

Listas o Secuencias definidas sobre algún conjunto dado**Listas**

Ahora introduciremos una estructura matemática denominada lista o secuencia, con la cual se pueden modelar muchas realidades. Una lista es una ordenación de elementos de un cierto conjunto, una lista puede ser vacía.

Sea A un conjunto cualquiera. Definiremos el conjunto de todas las listas que se pueden construir con elementos de A por inducción. Llamaremos a este conjunto $\text{list } A$

Utilizaremos el constructor `nil` que representa a la lista vacía y el constructor `cons` $\text{cons}: A \rightarrow \text{list } A \rightarrow \text{list } A$, que agrega un elemento a una lista, o sea recibe un elemento de A y una lista de elementos de A , y devuelve una lista de elementos de A .

Definición inductiva del conjunto $\text{list } A$

1. `nil` \in $\text{list } A$
2. $(\text{cons } a \ x) \in \text{list } (A)$ si $a \in A$ y $x \in \text{list } (A)$
3. CC

Revisemos el significado de esta definición:

El axioma `nil` \in $\text{list } A$ establece que `nil` es un elemento del conjunto de listas que se pueden formar con elementos de A , es decir la lista vacía es una lista.

La cláusula $(\text{cons } a \ x) \in \text{list } A$ si $a \in A$ y $x \in \text{list } A$ establece que siempre que a sea un elemento de A y x una lista de elementos de A , $(\text{cons } a \ x)$ es una lista de elementos de A .

En otras palabras establece la regla para construir una lista a partir de un elemento del conjunto A y de una lista dada. Con el constructor `cons` lo que hace es agregar el elemento del conjunto A al principio de la lista que se está construyendo.

Veamos algunos ejemplos de elementos de $\text{list } A$, considerando $A = \{a, b, c, d\}$...

1. `nil` \in $\text{list } A$ - Este elemento surge de la aplicación del axioma - regla i) de la definición
2. $(\text{cons } b \ x) \in \text{list } A$ - Este elemento surge de la aplicación de conclusión del punto 1 y de la regla ii) de la definición.
3. $(\text{cons } c \ x) \in \text{list } A$ - Este elemento surge de la aplicación de conclusión del punto 1 y de la regla ii) de la definición.
4. $(\text{cons } b \ (\text{cons } c \ \text{nil})) \in \text{list } A$ - Este elemento surge de la aplicación de conclusión del punto 3 y de la regla ii) de la definición.
5. $(\text{cons } b \ (\text{cons } b \ (\text{cons } c \ \text{nil}))) \in \text{list } A$ - Este elemento surge de la aplicación de conclusión del punto 4 y de la regla ii) de la definición.

Y así podríamos seguir y construir.....infinitos elementos de $\text{list } A$

Otra notación que simplifica un poco lo engorroso que resulta escribir siempre el constructor `cons`, es la siguiente, que introduce el uso de corchetes rectos para la representación de listas...

Los ejemplos anteriores se verán así con la notación simplificada:

1. `[]` - La lista vacía se representa con corchete recto que abre, seguido de corchete recto que cierra.
2. `[b]` - Lista que contiene como único elemento `a`.
3. `[c]` - Idem 2, con el elemento `c`.
4. `[b, c]` - Lista que contiene los elementos `b` y `c` en ese orden, primero `b` (observar que es el último que se incluyó con el constructor `cons`), luego `c`.
5. `[b, b, c]` - Lista que contiene 3 elementos, `b`, `b` y `c`.

Nota

Si consideramos una lista de la forma `(cons a x)`, se dice que `a` es el primer elemento de la lista y `x` es el resto de la lista. Observar que en el segundo argumento del constructor `cons` siempre debe haber una lista, es decir que el resto de una lista siempre es una lista, en particular puede ser de la forma `nil` o de la forma `cons`.

Definición de funciones por recursión sobre secuencias o listas.

