Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 2

Tipos de datos

1. Objetivo

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.

2. Ejercicios

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 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 ♯ (sostenido) o ♭ (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.

   Definir 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. 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.

2. **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:

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

Ahora es posible comparar carreras:

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

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.

3. **Polimorfismo ad hoc** Recordemos la función `sumatoria` del proyecto anterior:

```haskell
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 (de manera recursiva) cuál es el menor valor de una lista de tipo `[a]`. Asegurarse que sólo esté definida para listas no vacías.

b) Definir la función `minimoElemento'` de manera tal que el caso base de la recursión sea el de la lista vacía. Para ello revisar la clase `Bounded`.

Ayuda: *Para probar esta función dentro de* `ghci` *con listas vacías, indicar el tipo concreto con tipos de la clase* `Bounded`*, por ejemplo:* `([1,5,10]::[Int])`, `([]::[Bool])`, *etc.*

c) Usar 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.

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. 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. Programa `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. Programa `contar_futbolistas` sin usar igualdad, utilizando pattern matching.

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

5. **Definición de clases.** Vamos ahora a representar la 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 ($\sharp$) sube ligeramente la afinación (un semitono), por otro lado un bemol ($\flat$) 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) Definir el tipo enumerado `Alteracion` que consta de los constructores `Bemol`, `Natural` y `Sostenido`.

c) Definir el tipo `NotaMusical` que consta de un único constructor llamado `Nota` que 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.

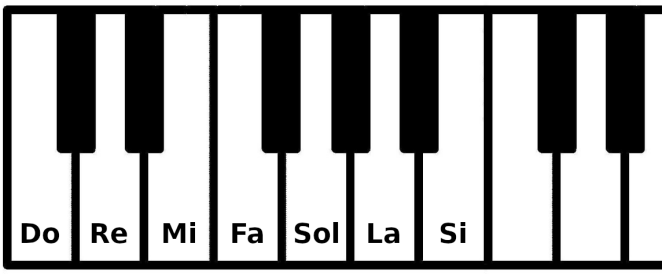
f) Incluí el tipo `NotaMusical` a la clase `Ord` definiendo el operador `<=`. Se debe definir que una nota es menor o igual a otra si y sólo si el valor de `sonidoCromatico` para la primera es menor o igual al valor de `sonidoCromatico` para la segunda.

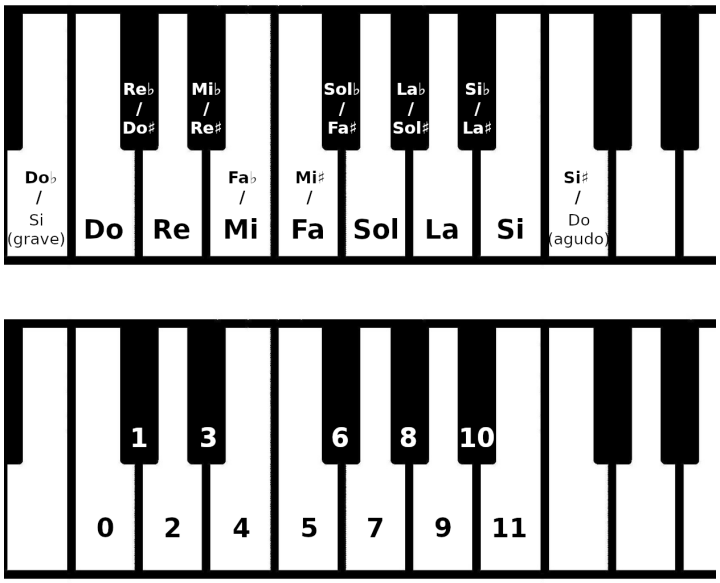
6. **Tipos enumerados con polimorfismo.**

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:

```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`.

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

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

7. **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.

a) 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`.

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

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`.

b) ¿A qué otro tipo se parece `Cola`?.

