Práctico 2: Especificación, Derivación y Verificación de Programas Funcionales

Algoritmos y Estructuras de Datos I
2$^{\text{do}}$ cuatrimestre 2024

Introducción

Esta guía tiene como objetivo obtener las habilidades necesarias para llevar adelante un proceso de derivación o verificación de programas recursivos a partir de especificaciones formales.

Completan esta guía algunos ejercicios para demostrar tanto por inducción sobre naturales, como por inducción ***estructural*** *sobre listas. Se busca mejorar la habilidad para utilizar esta técnica de prueba que es central para la tarea de cálculo de programas recursivos.*

Los ejercicios de cálculo de programas tienen una dificultad creciente: en los primeros, la derivación o verificación se obtiene de manera directa *a través de una demostración inductiva. Los ejercicios sucesivos son más complejos y requieren el uso de técnicas más avanzadas: modularización y generalización.*

Esta guía también incluye ejercicios de ***Laboratorio****. Se incluyen secciones específicas de laboratorio a modo de tutorial para introducir nuevos conceptos de programación en Haskell como la definición de nuevos tipos y de funciones polimórficas.*

1. Dado el **programa**

$$\begin{array}{|l} sum : [Num] \to Num \\ \hline sum.[\,] \doteq 0 \\ sum.(x \triangleright xs) \doteq x + sum.xs \end{array}$$

   a) ¿Qué hace esta función? Escriba en lenguaje natural el **problema** que resuelve.

   b) Escriba una **especificación** de la función con una expresión cuantificada.

   c) **Verifique** que esta especificación vale para toda lista xs.

   d) **Derive** la definición de la función a partir de su especificación.
   ¿Esta derivación es parecida a la demostración en el punto 1c?

2. A partir de las siguientes especificaciones, describir en lenguaje natural qué describe la expresión, dar el tipo de cada función y *derivar* una solución algorítmica para cada caso.

   a) $sum\_cuad.xs = \langle \sum i \;:\; 0 \le i < \#xs \;:\; xs.i * xs.i \rangle$

   b) $iga.e.xs = \langle\, \forall\, i \;:\; 0 \le i < \#xs :\; xs.i = e \,\rangle$

   c) $exp.x.n = x^n$

   d) $sum\_par.n = \langle \sum i \;:\; 0 \le i \le n \wedge par.i \;:\; i \rangle$
   donde $par.i \doteq i \bmod 2 = 0$.

   e) $cuántos.p.xs = \langle \mathrm{N}\; i \;:\; 0 \le i < \#xs \;:\; p.(xs.i) \rangle$

3. Para todos los ítems del ejercicio anterior, dar un ejemplo de uso de la función, es decir: elegir valores concretos para los parámetros y calcular el resultado usando la solución algorítmica obtenida. Las listas deben tener por lo menos tres elementos.

> **Laboratorio 1** Definí en Haskell las funciones derivadas en el ejercicio 2 y evaluá las mismas en los ejemplos utilizados en el ejercicio 3.

Tutorial de laboratorio Parte 1: Definición de tipos. Clases de tipos.

En esta sección trabajaremos con ejercicios de laboratorio que nos irán guiando para introducir nuevos conceptos para la implementación de programas en Haskell. En particular, definiremos nuestros propios tipos de datos. La importancia de poder definir nuevos tipos de datos reside en la facilidad con la que podemos modelar problemas y resolverlos usando las mismas herramientas que para los tipos pre-existentes.

Laboratorio 2 **Tipos enumerados.** Cuando los distintos valores que debemos distinguir en un tipo son finitos, podemos *enumerar* cada uno de los valores del tipo. Por ejemplo, podríamos representar las carreras que se dictan en nuestra facultad definiendo el siguiente tipo:

```
data Carrera = Matematica | Fisica | Computacion | Astronomia
```

Cada uno de estos valores es un *constructor*, ya que al utilizarlos en una expresión, generan un valor del tipo `Carrera`.

a) Implementá el tipo `Carrera` como está definido arriba.

b) Definí la siguiente función, usando *pattern matching*: `titulo :: Carrera -> String` que devuelve el nombre completo de la carrera en forma de *string*. Por ejemplo, para el constructor `Matematica`, debe devolver "Licenciatura en Matemática".

c) Para escribir música se utiliza la denominada *notación musical*, la cual consta de notas (do, re, mi, ...). Además, estas notas pueden presentar algún modificador $\sharp$ (sostenido) o $\flat$ (bemol), por ejemplo $do\sharp$, $si\flat$, etc. Por ahora nos vamos a olvidar de estos modificadores (llamados alteraciones) y vamos a representar las notas básicas.