Se plantea el siguiente problema:

Definir una función `largo` que dada una lista de naturales, devuelva su largo, es decir la cantidad de elementos que tiene:

1. `(largo nil) = 0`
2. `(largo (cons a x)) = 1 + (largo x)`

Hemos necesitado realizar una invocación a la propia función en la definición para la cláusula inductiva (recurrencia). Es importante que la invocación a la función en el lado derecho la hagamos siempre con un elemento estructuralmente menor al del lado izquierdo. En este caso el elemento estructuralmente menor a `(cons a x)` del que disponemos es precisamente `x`.

Ejercicios:

1. Definir en Haskell las siguientes funciones
 - a. `aNat` que dado un `Integer` devuelva su representación en el conjunto `N`.

Ejemplo:

```
aNat 4 = (S (S (S (S 0))))
```

- b. `deNat` que dado un elemento de `N` devuelva su representación en el conjunto `Integer`.

```
deNat (S (S 0)) = 2
```

2. Definir la función *igList* que toma dos listas de enteros e indica si son iguales.
3. Definir las siguientes funciones por análisis de casos.
 - a. *esVacía*, que dada una secuencia devuelve True si se trata de la secuencia vacía y False en otro caso.
 - b. *concatenar*, que dadas dos secuencias devuelve la secuencia resultado de agregar todos los elementos de la segunda a continuación de los de la primera.
4. El predicado *ordenada* que dada una lista de enteros especifica si la lista está ordenada en forma creciente según la relación $< : Z \rightarrow Z \rightarrow \text{Bool}$.
5. Definir las siguientes funciones por casos:
 - a. *sumaList*, que dada una secuencia de naturales devuelve la suma de todos los naturales que pertenecen a ella si la secuencia es vacía devuelve 0.
 - b. *multiList*, como *sumaList*, pero devuelve el producto de todos los naturales.
 - c. *invertir*, recibe una secuencia y devuelve la secuencia resultado de invertir la secuencia dada.
 - d. *tomar_n*, recibe un natural n y una secuencia y devuelve la secuencia formada por el resultado de tomar de la secuencia dada los n primeros elementos. Para la secuencia nil, siempre devuelve nil.
Ejemplos: $(\text{tomar_n } 3 [2,3,4,5,1]) = [2,3,4]$
 $(\text{tomar_n } 3 [1,1]) = [1,1]$
 - e. *borrar_n*, similar a tomar, pero devuelve el resultado de eliminar los n primeros elementos de la secuencia dada.
Ejemplos : $(\text{borrar_n } 3 [2,3,4,5,1]) = [5,1]$
 $(\text{borrar_n } 3 [1,1]) = \text{nil}$
 - f. *posicion*, dada una secuencia y un elemento devuelve la posición del elemento en la secuencia. Si el elemento no pertenece a la secuencia devuelve 0.
 - g. *mapear*, recibe una función de Z en Z y una secuencia de enteros. Devuelve una secuencia de enteros que es el resultado de aplicar la función dada a cada uno de los elementos de la secuencia original.
Ejemplo: $(\text{mapear factorial } [2,3,4,0]) = [2,6,24,1]$
 - h. *filtrar*, recibe una función de Z en Bool y una secuencia de enteros y devuelve la secuencia que tiene únicamente los elementos de la secuencia original que para los cuales la aplicación de la función recibida es Verdadera.
Ejemplo: $(\text{filtrar es_par } [2,3,3,5,4,6]) = [2,4,6]$

