Objetivos

En este proyecto 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.

El objetivo de este proyecto es aprender a declarar nuevos tipos de datos en Haskell y definir funciones para manipular expresiones que utilizan estos tipos.

Ejercicios

Ejercicio 1

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 est´a definido arriba.

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

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".

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

titulo :: Carrera -> String
titulo Matematica = "Licenciatura en Matemática"
titulo Fisica = "Licenciatura en Física"
titulo Computacion = "Licenciatura en Ciencias de la Computación"
titulo Astronomia = "Licenciatura en Astronomía"
```

`Carrera` es un tipo de dato algebraico definido con los constructores mencionados. La función `titulo` utiliza pattern matching para devolver el título correspondiente a la carrera especificada. La derivación `deriving Show` permite imprimir los valores del tipo `Carrera`.

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] (sostenido) o [(bemol), por ejemplo do], si[, etc. Por ahora nos vamos a olvidar de estos modificadores (llamados alteraciones) y vamos a representar las notas básicas. Definir el tipo NotaBasica con constructores Do, Re, Mi, Fa, Sol, La y Si.

```
data NotaBasica = Do | Re | Mi | Fa | Sol | La | Si deriving (Show, Eq, Ord, Bounded)
```

- `Show` permite imprimir valores de tipo `NotaBasica`.
- `Eq` permite realizar comparaciones de igualdad entre valores de tipo `NotaBasica`.
- `Ord` proporciona la capacidad de realizar comparaciones de orden entre valores de tipo `NotaBasica`.
- `Bounded` permite obtener los valores mínimo y máximo del tipo `NotaBasica` (`minBound` y `maxBound` respectivamente).

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. Programar 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.

```
cifradoAmericano :: NotaBasica -> Char
cifradoAmericano Do = 'C'
cifradoAmericano Re = 'D'
cifradoAmericano Mi = 'E'
cifradoAmericano Fa = 'F'
cifradoAmericano Sol = 'G'
cifradoAmericano La = 'A'
cifradoAmericano Si = 'B'
```

Ejercicio 2

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

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

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.

a) Completar 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.

```
data NotaBasica = Do | Re | Mi | Fa | Sol | La | Si deriving (Show, Eq, Ord, Bounded, Enum)
```

Con esta definición, las expresiones `Do <= Re` y `Fa 'min' Sol` serán válidas. La derivación de las clases `Show`, `Eq`, `Ord`, `Bounded`, y `Enum` proporciona las instancias necesarias para realizar comparaciones y otros operadores.

- `Show` permite imprimir valores de tipo `NotaBasica`.
- `Eq` permite realizar comparaciones de igualdad entre valores de tipo `NotaBasica`.
- `Ord` proporciona la capacidad de realizar comparaciones de orden entre valores de tipo `NotaBasica`.
- `Bounded` permite obtener los valores mínimo y máximo del tipo `NotaBasica` (`minBound` y `maxBound` respectivamente).
- `Enum` permite enumerar los valores del tipo `NotaBasica`.

Ejercicio 3

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) Definir usando polimorfismo ad hoc la función minimoElemento que calcula cuál es el menor valor de una lista de tipo [a]. Asegurarse que sólo esté definida para listas no vacías.

```
minimoElemento :: (Ord a) => [a] -> a
minimoElemento (x:[]) = x
minimoElemento (x:y:xs)
    | x < y = minimoElemento (x:xs)
    | otherwise = minimoElemento (y:xs)
```

- `minimoElemento (x:[]) = x`: Define el caso base cuando la lista tiene un solo elemento, en ese caso, el mínimo es el propio elemento.
- `minimoElemento (x:y:xs) | x < y = minimoElemento (x:xs)`: Si el primer elemento `x` es menor que el segundo elemento `y`, entonces se llama recursivamente a `minimoElemento` con la lista que comienza con `x` y el resto de la lista `xs`.
- `| otherwise = minimoElemento (y:xs)`: Si el primer elemento no es menor que el segundo (`otherwise`), entonces se llama recursivamente a `minimoElemento` con la lista que comienza con `y` y el resto de la lista `xs`.

b) Definir la función minimoElemento’ de manera tal que el caso base para la lista vacía esté definido. Para ello revisar la clase Bounded.

```
minimoElemento' :: (Ord a, Bounded a) => [a] -> a
minimoElemento' [] = maxBound
minimoElemento' [x] = x
minimoElemento' (x:y:xs)
    | x < y = minimoElemento' (x:xs)
    | otherwise = minimoElemento' (y:xs)
```

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

```
melodia :: [NotaBasica]
melodia = [Fa, La, Sol, Re, Fa]

notaMasGrave :: NotaBasica
notaMasGrave = minimoElemento melodia
```

En este código, `notaMasGrave` contendrá la nota más grave de la melodía. La función `minimoElemento` se encargará de comparar las notas según la derivación de `Ord` para el tipo `NotaBasica`, el orden es `Do < Re < Mi < Fa < Sol < La < Si`.

Ejercicio 4

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.

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

