Algebraic Data Types
Lab 3

1 Introduction: how do we define new types?

In Haskell we can define new data types. For example, we can define our own type that models various mobile devices we use:

```
data MobileDevice  =  Smartphone
                   |  Laptop
                   |  Tablet
```

Observația 1.1.
- The `data` keyword is used to create a new type;
- Before `=` we have the name of the newly created type: `MobileDevice`;
- After `=` we have the definition of *value constructors*, which start with a capital letter and are separated by the symbol `|`: `Smartphone`, `Laptop` and `Tablet`.

Exercițiul 1.2. Load into `ghci` a file containing the definition of the `MobileDevice` type. What are the types for `Smartphone`, `Laptop` and `Tablet`?

The type `Bool`, which is already defined in Haskell, looks like this:

```
data Bool  =  False
           |  True
```

If you try to evaluate in `ghci` directly the value `True` notice that `ghci` displays the value `True`:

```
cmd > True
True
```

However, for a value of type `MobileDevice` we do not have the same behavior:

```
cmd > Laptop
<interactive>:11:1:  error:
• No instance for (Show MobileDevice) arising from a use of ‘print’
• In a stmt of an interactive GHCi command:  print it
```

We get this error because `ghci` does not associate a string representation of the `Laptop` value. To solve this problem, one solution is to add `deriving (Show)` to the type definition like this:

```
    data MobileDevice  =  Smartphone
                       |  Laptop
                       |  Tablet
                          deriving (Show)
```

Regarding the phrase `deriving (Show)`, we only need to know for now that it allows constructing string representations for values of this type. We will discuss this construction at length in the next course. After reloading the definition, in `ghci` we get the expected behavior:

```
cmd > Laptop
Laptop
```

For now we've defined a new data type that has only three possible values: `Smartphone`, `Laptop`, and `Tablet`. We generally want to be able to construct types that have multiple values. For example, we can have tablets of different sizes. To achieve this, we can add new *fields* to the value constructors:

```
    data MobileDevice  =  Smartphone
                       |  Laptop
                       |  Tablet Int
                          deriving (Show)
```

Notice that for the `Tablet` constructor we added a field of type `Int`. This will allow us to construct more values of type `MobileDevice`:

```
cmd > :t (Tablet 12)
(Tablet 12) :: MobileDevice
cmd > :t (Tablet 15)
(Tablet 15) :: MobileDevice
```

For a constructor we can add multiple fields. For example, we can extend the above definition to also attach the brand of the tablets:

```
    data MobileDevice  =  Smartphone
                       |  Laptop
                       |  Tablet Int String
                          deriving (Show)
```

```
cmd > :t (Tablet 15 "Asus")
(Tablet 15 "Asus") :: MobileDevice
cmd > :t (Tablet 15 "Apple")
(Tablet 15 "Apple") :: MobileDevice
```

Exercițiul 1.3. Create a data type `Colors` that contains some colors. Next, change the `MobileDevice` type so that you can attach colors for each device.

As expected, we can also define functions on top of the new data types. When we want to write a function over a datatype of our own creation, we will use *pattern matching* over the type's values. We will explain this with an example: suppose we want to write a function that will return for each device a description. Considering the first definition for `MobileDevice`, the function will be implemented like this:

```
description ::  MobileDevice -> String
description Laptop      =  "Acesta este un laptop de culoare roz."
description Tablet      =  "Aceasta este o tableta mov."
description Smartphone  =  "Acesta este un telefon mobil."
```

Notice that the function is defined for each individual constructor. Thus, when `description` is executed, the function definition corresponding to the given argument is searched for. If the argument is `Smartphone`, then the last line is the appropriate one.

Exercițiul 1.4. Using the `Colors` data type, write a function that displays the color of each device.

2 Binary trees. Binary search trees.

Exercițiul 2.1. Define a data type for binary trees where nodes contain integers.

```
date Arb = Leaf | Node Integer Arb Arb deriving (Show, Eq)
```

Exercițiul 2.2. Write a recursive function `minBST :: Arb -> Integer` that returns the minimum element in a binary tree to search.

The function may be partial, in the sense that it is not defined for empty tree.

Hint: The minimum element is the leftmost element.

Exercițiul 2.3. Write a recursive function `maxBST :: Arb -> Integer` that returns the maximum element in a binary tree to search.

The function may be partial, in the sense that it is not defined for empty tree.

Hint: the maximum element is the rightmost element.

Exercițiul 2.4. Write a function that checks whether a binary tree is a tree search binary.

```
isBST :: Arb -> Bool
isBST Leaf = True
...
```

Hint: for every subtree T of the given tree at input, the function tests that the root node of T is:

(A) greater than all elements in its left subtree T and

(B) less than all elements in the right subtree of T.

Assuming that the subtrees of T are already known to be binary search trees, test (A) can be implemented simply using `maxBST`, and test (B) using `minBST`.

Exercițiul 2.5. Write a function that searches for an integer value in a tree search binary.

```
search :: Arb -> Integer -> Bool
```

Exercițiul 2.6. Write a function that inserts an integer value into a binary search tree.

```
insert :: Arb -> Integer -> Arb
```

Exercițiul 2.7. Write a function that deletes (an instance of) the largest element.

```
removeMax :: Arb -> Arb
```

Exercițiul 2.8. Write a function that deletes (an instance of) a value of type integer in a binary search tree. You will likely use `maxBST` and `removeMax` as helper functions.

```
remove :: Arb -> Integer -> Arb
```

Exercițiul 2.9. Write functions that compute pre-order, in-order traversal and post-order of the trees.

```
preOrder :: Arb -> [Integer]
inOrder :: Arb -> [Integer]
postOrder :: Arb -> [Integer]
```

3 Boolean Expressions

Exercițiul 3.1. Define a data type for Boolean expressions (Boolean variables, true/false constants, operators (and, or, not)).

Exercițiul 3.2. Define a function to simplify Boolean expressions.

Exercițiul 3.3. Define a function to bring a Boolean expression to the form normal conjunctiva.