Definí el tipo `NotaBasica` con constructores `Do`, `Re`, `Mi`, `Fa`, `Sol`, `La` y `Si`

d) El sistema de notación musical anglosajón, también conocido como notación o cifrado americano, relaciona las notas básicas con letras de la A a la G. Este sistema se usa por ejemplo para las tablaturas de guitarra. Programá usando *pattern matching* la función:

```
cifradoAmericano :: NotaBasica -> Char
```

que relaciona las notas `Do`, `Re`, `Mi`, `Fa`, `Sol`, `La` y `Si` con los caracteres `'C'` , `'D'`, `'E'`, `'F'`, `'G'`, `'A'` y `'B'` respectivamente.

Laboratorio 3 **Clases de tipos.** En Haskell usamos el operador `(==)` para comparar valores del mismo tipo:

```
*Main> 4 == 5
False
*Main> 3 == (2 + 1)
True
```

Sin embargo, si intentamos comparar dos valores del tipo `Carrera` veremos que el intérprete nos mostrará un error similar al siguiente:

```
*Main> Matematica == Matematica
• No instance for (Eq Carrera) arising from a use of '=='
```

El problema es que todavía no hemos equipado al tipo nuevo `Carrera` con una noción de igualdad entre sus valores. ¿Cómo logramos eso en Haskell? Debemos garantizar que el tipo `Carrera` sea un miembro de la *clase* `Eq`. Conceptualmente, una clase es un conjunto de tipos que proveen ciertas operaciones especiales:

- Clase `Eq`: tipos que proveen una noción de igualdad (operador ==).
- Clase `Ord`: tipos que proveen una noción de orden (operadores <=, >=, funciones `min`, `max` y más).
- Clase `Bounded`: tipos que proveen una cota superior y una cota inferior para sus valores. Tienen entonces un elemento más grande, definido como la constante `maxBound`, y un elemento más chico, definido como `minBound`.

- Clase `Show`: tipos que proveen una representación en forma de texto (función `show`).
- Muchísimas más.

Podemos indicar al intérprete que infiera automáticamente la definición de una clase para un tipo dado en el momento de su definición, usando `deriving` como se muestra a continuación:

```
data Carrera = Matematica | Fisica | Computacion | Astronomia deriving Eq
```

Ahora es posible comparar carreras:

```
*Main> Matematica == Matematica
True
*Main> Matematica == Computacion
False
```

a) Completá la definición del tipo `NotaBasica` para que las expresiones

```
*Main> Do <= Re
*Main> Fa `min` Sol
```

sean válidas y no generen error. Ayuda: usar *deriving* con múltiples clases.

Laboratorio 4 **Polimorfismo ad hoc** Recordemos la función `sumatoria` del proyecto anterior:

```
sumatoria :: [Int] -> Int
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs
```

La función suma todos los elementos de una lista. Está claro que el algoritmo que se debe seguir para sumar una lista de números enteros y el algoritmo para sumar una lista de números decimales es idéntico. Ahora, si queremos sumar números decimales de tipo `Float` usando nuestra función:

```
*Main> sumatoria [1.5, 2.7, 0.8 :: Float]
    Couldn't match expected type 'Int' with actual type 'Float'
    (:)
```

El error era previsible ya que `sumatoria` no es polimórfica. Si tratamos de usar polimorfismo paramétrico:

```
sumatoria :: [a] -> a
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs
```

cuando recarguemos la definición de `sumatoria`:

```
*Main> :r
    No instance for (Num a) arising from a use of '+'
    (:)
    No instance for (Num a) arising from the literal '0'
    (:)
```

