4 Algebraic data types

Exercise 4.1 (*Warm-up*: Algebraic data type notations, `ADTs.hs`). A number of algebraic types are defined below. If possible, give three *different* expressions for each of these types. For polymorphic types, what is the value of the type variables for these expressions?

```
data Day       = Mon | Tue | Wed | Thu | Fri | Sat | Sun
data Prop      = Prop :→ Prop | T | F
data Unit      = Unit
data Foo a     = Bar a
data Tuple a b = Two a b | One a | None
data Wrapped a = Wrapped (Wrapped a) | Bare a
```

Check your answers in GHCi!

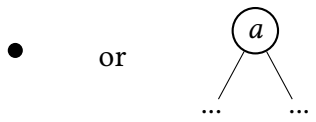
Note that in some of these cases, a name of a *data constructor* has the same name as the name of the *type*, and in other cases they are different. This exercise is of course deliberately confusing, but it often happens in actual Haskell code too! For some reflection on this, see Hint 4

Exercise 4.2 (*Warm-up*: Type inference and ADTs, `ADTTypes.hs`). Below a number of functions are given without their type signatures. Determine which type Haskell derives for them. These functions use the types defined in Exercise 4.1, and standard functions from the Prelude.

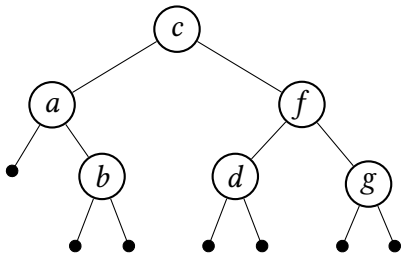
```
isWeekend Sat = True
isWeekend Sun = True
isWeekend _   = False

bval prop = case prop of
              T         → True
              F         → False
              (p :→ q) → not (bval p) || bval q

incr (Bar k) = Bar (lines k)
```

As always, check your answers in GHCi!

Exercise 4.3 (*Warm-up*: Binary Trees, `Tree.hs`). Algebraic data types lend themselves well for implementing all kinds of trees; the most well-known is the *binary tree*. A *binary tree* is a data structure that is either empty, or a branching node that consists of an element, and continues in a left subtree and a right subtree. Visualized as:



Terminology surrounding trees is inspired by biology (root, leaf, etc.) and family relations (child, parent, etc.). In computer science, trees grow downwards. Hence, the top element is called the *root*, and the empty trees at the bottom are the *leaves*.



Consider the binary tree shown above. The node labelled *c* is the *root*. It has two children: a tree that starts at the node labelled *a* and one that starts at the node labelled *f*. Not all nodes have two children: *a* only has a right child, and *b* has none. Often, the leaves of the tree are not drawn.

The recursive definition of trees can be captured as an algebraic data type:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

Like `[a]`, the type of a list, `Tree a` is a polymorphic type: it stores elements of type `a`. Thus, we can have a tree *of* strings, a tree *of* integers, a tree *of* lists of things, and even trees *of* trees ...

A binary tree is called a *binary search tree* if the elements can be ordered (i.e., they are instances of the `Ord` type class), and if for *every* node *A* it is the case that all elements stored in its left child are less than its own element, and all elements in the right child are greater than its own element. This implies there are no duplicate elements.

1. The tree above stores characters. Give the expression of the type `Tree Char` that represents it using the type constructors `Leaf` and `Node`. Is it a binary search tree?

2. Draw the tree associated with this expression:

   ```
   Node 3 (Node 2 (Node 1 Leaf Leaf) Leaf) (Node 4 (Node 5 Leaf Leaf) Leaf)
   ```

3. Here are some tree-functions that you can write using the *tree design pattern*, as explained during the lecture. Implement them!

   (a) `leaves :: Tree a → Int`, which counts the number of `Leaf` nodes.

   (b) `nodes :: Tree a → Int`, which counts the number of non-`Leaf` nodes.
   If a binary tree has *n* `leaves`, how many `nodes` does it have?