```
type Altura = Int
type NumCamiseta = Int
data Zona = Arco | Defensa | Mediocampo | Delante deriving Show
data TipoReves = DosManos | UnaMano deriving Show
data Modalidad = Carretera | Pista | Monte | BMX deriving Show
data PiernaHabil = Izquierda | Derecha deriving Show
type ManoHabil = PiernaHabil
data Deportista = Ajedrecista
                | Ciclista Modalidad
                | Velocista Altura
                | Tenista TipoReves ManoHabil Altura
                | Futbolista Zona NumCamiseta PiernaHabil Altura
               deriving Show
```

1. **Tipos sinónimos:**
    - `Altura`: Un sinónimo para representar la altura de una persona, utilizando valores enteros.
    - `NumCamiseta`: Un sinónimo para representar el número de camiseta de un futbolista, utilizando valores enteros.

2. **Datos enumerados:**
    - `Zona` : Representa las distintas zonas en las que puede jugar un futbolista (Arco, Defensa, Mediocampo, Delante).
    - `TipoReves` : Representa los tipos de revés que puede tener un tenista (DosManos, UnaMano).
    - `Modalidad` : Representa las distintas modalidades de un ciclista (Carretera, Pista, Monte, BMX).
    - `PiernaHabil` : Representa la pierna hábil de un deportista (Izquierda, Derecha).

3. **Tipo de dato algebraico:**
    - `Deportista` : Representa distintos tipos de deportistas.
        - `Ajedrecista` : Un ajedrecista que no tiene información adicional.
        - `Ciclista Modalidad` : Un ciclista con una modalidad específica.
        - `Velocista Altura` : Un velocista con una altura específica.
        - `Tenista TipoReves ManoHabil Altura` : Un tenista con tipo de revés, mano hábil y altura específicos.
        - `Futbolista Zona NumCamiseta PiernaHabil Altura` : Un futbolista con información sobre la zona de juego, número de camiseta, pierna hábil y altura.

b) ¿Cuál es el tipo del constructor Ciclista?

```
Ciclista :: Modalidad -> Deportista
```

Esto significa que el constructor `Ciclista` toma un valor de tipo `Modalidad` y devuelve un valor de tipo `Deportista`. Puedes usar este constructor para crear instancias del tipo `Deportista` que sean ciclistas y que tengan asociada una modalidad específica.

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.

```
contar_velocistas :: [Deportista] -> Int
contar_velocistas [] = 0
contar_velocistas (Velocista _ : xs) = 1 + contar_velocistas xs
contar_velocistas (_ : xs) = contar_velocistas xs
```

- En el primer patrón `(Velocista _ : xs)`, si el primer elemento de la lista es un `Velocista`, incrementamos el contador en 1 y continuamos con el resto de la lista.
- En el segundo patrón `(_ : xs)`, estamos coincidiendo con cualquier otro elemento de la lista que no sea un `Velocista`. Simplemente continuamos con el resto de la lista sin incrementar el contador.

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.

```haskell
fulboZona :: Zona -> Deportista -> Bool
fulboZona Arco (Futbolista Arco _ _ _) = True
fulboZona Defensa (Futbolista Defensa _ _ _) = True
fulboZona Mediocampo (Futbolista Mediocampo _ _ _) = True
fulboZona Delante (Futbolista Delante _ _ _) = True
fulboZona _ _ = False

contar_futbolistas :: [Deportista] -> Zona -> Int
contar_futbolistas [] _ = 0
contar_futbolistas (x:xs) zona
    | fulboZona zona x = 1 + contar_futbolistas xs zona
    | otherwise = contar_futbolistas xs zona
```

- La función `fulboZona` es un predicado que verifica si un futbolista juega en una zona específica del campo. La función `contar_futbolistas` cuenta la cantidad de futbolistas en una lista que juegan en una zona específica del campo.

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

```
contar_futbolistas :: [Deportista] -> Zona -> Int
contar_futbolistas deportistas zona =
  length (filter (fulboZona zona) deportistas)
```

1. **Declaración del Tipo:**
   `contar_futbolistas :: [Deportista] -> Zona -> Int`. Esto establece que `contar_futbolistas` es una función que toma dos argumentos: una lista de `Deportista` y una `Zona`, y devuelve un valor de tipo `Int`.
2. **Definición de la Función:**
   - `deportistas` es la lista de deportistas sobre la cual vamos a realizar la operación.
   - `zona` es la zona específica que estamos buscando.
   - `filter (fulboZona zona) deportistas` utiliza la función `fulboZona zona` para filtrar aquellos deportistas cuya zona coincide con la zona proporcionada. Esto devuelve una nueva lista que contiene solo los deportistas que juegan en esa zona.
   - `length` se aplica a la lista filtrada para contar cuántos elementos hay en esa lista.

Ejercicio 5

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.

a) Implementa la función sonidoNatural como está definida arriba.

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

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

```
data Alteracion = Bemol | Natural | Sostenido
```

c) Definir 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.