Esto sucede porque en la definición, la variable de tipo a no tiene ninguna restricción, por lo que el tipo no tiene que tener definida necesariamente la suma `(+)` ni la constante `0`. El algoritmo de la función `sumatoria` mientras trabaje con tipos numéricos como `Int`, `Integer`, `Float`, `Double` debería funcionar bien. Todos estos tipos numéricos (y otros más) son justamente los que están en la clase `Num`. Para restringir el polimorfismo de la variable a a esa clase de tipo se escribe:

```
sumatoria :: Num a => [a] -> a
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs
```

Este tipo de definiciones se llaman *polimorfismo ad hoc*, ya que no es una definición completamente genérica.

a) Definí usando polimorfismo *ad hoc* la función `minimoElemento` que calcula (de manera recursiva) cuál es el menor valor de una lista de tipo `[a]`. Asegurate que sólo esté definida para listas no vacías.

b) Definí la función `minimoElemento'` de manera tal que el caso base de la recursión sea el de la lista vacía. Para ello revisá la clase `Bounded`. **Ayuda:** Para probar esta función dentro de `ghci` con listas vacías, indicá el tipo concreto con tipos de la clase `Bounded`, por ejemplo: `([1,5,10]::[Int])`, `([]::[Bool])`, etc.

c) Usá la función `minimoElemento` para determinar la nota más grave de la melodía: `[Fa, La, Sol, Re, Fa]`

En las definiciones de los ejercicios siguientes, deben agregar *deriving* sólo cuando sea estrictamente necesario.

Laboratorio 5 Sinónimo de tipos; constructores con parámetros. En este ejercicio, introducimos dos conceptos: los sinónimos de tipos y tipos algebraicos cuyos constructores llevan argumentos. Un sinónimo de tipo nos permite definir un nuevo nombre para un tipo ya existente, como el ya conocido tipo `String` que no es otra cosa que un sinónimo para `[Char]`. Por ejemplo, si queremos modelar la altura (en centímetros) de una persona, podemos definir:

```
-- Altura es un sinonimo de tipo.
type Altura = Int
```

Los tipos algebraicos tienen constructores que llevan parámetros. Esos parámetros permiten agregar información, generando potencialmente infinitos valores dentro del tipo. Por ejemplo, si queremos modelar datos sobre deportistas, podríamos definir los siguientes tipos:

```
-- Sinónimos de tipo
type Altura = Int
type NumCamiseta = Int

-- Tipos algebráicos sin parámetros (aka enumerados)
data Zona = Arco | Defensa | Mediocampo | Delantera
data TipoReves = DosManos | UnaMano
data Modalidad = Carretera | Pista | Monte | BMX
data PiernaHabil = Izquierda | Derecha
-- Sinónimo
type ManoHabil = PiernaHabil

-- Deportista es un tipo algebraico con constructores paramétricos
data Deportista = Ajedrecista                                -- Constructor sin argumentos
                | Ciclista Modalidad                         -- Constructor con un argumento
                | Velocista Altura                           -- Constructor con un argumento
                | Tenista TipoReves ManoHabil Altura         -- Constructor con tres argumentos
                | Futbolista Zona NumCamiseta PiernaHabil Altura  -- Constructor con ...
```

a) Implementá el tipo `Deportista` y todos sus tipos accesorios (`NumCamiseta`, `Altura`, `Zona`, etc) tal como están definidos arriba.

b) ¿Cuál es el tipo del constructor `Ciclista`?

c) Programá la función `contar_velocistas :: [Deportista] -> Int` que dada una lista de deportistas xs, devuelve la cantidad de velocistas que hay dentro de xs. Programar `contar_velocistas` sin usar igualdad, utilizando pattern matching.

d) Programá la función `contar_futbolistas :: [Deportista] -> Zona -> Int` que dada una lista de deportistas xs, y una zona z, devuelve la cantidad de futbolistas incluidos en xs que juegan en la zona z. No usar igualdad, sólo pattern matching.

e) ¿La función anterior usa `filter`? Si no es así, reprogramala para usarla.

Modularización

En algunas ocasiones la solución del problema original requiere la solución de un "sub-problema". En estos casos suele ser conveniente no atacar ambos problemas simultáneamente, sino "por módulos", cada uno de los cuales debe ser independiente de los demás. Esta técnica se denomina modularización.