   (c) `height :: Tree a → Int`, which returns the maximum *height* of the tree, that is, the length of the longest path from the root of the tree to a `Leaf` node. The height in the picture above has height 3.

   (d) `elems :: Tree a → [a]`, giving all elements in a tree (in any order you like).
   After you have written this function, you can uncomment the `layout` and `putTree` function in `Tree.hs` to get a function that displays trees in a spatial format.

   (e) `isSearchTree :: (Ord a) ⇒ Tree a → Bool`, which checks if a tree is a *binary search tree*. Don't worry about efficiency, just make it as simple as you can (list comprehensions may be useful).

   If at some point you get errors about 'missing number of arguments' in one of these functions, see Hint 1.

Exercise 4.4 (*Mandatory*: Binary search trees, `Tree.hs`).
This exercise continues with binary trees (see Exercise 4.3), but focuses on *binary search trees*.
Write the following functions:

1. `member :: (Ord a) ⇒ a → Tree a → Bool`, which checks whether an element can be found in a binary search tree. (The function can assume that the input tree is a *binary search tree*.)

2. `insert :: (Ord a) ⇒ a → Tree a → Tree a`, which inserts an element in a binary search tree, returning the updated search tree.

   ```
   >>> (insert "Marc" . insert "Twan" . insert "Sjaak") Leaf
   Node "Sjaak" (Node "Marc" Leaf Leaf) (Node "Twan" Leaf Leaf)
   ```

   If an element was already present in the tree, the tree is not changed.

3. `fromList :: (Ord a) ⇒ [a] → Tree a`, which constructs a tree from the elements in the provided list by repeatedly calling `insert`.

   In the file `Words.hs`, you can find a selection of 2500 rather obscure words taken from the RIDYHEW[1] list; you can use it as a non-trivial data set to test your functions with, e.g.

   ```
   >>> dict = fromList english5
   >>> member "blurb" dict
   True
   ```

4. `delete :: (Ord a) ⇒ a → Tree a → Tree a`, which removes an element from a binary search tree (if indeed it is a member), returning the updated search tree. (This requires a bit more thought than insertion!)

Exercise 4.5 (*Mandatory*: Converting trees to/from lists, `Tree.hs`).
This exercise explores alternatives to `elems` and `fromList` from Exercise 4.4.
Write the following functions:

1. `inOrder :: Tree a → [a]`, which returns the elements of a binary search tree in ascending order. Note: the type of this function forbids defining it as `inOrder xs = sort (elems xs)`.

   For the tree at the top of page 2: `inOrder tree` ⟹ `"abcdfg"`.

2. `fromAscList :: [a] → Tree a`, which, when given a list that is already *sorted in ascending order* constructs a binary search tree that is *balanced*—that is: for every node the height of both children should differ by at most one. (Using `fromList` from Exercise 4.4 on a sorted list probably produces tree that is very *unbalanced*.)

3. `breadthFirst :: Tree a → [a]`, which returns the elements of a tree in left-to-right, *breadth-first* order. That is, first the value contained at the root node, then all the values contained in its children, then all the values in its grandchildren, and so on.

   For the tree at the top of page 2: `breadthFirst tree` ⟹ `"cafbdg"`.

   This exercise may be more difficult! You cannot immediately jump to the *list design pattern*, but you will have to think for a bit first and use helper functions. If you are stuck, see Hint 3.

[1] `http://www.codehappy.net/wordlist.htm`

Exercise 4.6 (*Mandatory*: Expression trees, `AST.hs`). In the lecture, a start was made defining a data type for *arithmetic expressions*, which is a simple example of an *abstract syntax tree*, a fundamental concept in compiler construction. Recall the definition:

```
data Expr = Lit Integer | Add Expr Expr | Mul Expr Expr
```

It was shown this can also be created with infix constructors:

```
data Expr = Lit Integer | Expr :+: Expr | Expr :*: Expr
infixl 6 :+:
infixl 7 :*:
```

In this exercise, pick either of these representations (the one you like the best), and then do the following. It is easier if you work on the first two subtasks in parallel.