- i. *existe*, recibe una secuencia de elementos de un cierto conjunto A y una función $f:A \rightarrow \text{Bool}$ y devuelve un valor del conjunto Bool. Devuelve True si hay por lo menos un elemento x en la lista que verifica f, en otro caso devuelve False.
- j. *pertenece*, que recibe un elemento y una lista y devuelve True si el elemento dado está en la lista, False en caso contrario.
- k. *borrar*, recibe una lista y un elemento n, y devuelve la lista resultado de borrar todas las ocurrencias de n en la lista original. Si el elemento no ocurre en la secuencia dada, la devuelve tal cual, si ocurre más de una vez, borra todas las ocurrencias.
- l. *insertaOrd*, recibe una lista de enteros ordenada en forma creciente y un entero e inserta el entero en la lista de forma que el resultado siga siendo una lista ordenada.
- m. *multiplicar*, recibe un número y una secuencia de naturales. Devuelve la secuencia resultado de multiplicar todos los números de la secuencia original por el número dado. Se pide que se defina utilizando mapear y la función multiplicación en N, en su versión prefija elaborada en el práctico $\text{mult}: N \rightarrow N \rightarrow N$.
- n. *capicua*, recibe una secuencia y devuelve un valor del conjunto bool. Devuelve True si y solo si la secuencia recibida es capicúa.
Ejemplos: $(\text{capicua } [2,3,5,6,5,3,2]) = \text{True}$
 $(\text{capicua } [a, b, b, a, b]) = \text{False}$
 $(\text{capicua } []) = \text{True}$

6. Definir todas las funciones pedidas anteriormente en Haskell

Listas en Haskell

Las listas son las estructuras de datos más comúnmente usadas en la programación funcional. Una lista se denota como una secuencia de elementos, separados por comas, rodeado por corchetes, por ejemplo, $[1,2,3]$ es una lista de números enteros.

Una lista puede contener cualquier número de elementos, pero todos los elementos deben tener el mismo tipo. El tipo de una lista de elementos del un cierto conjunto A, es $[A]$.

Por ejemplo:

```
[13,9, -2]:: [Int]
["gato", "perro"]:: [string]
[[1,2], [3,7,1 ],[],[ 900]]:: [[Int]]
```

En las listas pueden aparecer expresiones, que tengan el mismo tipo del resto de los elementos de la lista:

Si está definida la ecuación $x = 10$, entonces
[13,2 2,5 * x] evalúa a la lista [13,4,50].

Un string o cadena es en realidad una lista de caracteres. La constante "abc" no es más que una notación diferente para ['a', 'b', 'c'].

En Haskell se puede especificar en forma corta una lista que es una secuencia de números, ejemplo: la notación [1 .. 10] representa a la lista [1,2,3,4,5,6,7,8,9,10].

Normalmente, los números se incrementan en 1, pero si se dan dos números antes de los dos puntos, se utiliza el incremento sugerido. Por ejemplo, [1,3 .. 10] va de 2 en 2 y su valor es [1,3,5,7,9].

De la misma forma se pueden definir secuencias de caracteres. Ejemplos:

```
['A' .. 'z'] => "abcdefghijklmnopqrstuvwxyz": String
['0' .. '9'] => "0123456789": String
[0 .. 9] => [0,1,2,3,4,5,6,7,8,9]: [Int]
[10,9 .. 0] => [10,9,8,7,6,5,4,3,2,1,0]: [Int]
```

Haskell tiene muchas características que hacen que el uso de las listas sea muy sencillo.

El constructor cons de listas se puede utilizar también con la notación (:).

Al igual que cons, es una función que toma un valor y una lista de e inserta el valor en el primer lugar de la lista:

```
(: :: a -> [a] -> [a]
Entonces:
1: [2,3] = [1,2,3]
1: [] = [1]
```

Cada lista se construye con el operador (:) a partir de una lista dada.

```
[1,2,3,4] = 1: (2: (3: (4: [])))
"abc" = 'a': ('b': ('c': []))
```

El paréntesis pueden omitirse, ya que el (:) el operador asocia a la derecha. Esto permite escribir las ecuaciones más simplemente:

```
[1,2,3,4] = 1: 2: 3: 4: []
"abc" = 'a': 'b': 'c': []
```

Es una práctica común en Haskell utilizar la notación x:xs o y:ys para referirse a la lista no vacía, en el entendido de que x e y son elementos de la lista (los primeros) y xs e ys (o cualquier otra notación que se nos ocurra) es una lista.

Si escribimos en Haskell p:s, entonces es una lista cuyo primer elemento es p y el resto de la lista es s.