En esta sección trabajaremos fundamentalmente con ejercicios que requieren aplicar la técnica de modularización para poder derivarse.

4. Derivá las siguientes funciones.

   a) $f : Num \to Nat \to Num$ computa la suma de potencias de un número, esto es

   $$f.x.n = \langle \sum i \ : 0 \leq i < n : \ x^i \rangle \ .$$

   b) $pi : Nat \to Num$ computa la aproximación del número π

   $$pi.n = 4 * \langle \sum i \ : 0 \leq i < n : \ (-1)^i/(2 * i + 1) \rangle$$

   Ayuda: Modularizar dos veces. La segunda con la función *exp* del ejercicio 2c

   c) $f : Nat \to Nat$ computa el cubo de un número natural x utilizando únicamente sumas. La especificación es muy simple: $f.x = x^3$.

   Ayuda: Usar inducción y modularización varias veces

   d) $f.xs = \langle \exists i \ : 0 < i \leq \#xs : \ \langle \prod j \ : 0 \leq j < \#(xs{\downarrow}i) : \ (xs{\downarrow}i).j \rangle = xs.(i-1) \rangle$

Laboratorio 6 Implementá en Haskell las funciones derivadas en el ejercicio anterior.

5. Especificá formalmente utilizando cuantificadores cada una de las siguientes funciones descriptas informalmente. Luego, *derivá* soluciones algorítmicas para cada una.

   a) *iguales* : $[A] \to Bool$, que determina si los elementos de una lista de tipo A son todos iguales entre sí. Suponga que el operador $=$ es la igualdad para el tipo A.

   b) *minimo* : $[Int] \to Int$, que calcula el mínimo elemento de una lista **no vacía** de enteros.

   Nota: La función no debe devolver $\pm\infty$.

   c) *creciente* : $[Int] \to Bool$, que determina si los elementos de una lista de enteros están ordenados en forma creciente.

   d) *prod* : $[Num] \to [Num] \to Num$, que calcula el producto entre pares de elementos en iguales posiciones de las listas y suma estos resultados (producto punto). Si las listas tienen distinto tamaño se opera hasta la última posición de la más chica.

Laboratorio 7 Implementá en Haskell las funciones derivadas en el ejercicio anterior.

Generalización por abstracción

Una técnica muy poderosa es la generalización por abstracción (por algunos autores llamada inmersión). Aquí la utilizaremos como un medio para resolver una derivación cuando la hipótesis inductiva no puede aplicarse de manera directa. La idea consiste en buscar una especificación más general uno de cuyos casos particulares sea la función en cuestión. Para encontrar la generalización adecuada se introducen parámetros nuevos los cuales servirán para dar cuenta de las subexpresiones que no permiten aplicar la hipótesis inductiva en la derivación original.

6. A partir de las siguientes especificaciones expresá en lenguaje natural qué devuelven las funciones, identificá su tipo y derivalas:

   a) $psum.xs = \langle \forall i \ : 0 \leq i \leq \#xs : \ sum.(xs{\uparrow}i) \geq 0 \rangle$, con *sum* la función del ejercicio 1.

   b) $sum\_ant.xs = \langle \exists i \ : 0 \leq i < \#xs : \ xs.i = sum.(xs{\uparrow}i) \rangle$

   c) $sum8.xs = \langle \exists i \ : 0 \leq i \leq \#xs : \ sum.(xs{\uparrow}i) = 8 \rangle$.

   d) $f.xs = \langle \text{Max} \ i \ : 0 \leq i < \#xs \wedge sum.(xs{\uparrow}i) = sum.(xs{\downarrow}i) : \ i \rangle$.

Laboratorio 8 Implementá en Haskell las funciones derivadas en el ejercicio anterior.

7. Especificá formalmente utilizando cuantificadores cada una de las siguientes funciones descriptas informalmente. Luego, *derivá* soluciones algorítmicas para cada una.

   a) $cuad : Nat \to Bool$, que dado un natural determina si es el cuadrado de un número.