1. Finish `Expr` so that it supports:
   - The arithmetic operations: addition, subtraction, multiplication, and division.
   - Integer constants.
   - An single unknown variable x; so that `Expr` can represent a formula like "$2x + 1$".
2. Write a function `eval :: (Fractional a) ⇒ Expr → a → Maybe a` which evaluates the expression given as the first argument, with the second argument becoming the value for "x" in your `Expr`. In case the expression tries to divide by zero, your function **must not** produce a runtime error but instead return `Nothing`; in all other cases it should return the result.

   So—assuming you picked the first representation for `Expr`— we expect an output like:

   ```
   eval (Lit 37) 5 ⟹ Just 37
   eval (Add (Lit 37) VarX) 5 ⟹ Just 42
   eval (Div (Lit 37) VarX) 0 ⟹ Nothing
   ```

3. *Optional/extra:* Write a function `deriv :: Expr → Expr` which computes the *mathematical derivative* of an expression. So, if an `Expr` represents "$x \cdot x + 1$", the result of `deriv` should be an `Expr` that represents "x+x" or "2x" (or equivalent).

Some tips:

- Start simple: a common mistake made is to tackle an entire problem in one fell swoop. If you can do that, great! If not, work in steps, for example:
  1. Create a `eval :: Expr → Integer`;
  2. Evolve it into `eval :: (Fractional a) ⇒ Expr → a`;
  3. Add support for the variable 'x': `eval :: (Fractional a) ⇒ Expr → a → a`;
  4. Add code preventing division by zero: `eval :: (Fractional a) ⇒ Expr → a → Maybe a`
- Dealing with `Maybe` expressions in recursive functions can be little bit annoying. Soon we will introduce techniques to make this easier.

Exercise 4.7 (*Extra*: Type class instances (worked example), `Pronounce.hs`).
This exercise is a step-by-step example illustrating a simple use of type classes; the amount of code you will have to write is minimal.

1. In Week 1, we implemented a `say :: Integer → String` function. Now, having this function only work on `Integer` is a bit restrictive, so suppose we want to also have this function available for data types `Int`, `Char`, `Float`, `Double`, etc. We are going to create a type class for this, and have already added an instance for `Char`:

   ```
   class Pronounceable a where
     pronounce :: a → String

   instance Pronounceable Char where
     pronounce c = unwords ["the","character", "'"++[c]++"'"]
   ```

   Put your solution of `Say.hs` (or the solution provided to you) in the same directory as `Pronounce.hs`, and load `Pronounce.hs` in GHCi;

2. Create instances of the `Pronounceable` class for the type `Integer` and `Int`, in which the `pronounce` function will produce the same result as the `say` function from Exercise 1.6.

   (Reminder: you can use the `toInteger` function to convert a `Int` to `Integer`)

3. Create an instance of `Pronounceable` for `Double` which uses the `pronounce` instance for `Integer` to put floating point numbers in words, rounded to the first decimal:

   ```
   pronounce 23.0 ⟹ "twenty three point zero"
   pronounce 37.5 ⟹ "thirty seven point five"
   pronounce 3.14 ⟹ "three point one"
   ```

   (You can use the `round` and `truncate` functions to convert real numbers to integers.)

4. We have provided an instance of `Pronounceable` which can be used to pronounce lists of `Pronounceable` things; add an instance that work on tuples `(a,b)` when both `a` and `b` are `Pronounceable`.

Exercise 4.8 (*Extra*: Derived instances of standard type classes, `MyList.hs`).
In Haskell, if you really want, you can write your own list type, and you can get a lot of operations such as comparisons and a pretty printer for free:

```
data MyList a = a :# MyList a | Null
  deriving (Eq,Ord,Show)
```

However, Sometimes these 'free operations' may not be what you want them to be.

1. First of all, write functions `fromList :: [a] → MyList a` and `toList :: MyList a → [a]` which convert between `MyList` and Haskell lists. We will need these for this exercise.