```
data NotaMusical = Nota NotaBasica Alteracion
```

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.

```
sonidoCromatico :: NotaMusical -> Int
sonidoCromatico (Nota a Sostenido) = 1 + (sonidoNatural a)
sonidoCromatico (Nota a Bemol) = (sonidoNatural a) - 1
sonidoCromatico (Nota a Natural) = (sonidoNatural a)
```

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.

```
instance Eq NotaMusical where
  n1 == n2 = sonidoCromatico n1 == sonidoCromatico n2
```

Esta instancia indica que dos notas musicales son iguales `(==)` si tienen el mismo sonido cromático, es decir, si el resultado de `sonidoCromatico n1` es igual al resultado de `sonidoCromatico n2`.

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.

```
instance Ord NotaMusical where
  n1 <= n2 = sonidoCromatico n1 <= sonidoCromatico n2
```

Esta instancia indica que una nota musical `n1` es menor o igual (`<=`) a otra nota musical `n2` si el sonido cromático de `n1` es menor o igual al sonido cromático de `n2`. De esta manera, la relación de orden está basada en el sonido cromático de las notas.

Ejemplo de uso:

```
let nota1 = Nota Do Bemol
let nota2 = Nota Si Sostenido
let nota3 = Nota Re Natural

-- Usando la instancia de Ord
print (nota1 <= nota2)  -- Devuelve True
print (nota1 <= nota3)  -- Devuelve False
print (nota2 <= nota3)  -- Devuelve False
```

Ejercicio 6

Definir la función primerElemento que devuelve el primer elemento de una lista no vacía, o Nothing si la lista es vacía.

```
primerElemento :: [a] -> Maybe a
primerElemento [] = Nothing
primerElemento (x:xs) = Just x
```

Ejercicio 7

```
data Cola = VaciaC | Encolada Deportista Cola
  deriving (Show)
```

Programá las siguientes funciones:

1. `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.

```
atender :: Cola -> Maybe Cola
atender VaciaC = Nothing
atender (Encolada d c) = Just c
```

- Si la cola es vacía (`VaciaC`), entonces `atender` devuelve `Nothing` , ya que no hay elementos para atender.
- Si la cola tiene al menos un elemento (`Encolada _ c`), entonces `atender` devuelve `Just c` .

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

```
encolar :: Deportista -> Cola -> Cola
encolar d VaciaC = Encolada d VaciaC
encolar d1 (Encolada d2 c) = Encolada d2 (encolar d1 c)
```

- Si la cola está vacía (`VaciaC`), entonces `encolar` devuelve una nueva cola que consiste en el deportista `d` .
- Si la cola ya tiene elementos (`Encolada d2 c`), entonces `encolar` devuelve una nueva cola que tiene los mismos elementos, pero el deportista `d1` se agrega al final de la cola.

Ejemplo de uso

```
let colaInicial = VaciaC
let colaDespues1 = encolar Futbolista Delante 10 Derecha 175 colaInicial
let colaDespues2 = encolar Ciclista Carretera colaDespues1

-- colaDespues1 es Encolada (Futbolista Delante 10 Derecha 175) VaciaC
-- colaDespues2 es Encolada (Ciclista Carretera) (Encolada (Futbolista Delante 10 Derecha 175) VaciaC)
```

3. `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.

```
busca :: Cola -> Zona -> Maybe Deportista
busca VaciaC zona = Nothing
busca (Encolada (Futbolista z n p a) c) zona
  | fulboZona zona d = Just d
  | otherwise = busca c zona
where
    d = Futbolista z n p a
busca (Encolada d c) zona = busca c zona
```

Ejemplo de uso

Se tiene el siguiente dato:

```
cola = Encolada (Futbolista Defensa 5 "Ana") VaciaC $
       Encolada (Futbolista Mediocampo 7 "Carlos") $
       Encolada (Futbolista Delante 10 "Juan") VaciaC
```

Y queremos analizar el resultado de aplicarle la función de esta forma:

```
let resultado = busca cola Delante
```

Se llama a la función `busca` con la cola y la zona Delante como parámetros.

1. **Primera Iteración:**
    - La cola actual es `Encolada (Futbolista Defensa 5 "Ana") ...`.
    - El futbolista actual es `Futbolista Defensa 5 "Ana"`.
    - Se compara la zona del futbolista (`Defensa`) con la zona objetivo (`Delante`), y como no son iguales, se continúa buscando en la cola restante.

2. **Segunda Iteración:**
    - La cola actual es `Encolada (Futbolista Mediocampo 7 "Carlos") ...`.
    - El futbolista actual es `Futbolista Mediocampo 7 "Carlos"`.
    - Se compara la zona del futbolista (`Mediocampo`) con la zona objetivo (`Delante`), y como no son iguales, se continúa buscando en la cola restante.

3. **Tercera Iteración:**
    - La cola actual es `Encolada (Futbolista Delante 10 "Juan") VaciaC`.
    - El futbolista actual es `Futbolista Delante 10 "Juan"`.
    - Se compara la zona del futbolista (`Delante`) con la zona objetivo (`Delante`), y como son iguales, se devuelve `Just (Futbolista Delante 10 "Juan")`.

El resultado es `Just (Futbolista Delante 10 "Juan")`, ya que Juan es el primer futbolista que se encuentra en la zona Delante de la cola, a pesar de que Ana y Carlos también estén presentes en la cola.