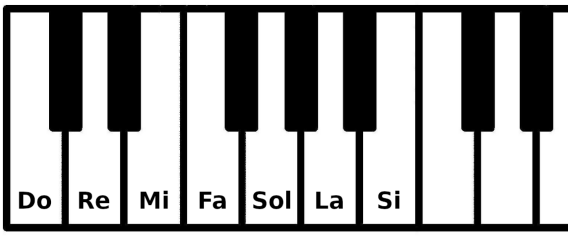
   b) $n8 : [Num] \to Nat$, que cuenta la cantidad de segmentos iniciales de una lista cuya suma es igual a 8.

Laboratorio 9 Implementá en Haskell las funciones derivadas en el ejercicio anterior.

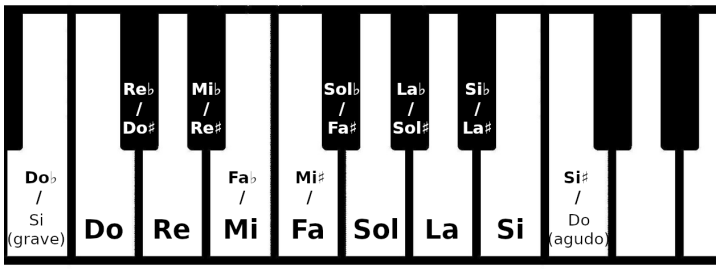
Tutorial de laboratorio parte 2: Clases. Tipos recursivos.

En esta sección retomamos el tutorial de laoratorio para introducir conceptos de tipos en Haskell. En particular, profundizaremos sobre Clases de Tipos, e introducimos tipos recursivos.

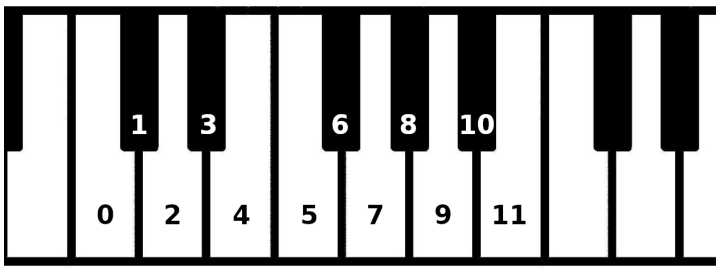
Laboratorio 10 **Definición de clases** Vamos ahora a representar las notas musicales con sus alteraciones. Desde que se utiliza el *sistema temperado* (desde mediados de siglo XVIII aprox) se considera que hay 12 sonidos que se obtienen a partir de las notas musicales. Con el tipo `NotaBasica` logramos representar las 7 notas básicas y definir el orden que hay entre ellas. Una buena manera de ilustrarlo (sin usar un pentagrama musical) es ubicando las notas en un teclado de piano:



Cada tecla del piano produce un sonido, y en ese sentido se puede ver que las teclas negras no las estamos representando y en consecuencia hay sonidos que quedan por fuera. Como se menciono anteriormente, las notas musicales pueden (o no) tener alteraciones. La alteración sostenido (♯) sube ligeramente la afinación (un semitono), por otro lado un bemol (♭) baja en la misma medida la afinación de la nota. Esto se puede ver en el teclado:



Notar que finalmente el sostenido (♯) nos lleva a la siguiente tecla del piano (si no hay una tecla negra intermedia, llegamos a una blanca), de forma similar el bemol (♭) nos lleva a la tecla anterior. Si se enumeran los sonidos partiendo de la nota `Do`, arrancando desde el cero, se pueden visualizar en el teclado:



Podemos definir entonces para las notas básicas la función `sonidoNatural`:

```
sonidoNatural :: NotaBasica -> Int
sonidoNatural Do = 0
sonidoNatural Re = 2
sonidoNatural Mi = 4
sonidoNatural Fa = 5
sonidoNatural Sol = 7
sonidoNatural La = 9
sonidoNatural Si = 11
```

a) Implementá la función `sonidoNatural` como está definida arriba.

b) Definí el tipo enumerado `Alteracion` que consta de los constructores `Bemol`, `Natural` y `Sostenido`.

