rcl@{\qquad}rcl} bin2dec.[\,] & \doteq & 0 & repetirUnos.0 & \doteq & [\,] \\ bin2dec.(x \triangleright xs) & \doteq & x + 2 * bin2dec.xs & repetirUnos.(n+1) & \doteq & 1 \triangleright repetirUnos.n \end{array}$$

demostrá que $bin2dec.(repetirUnos.n) = 2^n - 1$.

17. ☺ **Torres de Hanoi:** Se tienen tres postes numerados 0, 1 y 2, y n discos de distinto tamaño. Inicialmente se encuentran todos los discos ubicados en el poste 0, ordenados según el tamaño, con el disco más grande en la base. El problema consiste en llevar todos los discos al poste 2, con las siguientes restricciones:

    a) Se puede mover sólo un disco a la vez

    b) Sólo se puede mover el disco que se encuentra más arriba en algún poste.

    c) No se puede colocar un disco sobre otro de menor tamaño.

    Resolvé los siguientes items:

    a) Sea $B = \{0, 1, 2\}$. Definí la función $hanoi : B \to B \to B \to Nat \to [(B, B)]$ tal que $hanoi.a.b.c.n$ calcule la secuencia de *movimientos* para llevar n discos desde el poste a hacia el poste c, utilizando posiblemente el poste b de forma auxiliar. Un *movimiento* es un par (B, B) cuya primer componente indica el poste de salida, y la segunda el poste de llegada. Luego programála en `haskell`.

    Por ejemplo, *hanoi* para los postes 0, 1 y 2, con dos discos es: $hanoi.0.1.2.2 = [(0, 1),\ (0, 2),\ (1, 2)]$

    b) Demostrá que $\#hanoi.a.b.c.n = 2^n - 1$

    c) ¿En qué movimiento se cambia de poste por primera vez el disco de mayor tamaño?

Listas por comprension

Existe un mecanismo poderoso para definir listas, similar a la definición de conjuntos por comprensión. Veamos los siguientes ejemplos:

$$\begin{aligned} [0, 2, 4, 6] &= [2 * x \mid x \leftarrow [0, 1, 2, 3]\,] \\ [4, 16, 36, 64, 100] &= [x * x \mid x \leftarrow [1..10], x\ par] \\ [(1, 1), (1, 2), (1, 3)] &= [(a, b) \mid a \leftarrow [1], b \leftarrow [1, 2, 3]\,] \end{aligned}$$

El símbolo $x \leftarrow [0, 1, 2, 3]$ se lee "x viene de la lista [0,1,2,3]" y se lo denomina ***generador****. Como se puede ver en los ejemplos, una definición por comprensión es de la forma $[e \mid Q]$; donde* **e** *determina la forma de los valores que se incluirán en la lista, y* **Q** *es una secuencia de generadores y/o predicados que determina a partir de qué valores se formarán esos elementos. Un generador indica de qué lista se toman los elementos y un predicado determina qué elementos de la lista considerada son elegidos. Siempre debe haber al menos un generador.*

Algunas abreviaciones de listas muy útiles para utilizar para definir listas por comprension son $[n..m]$ y $[n..]$, donde n, m son números enteros. La primera representa la lista de todos los números entre n y m, y la segunda la lista de todos los números mas grandes que n. Existen otras abreviaciones que son interesantes, jugá con `haskell` *para descubrir cómo funcionan. Por ejemplo, ¿que lista está representada por $[2, 4..]$?*

Las ventajas de notación por comprensión son dos: es fácil de leer y su escritura es muy parecida a la de teoría de conjuntos. Una desventaja es que es más difícil manipular listas por comprensión que listas definidas usando los operadores introducidos anteriormente. Sin embargo las listas por comprensión resultan muy útiles para resolver algunos problemas, como veremos a continuación.

18. Describí con tus palabras qué características tienen los elementos de las siguientes listas definidas por comprensión. Luego escribilas en `haskell` y verificá tus respuestas.

    a) $[x \mid x \leftarrow [1, 2\,..],\ par.x]$

    b) $[(x, y, z) \mid x \leftarrow [0..100], y \leftarrow [0..100], z \leftarrow [0..100], x^2 + y^2 = z^2]$

    c) $[(x, x * x) \mid x \leftarrow [1, 2\,..]\,]$

d) $[(x, y) \mid x \leftarrow ['Juan','Pablo','Eugenia','Sofia','Ana','Laura'], y \leftarrow ['Julian','Eva']]$

e) $[xs \mid y \leftarrow ['Juan','Pablo','Eugenia','Sofia','Ana','Laura'],$
$xs \leftarrow [(x, y) \mid x \leftarrow ['Juan','Pablo','Eugenia','Sofia','Ana','Laura'], x \neq y]\,]$

19. Definir las siguientes funciones mediante listas por comprensión

a) **(!)** *duplica* : $[Int] \to [Int]$, que dada una lista de enteros multiplica por 2 cada cada uno de sus elementos.