8. **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
```

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

b) Programá las siguientes funciones:

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

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

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.

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

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`.

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

9. (**Punto** $\star$) 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.

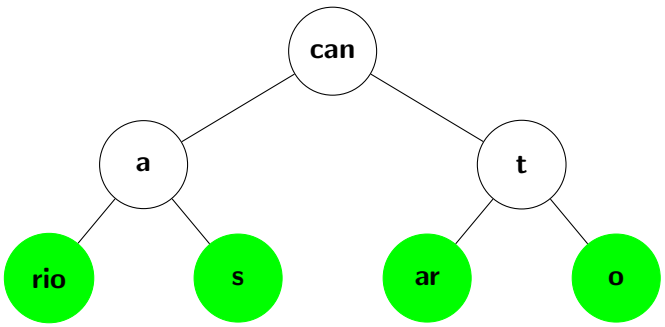
   En este ejercicio consideramos *árboles binarios*, es decir que cada *rama* tiene sólo dos descendientes inmediatos:

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

   Como se muestra a continuación, usando ese tipo de datos podemos por ejemplo representar los prefijos comunes de varias palabras.

```
type Prefijos = Arbol String

can, cana, canario, canas, cant, cantar, canto :: Prefijos
can     = Rama cana "can" cant
cana    = Rama canario "a" canas
canario = Rama Hoja "rio" Hoja
canas   = Rama Hoja "s" Hoja
cant    = Rama cantar "t" canto
cantar  = Rama Hoja "ar" Hoja
canto   = Rama Hoja "o" Hoja
```



   Programá las siguientes funciones:

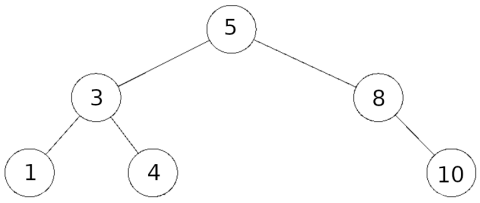
   a) `a_long :: Arbol a -> Int` que dado un árbol devuelve la cantidad de datos almacenados.

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

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

   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`.

10. (**Punto** $\star$) 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. Por ejemplo:



En este ABB, el valor 5 es la raíz. Los valores menores que 5 están en el subárbol izquierdo (3, 1, 4), y los valores mayores están en el subárbol derecho (8, 10). Esta estructura permite búsquedas eficientes: si deseas encontrar el valor 4, comienzas en la raíz y avanzas hacia la izquierda y luego hacia la derecha para encontrarlo en el subárbol izquierdo.

Programar los siguientes apartados:

a) : Definir el tipo recursivo `ABB` utilizando los constructores:

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

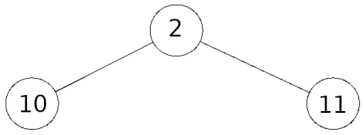
b) Definir una función `insertarABB` que tome un valor y un árbol binario como entrada y devuelva un nuevo árbol que contenga el valor insertado en el árbol original. La función tiene que tener el siguiente tipado:
`insertarABB :: Ord a => a -> ABB a -> ABB a`

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`

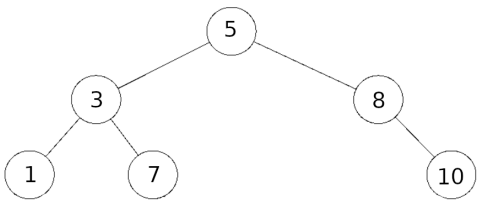
d) Notar que los constructores `RamaABB` y `VacioABB` permiten construir árboles binarios que no cumplen la propiedad fundamental de los *árboles binarios de búsqueda*. Por ejemplo:

```
RamaABB (RamaABB VacioABB 10 VacioABB) 2 (RamaABB VacioABB 11 VacioABB)
```

El árbol construido se ilustra a continuación:



Claramente no es un ABB, otro caso no tan trivial es:



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`

Asegurarse que la función `verificarABB` devuelva `False` para los dos ejemplos mostrados. Para probar la función en el primer ejemplo pueden definir:

```
ejemplo1 = RamaABB (RamaABB VacioABB 10 VacioABB)
                    2
                   (RamaABB VacioABB 11 VacioABB)
```

y luego evaluar

```
*Main> verificarABB ejemplo1
```

Se debe definir análogamente `ejemplo2` con la expresión que represente el árbol del segundo ejemplo. Además deben incluir como comentarios en el código el resultado de la ejecución de `verificarABB` para `ejemplo1` y `ejemplo2`.