c) Definí el tipo algebraico `NotaMusical` que debe tener un solo un constructor que llamaremos `Nota` el cual toma dos parámetros. El primer parámetro es de tipo `NotaBasica` y el segundo de tipo `Alteracion`. De esta manera cuando se quiera representar una nota alterada se puede usar como segundo parámetro del constructor un `Bemol` o `Sostenido` y si se quiere representar una nota sin alteraciones se usa `Natural` como segundo parámetro.

d) Definí la función `sonidoCromatico :: NotaMusical -> Int` que devuelve el sonido de una nota, incrementando en uno su valor si tiene la alteración `Sostenido`, decrementando en uno si tiene la alteración `Bemol` y dejando su valor intacto si la alteración es `Natural`

e) Incluí el tipo `NotaMusical` a la clase `Eq` de manera tal que dos notas que tengan el mismo valor de `sonidoCromatico` se consideren iguales.

f) Incluí el tipo `NotaMusical` a la clase `Ord` definiendo el operador <=. Se debe definir que una nota es menor o igual a otra si y sólo si el valor de `sonidoCromatico` para la primera es menor o igual al valor de `sonidoCromatico` para la segunda.

Laboratorio 11 **Tipos enumerados con polimorfismo.**

Usualmente nos encontramos con funciones que no están definidas para ciertos valores de su dominio. Por ejemplo, consideremos la siguiente función:

```
dividir :: Int -> Int -> Int
dividir x y = x `div` y
```

Podemos ver que, como la división por 0 no está definida, el intérprete de Haskell nos muestra un error:

```
$> dividir 4 0
*** Exception: divide by zero
```

¿Cómo podríamos cambiar la definición anterior para que la misma esté definida en todo el dominio? Podríamos usar el tipo `Maybe` que se define en el Preludio de Haskell:

```
data Maybe a = Nothing | Just a
```

Para indicar que la división entera 'no tiene valor' cuando el denominador es 0, usamos el constructor `Nothing`. Cuando tiene valor definido, usamos el constructor `Just`.

```
dividir :: Int -> Int -> Maybe Int
dividir x 0 = Nothing
dividir x y = Just (x `div` y)
```

Y esta vez no recibimos un error si usamos el 0 como denominador:

```
$> dividir 4 0
Nothing

$> dividir 4 2
Just 2
```

Notar en la definición anterior que `Maybe a` es un tipo que depende de la variable a, y en consecuencia `Maybe` es un constructor de tipos polimórfico.

a) Definí la función `primerElemento` que devuelve el primer elemento de una lista no vacía, o `Nothing` si la lista es vacía.

```
primerElemento :: [a] -> Maybe a
```

Laboratorio 12 **Tipos recursivos.** Supongamos que queremos representar una *cola* de deportistas, como aquellas que forman fila para retirar sus credenciales en la villa olímpica. Un deportista llega y se coloca al final de la cola y espera su turno. El orden de atención respeta el orden de llegada, es decir, quien primero llega, es atendido primero. Podemos representar esta situación con el siguiente tipo:

```
data Cola = VaciaC | Encolada Deportista Cola
```

En esta definición, el tipo que estamos definiendo (`Cola`) aparece como un parámetro de uno de sus constructores; por ello se dice que el tipo es *recursivo*. Así una cola o bien está vacía, o bien contiene a una persona encolada, seguida del resto de la cola. Esto nos permite representar colas cuya longitud no conocemos *a priori* y que pueden ser arbitrariamente largas.

I. Programá las siguientes funciones:

a) `atender :: Cola -> Maybe Cola`, que elimina de la cola a la persona que está en la primer posición de una cola, por haber sido atendida. Si la cola está vacía, devuelve `Nothing`.

b) `encolar :: Deportista -> Cola -> Cola`, que agrega una persona a una cola de deportistas, en la última posición.

c) `busca :: Cola -> Zona -> Maybe Deportista`, que devuelve el/la primera futbolista dentro de la cola que juega en la zona que se corresponde con el segundo parámetro. Si no hay futbolistas jugando en esa zona devuelve `Nothing`.

II. ¿A qué otro tipo se parece `Cola`?.

Laboratorio 13 **Tipos recursivos y polimórficos.** Consideremos los siguientes problemas:

- Encontrar la definición de una palabra en un diccionario;
- encontrar el lugar de votación de una persona.