```
resultado = Just (Futbolista Delante 10 "Juan")
```

¿A qué otro tipo se parece Cola?

Se puede ver una similitud con el tipo Lista de Haskell donde VaciaC podria ser [] y Encolada representaria la definición recursiva de la lista a:[]

Ejercicio 8

```haskell
data ListaAsoc a b = Vacia | Nodo a b (ListaAsoc a b)
    deriving (Show)
```

a) ¿Como se debe instanciar el tipo ListaAsoc para representar la información almacenada en una guía telefónica?

```haskell
type GuiaTelefonica = ListaAsoc Int String
```

b) Programá las siguientes funciones

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

```
la_long :: ListaAsoc a b -> Int
la_long Vacia = 0
la_long (Nodo a b lA) = 1 + la_long lA
```

La función `la_long` calcula la longitud de una lista asociativa:

- Si la lista es vacía (`Vacia`), entonces su longitud es 0.
- Si es un nodo (`Nodo _ _ lA`), la longitud es 1 más la longitud de la lista asociativa restante (`lA`).

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

```
la_concat :: ListaAsoc a b -> ListaAsoc a b -> ListaAsoc a b
la_concat Vacia la2 = la2
la_concat la1 Vacia = la1
la_concat (Nodo a b lA1) lA2 = Nodo a b (la_concat lA1 lA2)
```

- Si la primera lista (`la1`) es vacía (`Vacia`), entonces la concatenación es simplemente la segunda lista (`la2`).
- Si la segunda lista (`la2`) es vacía (`Vacia`), entonces la concatenación es simplemente la primera lista (`la1`).
- Si ambas listas son nodos (`Nodo a b lA1` y `lA2`), entonces se crea un nuevo nodo con los mismos valores (`a` y `b`) y la concatenación de las listas restantes (`lA1` y `lA2`).

3. `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.

```haskell
la_agregar :: (Eq a) => ListaAsoc a b -> a -> b -> ListaAsoc a b
la_agregar Vacia a b = Nodo a b Vacia
la_agregar (Nodo a b lA) a' b' | a == a' = Nodo a b' lA
                               | otherwise = Nodo a b (la_agregar lA a' b')
```

- Si la lista es vacía (`Vacia`), se crea un nuevo nodo con los valores proporcionados (`Nodo a b Vacia`).
- Si la lista es un nodo (`Nodo a b lA`):
  - Si el valor `a` coincide con el valor `a'` , se actualiza el valor `b` en el nodo existente (`Nodo a b' lA`).
  - Si no hay coincidencia, se crea un nuevo nodo con los valores proporcionados y se llama recursivamente a la función para el resto de la lista asociativa (`Nodo a b (la_agregar lA a' b')`).

Ejemplo de uso

Supongamos que se tiene el siguiente dato:

```
let listaOriginal = Nodo 1 "Juan" (Nodo 2 "Ana" Vacia)
```

Ahora vamos a agregar un nuevo par de valores usando la función definida:

```
-- Agregar un nuevo par de valores
let listaNueva = la_agregar listaOriginal 3 "Carlos"
```

- La lista original no es vacía y es un nodo (`Nodo 1 "Juan" (Nodo 2 "Ana" Vacia)`).
- El valor `a` del primer nodo es `1`, y no coincide con `3`, por lo que se crea un nuevo nodo con los mismos valores y se llama recursivamente para el resto de la lista.
- La llamada recursiva se realiza con el resto de la lista (`Nodo 2 "Ana" Vacia`).
- En el siguiente nodo (`Nodo 2 "Ana" Vacia`), el valor `a` es `2`, y no coincide con `3`, por lo que se crea un nuevo nodo con los mismos valores y el resto de la lista se convierte en `Vacia`.
- Finalmente, se crea un nuevo nodo con los valores proporcionados, y el resultado es `Nodo 1 "Juan" (Nodo 2 "Ana" (Nodo 3 "Carlos" Vacia))`.

Y el resultado final sería:

```
listaNueva = Nodo 1 "Juan" (Nodo 2 "Ana" (Nodo 3 "Carlos" Vacia))
```

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

```haskell
la_pares :: ListaAsoc a b -> [(a,b)]
la_pares Vacia = []
la_pares (Nodo a b lA) = (a,b) : la_pares lA
```

La función `la_pares` tiene como objetivo convertir una lista asociativa (`ListaAsoc`) en una lista de pares `(a, b)`

- Si la lista es vacía (`Vacia`), el resultado es una lista vacía `[]` .
- Si la lista es un nodo (`Nodo a b lA`), se crea un par `(a, b)` y se concatena con la lista resultante de aplicar recursivamente la función al resto de la lista (`lA`).

5. `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.