2. Since we derived `Ord`, we get comparisons for free. For example, we can ask to compute `fromList [1,2,3] ≤ fromList [4,5,6]`, which will be `True`. However, the ordering that `≤` gives has something odd. Find two lists `x` and `y` such that `x ≤ y`, but for which it does not hold that `fromList x ≤ fromList y`. What do you think causes this? *(Note: compare the definition of `MyList a` with the one for `List a` on the slides.)*

3. If you type `fromList [1,2,3]` in GHCi, it outputs the raw `MyList` syntax at you. This suffices at first, but it's not very nice; after all for normal lists we get the much nicer `show [1,2,3] ⟹ "[1,2,3]"`.

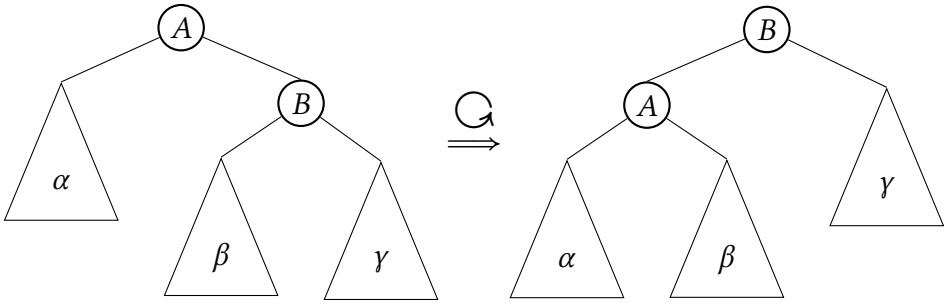
   Create an instance of the `Show` type class for `MyList a` so that:

   ```
   show(fromList [1,2,3]) ⟹ "fromList [1,2,3]"
   ```

   Obviously, this only works if the type `a` is also an instance of `Show`.

Exercise 4.9 (*Extra*: Tree rotations, `RotateTree.hs`).
A problem with naive binary search trees is that they can become unbalanced. One solution is to use *balanced* search trees such as AVL trees or Red-Black trees. These modify the tree after every insert/delete operation using *tree rotations*, which change the *shape* of the tree, but keep the *binary search* property intact. This exercise focuses on that operation.

Rotating a tree rooted at node A switches the position of node A with one of its children B, so that B becomes the new root; re-arranging the other parts of the tree accordingly. Below a *left-rotation* is shown, where A is being replaced by its right child:



Wikipedia has a lovely animation of this operation.[2]

1. Write functions `rotateLeft, rotateRight :: Tree a → Tree a` that rotate a single node to the left or right, as illustrated above. Tip: start by expressing the trees shown above using `Tree` constructors as you did in Exercise 4.3.1.

   If the tree cannot be rotated due to missing children, simply return it as-is.

2. Write a function `rotateToRoot :: (Ord a) ⇒ a → Tree a → Tree a` which finds an element in a binary search tree and moves it to the *root* of the tree by repeated left- and right-rotations.

   ```
   >>> tree = fromList [1..4]
   >>> rotateToRoot 3 tree
   Node 3 (Node 1 Leaf (Node 2 Leaf Leaf)) (Node 4 Leaf Leaf)
   ```

   If the element is not found, you may cause a runtime error (see Hint 2).

3. Instead of using `rotateLeft`/`Right` on a freshly constructed `Node`, we can also use functions `left`/`rightRotatedNode` which directly construct a new node such that:

   ```
   leftRotatedNode key lt rt = rotateLeft (Node key lt rt)
   ```

   Write such functions and use them in `rotateToRoot`.

   (Using a function that has the same type signature as a a data constructor, but performs extra checks or optimizations is a Haskell idiom called a *smart constructor*.)

4. If we use `rotateToRoot` after every successful call to `member`, we can create a tree that places frequently accessed items near the root. Can you think of a drawback of using a 'self-optimizing' tree like that, especially in Haskell?

[2] `https://en.wikipedia.org/wiki/Tree_rotation`

Hints to practitioners 1. In every functional programming language, not getting your parentheses right is really annoying (this is certainly true for the very first functional language, LISP).