Ambos problemas se resuelven eficientemente, usando un diccionario o un padrón electoral. Estos almacenan la información *asociandola* a otra que se conoce; en el caso del padrón será el número de documento, mientras que en el diccionario será la palabra en sí.

Puesto que reconocemos la similitud entre un caso y el otro, deberíamos esperar poder representar con un único tipo de datos ambas situaciones; es decir, necesitamos un tipo polimórfico que asocie a un dato bien conocido (la clave) la información relevante (el dato).

Una forma posible de representar esta situación es con el tipo de datos recursivo *lista de asociaciones* definido como:

```
data ListaAsoc a b = Vacia | Nodo a b (ListaAsoc a b)
```

Los parámetro del tipo tipo `a` y `b` indican que se trata de un tipo *polimórfico*. Tanto `a` como `b` son variables de tipo, y se pueden *instanciar* con distintos tipos, por ejemplo:

```
type Diccionario = ListaAsoc String String
type Padron      = ListaAsoc Int    String
```

I. ¿Como se debe instanciar el tipo `ListaAsoc` para representar la información almacenada en una guía telefónica?

II. Programá las siguientes funciones:

a) `la_long :: ListaAsoc a b -> Int` que devuelve la cantidad de datos en una lista.

b) `la_concat :: ListaAsoc a b -> ListaAsoc a b -> ListaAsoc a b`, que devuelve la concatenación de dos listas de asociaciones.

c) `la_agregar :: Eq a => ListaAsoc a b -> a -> b -> ListaAsoc a b`, que agrega un nodo a la lista de asociaciones si la clave no está en la lista, o actualiza el valor si la clave ya se encontraba.

d) `la_pares :: ListaAsoc a b -> [(a, b)]` que transforma una lista de asociaciones en una lista de pares *clave-dato*.

e) `la_busca :: Eq a => ListaAsoc a b -> a -> Maybe b` que dada una lista y una clave devuelve el dato asociado, si es que existe. En caso contrario devuelve `Nothing`.

f) `la_borrar :: Eq a => a -> ListaAsoc a b -> ListaAsoc a b` que dada una clave a elimina la entrada en la lista.

Segmentos

Dada una lista xs, un segmento de ella es una lista cuyos elementos están en xs, en el mismo orden y consecutivamente. Por ejemplo: si $xs = [2,4,6,8]$*, entonces* $[2,4], [4,6,8], []$*, son segmentos, mientras que* $[4,2]$ *o* $[2,6,8]$ *no lo son. En problemas como este, suele ser conveniente expresar la lista como una concatenación. Si escribimos* $xs = as +\!\!+ bs +\!\!+ cs$*, entonces as, bs y cs son segmentos de xs, por lo que podemos usarlos para expresar las condiciones que necesitamos que se satisfagan.*

8. **Segmentos de lista:** Para las siguientes expresiones cuantificadas:

   a) $\langle \forall\, as, bs \;:\; xs = as +\!\!+ bs :\; sum.as \geq 0 \,\rangle$

   b) $\langle \text{Min}\;\; as, bs, cs \;:\; xs = as +\!\!+ bs +\!\!+ cs :\; sum.bs \,\rangle$

   c) $\langle \text{N}\, as, b, bs \;:\; xs = as +\!\!+ (b \triangleright bs) :\; b > sum.bs \,\rangle$

   d) $\langle \text{Max}\;\; as, bs, cs \;:\; xs = as +\!\!+ bs \wedge ys = as +\!\!+ cs :\; \#as \,\rangle$

   - Calculá los rangos como conjuntos de tuplas. Tomar $xs = [9, -5, 1, -3]$, $ys = [9, -5, 3]$.
   - Calculá los resultados para las listas dadas.
   - Expresá en lenguaje natural qué significa cada expresión.

9. Expresá utilizando cuantificadores las siguientes sentencias del lenguaje natural:

   a) La lista xs es un segmento inicial de la lista ys.

   b) La lista xs es un segmento de la lista ys.

   c) La lista xs es un segmento final de la lista ys.

   d) Las listas xs e ys tienen en común un segmento no vacío.

   e) La lista xs de numeros enteros tiene la misma cantidad de elementos pares e impares.

   f) La lista xs posee un segmento **no** inicial y **no** final cuyos valores son mayores a los valores del resto de la misma.