```haskell
la_busca :: (Eq a) => ListaAsoc a b -> a -> Maybe b
la_busca Vacia a = Nothing
la_busca (Nodo a b lA) a' | a == a' = Just b
                          | otherwise = la_busca lA a'
```

La función `la_busca` busca un valor asociado a una clave en una lista asociativa (`ListaAsoc`).

- Si la lista es vacía (`Vacia`), no hay ningún valor asociado a la clave, y se devuelve `Nothing`.
- Si la lista es un nodo (`Nodo a b lA`):
  - Si la clave `a` coincide con la clave buscada `a'`, se devuelve `Just b`, que es el valor asociado.
  - Si no hay coincidencia, se llama recursivamente a la función con el resto de la lista asociativa (`lA`).

Ejemplo de uso

Se tiene el dato

```
listaAsociativa = Nodo 1 "Juan" (Nodo 2 "Ana" Vacia)
```

y se quiere buscar el valor asociado a la clave 2:

```
la_busca listaAsociativa 2
```

- La lista no es vacía y es un nodo (`Nodo 1 "Juan" (Nodo 2 "Ana" Vacia)`).
- La clave `a` del primer nodo es `1`, que no coincide con la clave buscada `2`, por lo que se llama recursivamente a la función con el resto de la lista asociativa (`Nodo 2 "Ana" Vacia`).
- En el siguiente nodo (`Nodo 2 "Ana" Vacia`), la clave `a` es `2`, que coincide con la clave buscada `2`, por lo que se devuelve `Just "Ana"`.

```
resultado = Just "Ana"
```

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

```
la_borrar :: (Eq a) => a -> ListaAsoc a b -> ListaAsoc a b
la_borrar a Vacia = Vacia
la_borrar a' (Nodo a b lA) | a' == a = lA
                           | otherwise = Nodo a b (la_borrar a' lA)
```

La función `la_borrar` tiene como objetivo eliminar un par clave-valor asociado a una clave dada en una lista asociativa (`ListaAsoc`).

- Si la lista es vacía (`Vacia`), no hay nada que borrar, y se devuelve la lista vacía.
- Si la lista es un nodo (`Nodo a b lA`):
  - Si la clave `a'` coincide con la clave del nodo `a` , se devuelve el resto de la lista asociativa (`lA`), eliminando así el par clave-valor asociado a `a'` .
  - Si no hay coincidencia, se crea un nuevo nodo con los mismos valores y se llama recursivamente a la función para el resto de la lista asociativa (`lA`).

Ejemplo de uso

Se tiene el dato

```
listaAsociativa = Nodo 1 "Juan" (Nodo 2 "Ana" Vacia)
```

y se quiere borrar el par asociado a la clave 2:

```
la_borrar 2 listaAsociativa
```

- La lista no es vacía y es un nodo (`Nodo 1 "Juan" (Nodo 2 "Ana" Vacia)`).
- La clave `a'` que queremos borrar es `2` , y la clave del primer nodo `a` también es `2` .
- En este caso, la función devuelve el resto de la lista asociativa (`Vacia`), ya que estamos eliminando el nodo que contiene el par asociado a la clave `2` .

```
listaSinElemento = Nodo 1 "Juan" Vacia
```

Ejercicio 9

Otro tipo de datos muy útil y que se puede usar para representar muchas situaciones es el árbol; por ejemplo, el análisis sintáctico de una oración, una estructura jerárquica como un árbol genealógico o la taxonomía de Linneo.
En este ejercicio consideramos árboles binarios, es decir que cada rama tiene sólo dos descendientes inmediatos.

```
data Arbol a = Hoja | Rama (Arbol a) a (Arbol a)
```

a) `a_long :: Arbol a -> Int` que dado un árbol devuelve la cantidad de datos almacenados.

```
a_long :: Arbol a -> Int
a_long Hoja = 0
a_long (Rama aIzq _ aDer) = 1 + a_long aIzq + a_long aDer
```

- Si el árbol es una `Hoja`, la longitud es `0` ya que no tiene nodos.
- Si el árbol es una `Rama`, la longitud es `1` (el nodo actual) más la longitud de los subárboles izquierdo (`aIzq`) y derecho (`aDer`). Esto se calcula recursivamente sumando las longitudes de los subárboles.

Por ejemplo:

```
-- Crear un árbol
let arbolEjemplo = Rama (Rama Hoja 1 Hoja) 2 (Rama Hoja 3 Hoja)

-- Calcular la longitud del árbol
let longitudArbol = a_long arbolEjemplo
-- La longitud del árbol sería 3
```

b) `a_hojas :: Arbol a -> Int` que dado un árbol devuelve la cantidad de hojas.