If you get an error message like: *"The constructor 'Foo' should have n arguments, but has been given none."*, that probably means you have written something like:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)

foo Node x left right = x
```

Which Haskell will see as a function 'foo' that takes 4 arguments, the first of which not being a valid pattern. The correct way to write it is:

```
foo (Node x left right) = x
```

Also, do not confuse *type classes* with *polymorphic data types*:

```
foo :: Eq a ⇒ Tree a → a
```

Here, the difference between ⇒ and → is very important, but easily missed. For that reason, we prefer to write type classes in parentheses:

```
foo :: (Eq a) ⇒ Tree a → a
```

Hints to practitioners 2. Sometimes a function simply cannot do anything sensible for a certain input. Suppose we want a function that must return the second element of a list. We can do this with a pattern match:

```
secondElement :: [a] → a
secondElement (_:y:_) = y
```

But GHC will complain (with `-Wall`) about missing cases for `[]` and `[x]`. But for those cases, the function does not make sense. To indicate that, we can have it return a nice error message:

```
secondElement :: [a] → a
secondElement (_:y:_) = y
secondElement _ = error "no second element in a list of length ≤ 1"
```

This is a rare exception to the rule that Haskell has no side-effects; but it is also a rather ultimate side-effect: as soon as you *examine the result* of `secondElement`, your program will halt. (Since Haskell is lazy, *examining the result* might not happen when you expect it to. More on this later!)

Hints to practitioners 3. There are at least two ways to perform a *breadth-first* walk of a tree; both of them involve helper functions.

1. You can use a list to maintain a first-in, first-out (FIFO) queue of nodes that have to be processed. Initialize this queue with the root of the tree, and then iteratively take out the first element of the queue, and append any children it may have to the rear of the queue. Repeat this until the queue is empty. (This algorithm was invented by Konrad Zuse in 1945; Wikipedia has more info)

2. More brute-force: you can use the *tree design pattern* to define a function `Tree a → [[a]]` which computes a list-of-lists of elements in a tree: the first list containing the root element, the second its children, the third its grandchildren, etc. In the recursive step, a second helper function will be necessary to combine two of such lists.

Hints to practitioners 4. Observe this data type:

```
data Literal = Lit Int
```

Here, `Literal` is the name of the type, and `Lit` is a type constructor. Essentially, the type `Literal` is just an integer with a fancy name: you write `Lit 5` instead of `5`. This might not seem very useful right now, but it can be. In this case, it can actually be confusing to have to remember *two* names for such a simple data type, and they are usually chosen to be the same:

```
data Literal = Literal Int
```

Which is annoying for us, since it makes it easy to confuse *types* and *expressions*.

But inventing names is hard. In Haskell, we have names for values, functions, types and type variables, data types and their constructors, type classes and their methods , and modules. As a rule of thumb, the wider the scope of an entity, the more care should be exercised in choosing a suitable identifier.

For example, the argument of a function only scopes over the function body:

```
swap :: (a,b) → (b,a)
swap (x,y) = (y,x)
```

Hence it is acceptable to use laconic names such as `x` and `y`. Or would you prefer to read the definition below?

```
swap' (firstComponent, secondComponent) = (secondComponent, firstComponent)
```

Likewise, the type variables `a` and `b` only scope over the signature of `swap`. Again, it acceptable to use short names. Or would you prefer the signature below?

```
swap' ::  (typeOfFirstComponent, typeOfSecondComponent)
       → (typeOfSecondComponent, typeOfFirstComponent)
```

By contrast, the name `swap` is visible in the entire module where its definition resides. And beyond that, if `swap` is part of a module. Are you happy with `swap` or would you prefer `swapTheComponentsOfAPair`?

The same applies to names of data types, constructors, and modules, all of which are potentially globally visible.

Tastes differ, of course. If you intensively dislike the name of a library function or type, you are free to introduce synonyms e.g.

```
type Z = Integer
sort = insertionSort
```

We can also rename module names in qualified imports e.g.
`import qualified BinarySearchTree as Set`.