10. Derivá funciones recursivas a partir de cada una de las especificaciones que escribiste para los ejercicios 9a y 9b.

Laboratorio 14 Implementá en Haskell las funciones derivadas en el ejercicio anterior.

11. Derivá las funciones especificadas como:

    a) $sumin.xs = \langle \text{Min}\;\; as, bs, cs \;:\; xs = as +\!\!+ bs +\!\!+ cs :\; sum.bs \,\rangle$
    (suma mínima de un segmento).

    b) $f.xs = \langle \text{N}\, as, bs, cs \;:\; xs = as +\!\!+ bs +\!\!+ cs :\; 8 = sum.bs \,\rangle$
    (cantidad de segmentos que suman eso).

c) $maxiga.e.xs = \langle \text{Max } as, bs, cs : xs = as +\!\!+ bs +\!\!+ cs \wedge iga.e.bs : \#bs \rangle$
(máxima longitud de iguales a e)
donde iga es la función del ejercicio 2b.

Laboratorio 15 Implementá en Haskell las funciones derivadas en el ejercicio anterior.

12. Describí en lenguaje natural y dar el *tipo* de cada una de las siguientes funciones especificadas formalmente:

a) $f.xs = \langle \text{N } i, j : 0 \leq i < j < \#xs : xs.i = xs.j \rangle$
b) $g.xs = \langle \text{Max } p, q : 0 \leq p < q < \#xs : xs.p + xs.q \rangle$
c) $h.xs = \langle \text{N } k : 0 \leq k < \#xs : \langle \forall i : 0 \leq i < k : xs.i < xs.k \rangle \rangle$
d) $k.xs = \langle \forall i, j : 0 \leq i \wedge 0 \leq j \wedge i + j = \#xs - 1 : xs.i = xs.j \rangle$
e) $l.xs = \langle \text{Max } p, q : 0 \leq p \leq q < \#xs \wedge \langle \forall i : p \leq i < q : xs.i \geq 0 \rangle : q - p \rangle$

13. Derivá la función definida en el ejercicio 12b.

Laboratorio 16 Implementá en Haskell la función derivada en el ejercicio anterior.

Ejercicios integrados

14. Expresá utilizando cuantificadores las siguientes sentencias del lenguaje natural:

a) El elemento e ocurre un número par de veces en la lista xs.
b) El elemento e ocurre en las posiciones pares de la lista xs.
c) El elemento e ocurre únicamente en las posiciones pares de la lista xs.
d) Si e ocurre en la lista xs, entonces l ocurre en alguna posicion anterior en la misma lista.
e) Existe un elemento de la lista xs que es estrictamente mayor a todos los demás.
f) En la lista xs solo ocurren valores que anulan la funcion f.

15. Derivá funciones recursivas a partir de cada una de las especificaciones del ejercicio 14.

Laboratorio 17 Implementá en Haskell las funciones derivadas en el ejercicio anterior.

Ejercicios extra

16. Sea *fib* la definición recursiva *estándar* para la función de Fibonacci. Derivá la función de Fibolucci, $fbl : Nat \to Nat$, especificada como:

$$fbl.n = \langle \sum i : 0 \leq i < n : fib.i \times fib.(n - i) \rangle$$

17. Derivá la función recursiva a partir de la especificacion del ejercicio 9c.

18. Especificá formalmente utilizando cuantificadores cada una de las siguientes funciones descriptas informalmente. Luego, *derivá* soluciones algorítmicas para cada una.

a) $listas\_iguales : [A] \to [A] \to Bool$, que determina si dos listas son iguales, es decir, contienen los mismos elementos en las mismas posiciones respectivamente.
b) $primo? : Nat \to Bool$, que determina si un numero es primo.

19. Derivá las funciones a partir de las especificaciones en el ejercicio 12.

20. *[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:

   I. Se puede mover sólo un disco a la vez

   II. Sólo se puede mover el disco que se encuentra mas arriba en algún poste.

   III. 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 $hanoi.a.b.c.n$ calcula 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.

      Ayuda: 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 $\#hanoi.a.b.c.n = 2^n - 1$

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