```
a_vacio :: Arbol a -> Bool
a_vacio Hoja = True
a_vacio (Rama _ _ _) = False
a_hojas :: Arbol a -> Int
a_hojas Hoja = 0
a_hojas (Rama aIzq _ aDer) =
  if a_vacio aIzq && a_vacio aDer
    then 1
      else a_hojas aIzq + a_hojas aDer
```

Función a_vacio

- Si el árbol es una `Hoja`, entonces está vacío, y la función devuelve `True`.
- Si el árbol es una `Rama`, entonces no está vacío, y la función devuelve `False`.

Función a_hojas

- Si el árbol es una `Hoja`, la cantidad de hojas es `0`, ya que no hay ninguna hoja.
- Si el árbol es una `Rama`, se verifica si tanto el subárbol izquierdo (`aIzq`) como el subárbol derecho (`aDer`) están vacíos utilizando la función `a_vacio`.
- Si ambos subárboles están vacíos, entonces el nodo actual es una hoja, y se cuenta `1`.
- Si al menos uno de los subárboles no está vacío, la función se llama recursivamente sobre ambos subárboles y se suma la cantidad de hojas.

Ejemplo de uso

Se tiene el siguiente arbol:

```
-- Crear un árbol con una hoja y dos nodos
let arbolEjemplo = Rama (Rama Hoja 1 Hoja) 2 (Rama Hoja 3 Hoja)
```

`arbolEjemplo` se define como un árbol con una estructura que tiene una `Rama` en la raíz. El subárbol izquierdo de la raíz es otra `Rama` con una `Hoja` conteniendo el valor `1`. El subárbol derecho de la raíz es otra `Rama` con una `Hoja` conteniendo el valor `3`.
Ahora queremos contar la cantidad de hojas en el árbol:

```
-- Contar la cantidad de hojas en el árbol
let cantidadHojas = a_hojas arbolEjemplo
```

- La función `a_hojas` se llama con `arbolEjemplo`.
- Se verifica si el árbol es una `Hoja`. En este caso, no es una `Hoja`, sino una `Rama`.
- Se evalúa la expresión `if a_vacio aIzq && a_vacio aDer then 1 else a_hojas aIzq + a_hojas aDer` para la `Rama`:
  - Se llama a `a_hojas` recursivamente en el subárbol izquierdo (`Rama Hoja 1 Hoja`). Este subárbol tiene una hoja, por lo que devuelve `1`.
  - Se llama a `a_hojas` recursivamente en el subárbol derecho (`Rama Hoja 3 Hoja`). Este subárbol también tiene una hoja, por lo que devuelve `1`.
  - La expresión final es `1 + 1`, lo que resulta en `2`.
- La cantidad de hojas en `arbolEjemplo` es `2`.

c) `a_inc :: Num a => Arbol a -> Arbol a` que dado un árbol que contiene números, los incrementa en uno.

```haskell
a_inc :: (Num a) => Arbol a -> Arbol a
a_inc Hoja = Hoja
a_inc (Rama aIzq a aDer) = Rama (a_inc aIzq) (a + 1) (a_inc aDer)
```

1. **Caso Base:**
   - Si el árbol es una `Hoja`, la función devuelve la misma `Hoja`. No hay valores que incrementar en una hoja.
2. **Caso Recursivo (Rama):**
   - Si el árbol es una `Rama`, se aplica la función recursivamente a los subárboles izquierdo (`aIzq`) y derecho (`aDer`).
   - El valor en el nodo actual (`a`) se incrementa en `1` y se coloca en un nuevo nodo `Rama`.

Ejemplo de uso

```
-- Crear un árbol con una hoja y dos nodos
let arbolOriginal = Rama (Rama Hoja 1 Hoja) 2 (Rama Hoja 3 Hoja)

-- Aplicar la función a_inc al árbol
let arbolIncrementado = a_inc arbolOriginal
```

El árbol es el siguiente:

```
  2
 / \
1   3
```

`arbolOriginal` es un árbol con una estructura que tiene una `Rama` en la raíz. El subárbol izquierdo de la raíz es otra `Rama` con una `Hoja` conteniendo el valor `1`. El subárbol derecho de la raíz es otra `Rama` con una `Hoja` conteniendo el valor `3`.

- Se aplica `a_inc` al árbol original.

- En cada nodo, el valor se incrementa en `1`.

```
  3
 / \
2   4
```

`arbolIncrementado` es el nuevo árbol donde cada valor del árbol original ha sido incrementado en `1`.

d) `a_map :: (a -> b) -> Arbol a -> Arbol b` que dada una función y un árbol, devuelve el árbol con la misma estructura, que resulta de aplicar la función a cada uno de los elementos del árbol. Revisá la definición de la función anterior y reprogramala usando a_map.