Por ejemplo $duplica.[2, 5, 12] = [4, 10, 24]$

b) **(!)** *divisores* : $Int \to [Int]$, que dado un número natural n genera una lista con todos sus divisores.

Por ejemplo: $divisores.6 = [1, 2, 3, 6]$

c) ☺ *primos* : $Int \to [Int]$, que dado un número natural n genera una lista con todos los primos menores que n.

Por ejemplo: $primos.10 = [1, 2, 3, 5, 7]$

20. Evaluá las siguientes expresiones

a) $[\#xs \mid xs \leftarrow [\,]]$

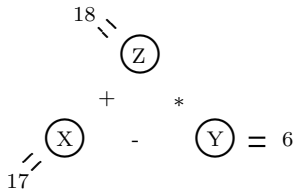
b) $[\#xs \mid xs \leftarrow [\,[], [\,[]\,]\,]]$

c) $[xs +\!\!+ zs \mid xs \leftarrow [\,[], [\,[]\,]\,], ys \leftarrow [\,], zs \leftarrow [\,[]\,]]$

21. *a*) Cuál es la cantidad de elementos que tiene la siguiente lista
$[(x, y) \mid x \leftarrow [1..4], y \leftarrow [3..8]\,]$

b) calcule la cardinalidad de $[\,expr \mid x \leftarrow xs, y \leftarrow ys\,]$ en términos de $\#xs$ y $\#ys$.

22. ☺ Considerá el siguiente puzzle de números:



Encontrá todas ternas de valores que son soluciones, sabiendo que $0 \leqslant x, y, z \leqslant 32$.

23. ☺ Intentá resolver el ejercicio del punto anterior de una forma más eficiente, es decir con menos predicados y generadores más simples.

Apéndice

Niveles de Precedencia

$E(x := a)$, .	sustitución y evaluación
$\sqrt{}, (\cdot)^2$	raíces y potencias
$*, /$	producto y división
máx, mín	máximo y mínimo
$+, -$	suma y resta
$=, \leqslant, \geqslant$	operadores de comparación
$\neg$	negación
$\vee \wedge$	disyunción y conjunción
$\Rightarrow \Leftarrow$	implicación y consecuencia
$\equiv \not\equiv$	equivalencia y discrepancia

Tipos de los operadores

$-x$	$- : Num \to Num$
x^y	$(-)^{(-)} : Num \to Num \to Num$
$\sqrt[r]{x}$	$\sqrt{} : Num \to Num \to Num$
$x * y$	$* : Num \to Num \to Num$
x/y	$/ : Num \to Num \to Num$
x máx y	máx : $Num \to Num \to Num$
x mín y	mín : $Num \to Num \to Num$
$x + y$	$+ : Num \to Num \to Num$
$x - y$	$- : Num \to Num \to Num$
$x > y$	$>: Num \to Num \to Bool$
$x \geqslant y$	$\geqslant: Num \to Num \to Bool$
$x < y$	$<: Num \to Num \to Bool$
$x \leqslant y$	$\leqslant: Num \to Num \to Bool$
$x = y$	$=: Num \to Num \to Bool$
$\neg p$	$\neg : Bool \to Bool$
$p \wedge q$	$\wedge : Bool \to Bool \to Bool$
$p \vee q$	$\vee : Bool \to Bool \to Bool$
$head.xs$	$head : [A] \to A$
$tail.xs$	$tail : [A] \to [A]$
$x \triangleright xs$	$\triangleright : A \to [A] \to [A]$
$xs \triangleleft x$	$\triangleleft : [A] \to A \to [A]$
$xs \uparrow n$	$\uparrow: [A] \to Int \to [A]$
$xs \downarrow n$	$\downarrow: [A] \to Int \to [A]$
$xs +\!\!+ ys$	$+\!\!+ : [A] \to [A] \to [A]$
$\#xs$	$\# : [A] \to Int$
$xs.n$	$(-).(-) : [A] \to Int \to A$

Operadores en `haskell`

x^2	`x^2`
x^n	`x^n` con n entero
x^p	`x**p`
$\sqrt{x}$	`sqrt x`
$\sqrt[r]{x}$	`x**(1/r)`
x máx y	`x ‘max‘ y` o bien `max x y`
x mín y	`x ‘min‘ y` o bien `min x y`
$x \geqslant y$	`x >= y`
$x \leqslant y$	`x <= y`
$x = y$	`x == y`
$\neg p$	`not p`
$p \wedge q$	`p && q`
$p \vee q$	`p || q`
$head.xs$	`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`