```
a_map :: (a -> b) -> Arbol a -> Arbol b
a_map f Hoja = Hoja
a_map f (Rama aIzq a aDer) = Rama (a_map f aIzq) (f a) (a_map f aDer)
```

La función `a_map` toma una función `f` que transforma valores de tipo `a` en valores de tipo `b` y un árbol de valores de tipo `a`, y devuelve un nuevo árbol donde la función `f` ha sido aplicada a cada valor del árbol original.

1. **Caso Base:**
   - Si el árbol es una `Hoja`, la función devuelve la misma `Hoja`. No hay valores que transformar en una hoja.
2. **Caso Recursivo (Rama):**
   - Si el árbol es una `Rama`, se aplica la función recursivamente a los subárboles izquierdo (`aIzq`) y derecho (`aDer`).
   - La función `f` se aplica al valor en el nodo actual (`a`).
   - Se construye un nuevo nodo `Rama` con los subárboles transformados y el nuevo valor.

Ejemplo de uso

```
-- Crear un árbol con una hoja y dos nodos
let arbolOriginal = Rama (Rama Hoja 1 Hoja) 2 (Rama Hoja 3 Hoja)

-- Definir una función que duplica los valores
let duplicar = (*2)

-- Aplicar la función a_map al árbol con la función duplicar
let arbolDuplicado = a_map duplicar arbolOriginal
```

El árbol es el siguiente

```
  2
 / \
1   3
```

`arbolOriginal` es un árbol con una estructura que tiene una `Rama` en la raíz. El subárbol izquierdo de la raíz es otra `Rama` con una `Hoja` conteniendo el valor `1`. El subárbol derecho de la raíz es otra `Rama` con una `Hoja` conteniendo el valor `3`.

- Se define una función `duplicar` que multiplica un número por `2`.
- Se aplica `a_map` al árbol original con la función `duplicar`.
- En cada nodo, la función `duplicar` se aplica al valor.

```
   4
  / \
 2   6
```

`arbolDuplicado` es el nuevo árbol donde la función `duplicar` ha sido aplicada a cada valor del árbol original.

Para redefinir `a_inc` usando la función `a_map`, se hace de la siguiente manera:

```
incrementar :: (Num a) => a -> a
incrementar x = x + 1

a_inc :: (Num a) => Arbol a -> Arbol a
a_inc = a_map incrementar
```

Aquí, se define la función `incrementar` que toma un valor `x` y lo incrementa en `1`. Luego, `a_map` se utiliza con `incrementar` como argumento en la definición de `a_inc`.

También se podría utilizar una función lambda de la siguiente forma:

```
a_inc :: (Num a) => Arbol a -> Arbol a
a_inc = a_map (\x -> x + 1)
```

Ejercicio 10

Un tipo también muy útil, es el árbol binario de búsqueda (ABB). Un ABB es una estructura de datos donde cada nodo tiene un valor y cumple con la propiedad de que los valores en el subárbol izquierdo son menores que el valor del nodo, y los valores en el subárbol derecho son mayores.

a) Definir el tipo recursivo ABB utilizando los constructores:

- RamaABB :: ABB a -> a -> ABB a
- VacioABB :: ABB a

```
data ABB a = VacioABB | RamaABB (ABB a) a (ABB a) deriving (Show)
```

b) Definir una función `insertarABB` que tome un valor y un árbol binario como entrada y devuelva un nuevo árbol que contenga el valor insertado en el árbol original. La función tiene que tener el siguiente tipado: `insertarABB :: Ord a => a -> ABB a -> ABB a`.

```haskell
insertarABB :: Ord a => a -> ABB a -> ABB a
insertarABB x VacioABB = RamaABB VacioABB x VacioABB
insertarABB x (RamaABB ri n rd)
  | x <= n = RamaABB (insertarABB x ri) n rd
  | otherwise = RamaABB ri n (insertarABB x rd)
```

La función `insertarABB` toma un valor `x` y un ABB, y devuelve un nuevo ABB que incluye el valor `x`. Si el árbol está vacío (`VacioABB`), se crea un nuevo nodo con `x` como raíz. Si el árbol no está vacío, se compara `x` con el valor en el nodo actual (`n`). Si `x` es menor o igual, se inserta en el subárbol izquierdo; de lo contrario, se inserta en el subárbol derecho.

Ejemplo de uso

```
-- Crear un árbol vacío
let arbolVacio = VacioABB

-- Insertar valores en el árbol
let arbol1 = insertarABB 5 arbolVacio
let arbol2 = insertarABB 3 arbol1
let arbol3 = insertarABB 7 arbol2
let arbol4 = insertarABB 1 arbol3
let arbol5 = insertarABB 4 arbol4
```

1. Árbol Vacío (`arbolVacio`):

```
VacioABB
```

2. Insertar 5 (`arbol1`):

```
    5
```

3. Insertar 3 (`arbol2`):

```
    5
   /
  3
```

4. Insertar 7 (`arbol3`):

```
    5
   / \
  3   7
```

5. Insertar 1 (`arbol4`):

```
    5
   / \
  3   7
 /
1
```

6. **Insertar 4 (`arbol5`):**

```
    5
   / \
  3   7
 / \
1   4
```

El resultado final es un ABB que cumple con las propiedades de un árbol binario de búsqueda. Los valores se insertan de manera que el subárbol izquierdo de cada nodo contiene valores menores y el subárbol derecho contiene valores mayores.

c) Define una función llamada `buscarEnArbol` que tome un valor y un árbol binario como entrada y devuelva True si el valor está presente en el árbol y False en caso contrario. La función tiene que tener el siguiente tipado: `buscarABB :: Eq a => a -> ABB a -> Bool`.

```haskell
buscarEnArbol :: Eq a => a -> ABB a -> Bool
buscarEnArbol x VacioABB = False
buscarEnArbol x (RamaABB ri n rd)
  | x == n = True
  | otherwise = buscarEnArbol x ri || buscarEnArbol x rd
```

La función `buscarEnArbol` es una función que busca un valor `x` en un Árbol Binario de Búsqueda (ABB). La función toma el valor a buscar (`x`) y el ABB, y devuelve `True` si el valor está presente en el árbol y `False` en caso contrario.

La implementación es recursiva y sigue las reglas de búsqueda en un ABB:

- Si el árbol está vacío (`VacioABB`), el valor no está presente, y la función devuelve `False` .
- Si el valor en el nodo actual (`n`) es igual a `x` , se ha encontrado el valor y la función devuelve `True` .
- Si `x` es menor que `n` , se busca en el subárbol izquierdo (`ri`).
- Si `x` es mayor que `n` , se busca en el subárbol derecho (`rd`).

La búsqueda se realiza recursivamente hasta encontrar el valor o llegar a un nodo vacío.

Ejemplo de uso

```
-- Crear un árbol
let arbol = RamaABB (RamaABB VacioABB 3 VacioABB) 5 (RamaABB VacioABB 7 VacioABB)

-- Buscar valores en el árbol
buscarEnArbol 5 arbol  -- Devuelve True, ya que 5 está en el árbol
buscarEnArbol 4 arbol  -- Devuelve False, ya que 4 no está en el árbol
```

1. **Árbol (`arbol`):**

```
   5
  / \
 3   7
```

1. **Buscar 5 (`buscarEnArbol 5 arbol`):**
    - Se encuentra el valor `5` en la raíz, por lo que devuelve `True`.
2. **Buscar 4 (`buscarEnArbol 4 arbol`):**
    - `4` es menor que `5`, por lo que se busca en el subárbol izquierdo (`3`).
    - `4` es mayor que `3`, por lo que se busca en el subárbol derecho de `3` (que es vacío).
    - Al llegar a un nodo vacío, devuelve `False` ya que `4` no está en el árbol.

d) Definir la función verificarABB que devuelve True si el árbol cumple con la propiedad fundamental o False en caso contrario. De manera auxiliar pueden definir las funciones :

- `mayor_a_todos :: Ord a => a -> ABB a -> Bool`
- `menor_a_todos :: Ord a => a -> ABB a -> Bool`

```haskell
mayorQueTodos :: Ord a => a -> ABB a -> Bool
mayorQueTodos x VacioABB = True
mayorQueTodos x (RamaABB ri n rd)
  | x > n = mayorQueTodos x ri && mayorQueTodos x rd
  | otherwise = False
```

La función `mayorQueTodos` verifica si `x` es mayor que todos los elementos en el ABB. La lógica es la siguiente:

- Si el árbol está vacío (`VacioABB`), entonces `x` es mayor que todos los elementos (porque no hay elementos).
- Si `x` es mayor que el valor en el nodo actual (`n`), entonces se verifica recursivamente en el subárbol izquierdo (`ri`) y en el subárbol derecho (`rd`). La función devuelve `True` solo si `x` es mayor que todos los elementos en ambos subárboles.
- Si `x` no es mayor que `n`, entonces devuelve `False`.

```haskell
menorQueTodos :: Ord a => a -> ABB a -> Bool
menorQueTodos x VacioABB = True
menorQueTodos x (RamaABB ri n rd)
  | x < n = menorQueTodos x ri && menorQueTodos x rd
  | otherwise = False
```

La función `menorQueTodos` verifica si `x` es menor que todos los elementos en el ABB. La lógica es similar a `mayorQueTodos`, pero se invierten las comparaciones.

Ambas funciones son útiles para verificar si un valor es extremo (mayor o menor) en un ABB en comparación con todos los demás valores del árbol.

```haskell
verificarABB :: Ord a => ABB a -> Bool
verificarABB VacioABB = True
verificarABB (RamaABB ri n rd) = mayorQueTodos n ri && menorQueTodos n rd
```

La función `verificarABB` tiene la intención de verificar si un Árbol Binario de Búsqueda (ABB) cumple con las propiedades de un ABB. La idea es comprobar que cada nodo en el árbol sea mayor que todos los elementos en su subárbol izquierdo y menor que todos los elementos en su subárbol derecho.

La lógica es la siguiente:

- Si el árbol está vacío (`VacioABB`), cumple con las propiedades y devuelve `True` .
- Si el árbol no está vacío (`RamaABB`), verifica que el valor en el nodo actual (`n`) sea mayor que todos los elementos en su subárbol izquierdo (`ri`) y menor que todos los elementos en su subárbol derecho (`rd`).