# Declaratieve Talen

Haskell 2

## 1 Datatype Drill

Haskell is famous for its type system, its type checker and its strongly, statically-typed compilation process. However, up until now you've only encountered predefined types. Using *algebraic data types* it is possible to define new types yourself.

### 1.1 Data type definitions

A newly created type has to be *defined* by specifying all possible *(data) constructors*. Each constructor is a function that can be used to create a value of this type. The different constructors are separated by the `|` symbol. Syntactically, this is done in the following manner:

```
data TypeName = Constructor1 Arg1 Arg2 ...
              | Constructor2 Arg1 Arg2 ...
              | ...
              | ConstructorN Arg1 Arg2 ...
```

Below are some examples. The data type definition often follows pretty easily from its natural language definition.

- A boolean can be either `True` or `False`.

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

- The data type `IntTree` for a tree of `Int`s. Such a tree can either be empty or consist of a node holding a value (an `Int`), and a left and right subtree.

  ```
  data IntTree = EmptyIntTree
               | IntTreeNode Int IntTree IntTree
  ```

- Like in Prolog, a list can either be an empty list, or an element appended with a (sub-)list.

  ```
  data List a = Nil | Cons a (List a)
  ```

or, in its infix form:

```
data [a] = [] | a : [a]
```

Because we don't know the type of the elements yet, and we wish that this data type definition works for *any* type of elements inside the list, we have to specify the desired type of the element as a *parameter* of our resulting list data type definition. Data types that use this kind of *parameter* are called *(parametrically) polymorphic*. In the above definition `a` represents the type parameter. Thus, `[Int]` is the Haskell type of a list of integers, `[Bool]` is the type of a list of bools etc.

Now it's your turn! Write the following data type definitions.

- Generalise the `IntTree` type to a polymorphic data type `Tree a`.
- Create the datatype `ChessPiece` that represents a piece on a chessboard (e.g. a bishop, a knight, ...). A chess piece has a name (of type `String`) and has a function that maps it current position (of a given type `Pos` whose definition should not matter) to a list of positions that can be reached with a valid move for that piece.
- [**note: this is an excerpt from a previous exam question!**] Create a datatype `Station a b` representing a workstation in a factory that uses elements of `a` as input and produces elements of type `b`. A workstation is either a *machine* or a series of workstations that are used one after the other. A *machine* has a list of requirements of type `[(a,Int)]` containing the desired amount of the resource of type `a`, and an object of type `b` that it produces.

## 1.2 Using self-declared data types

- Define a function `mapIntTree :: (Int -> Int) -> IntTree -> IntTree` which applies a function to each `Int` value in the tree. The function should return a tree with the same structure as the given tree, but the values should be the results of the function applications.
- Define a function `intTree2list :: IntTree -> [Int]` which traverses the tree in depth-first pre-order (i.e., the value in a node comes first, then the values in its left subtree, and then the values in its right subtree) and returns a list of the values.
- Make `IntTree` an instance of the `Eq` type class. Two trees are equal if and only if they have the same shape and all values in corresponding positions are equal as well.

- Generalise the `mapIntTree` function to `mapTree :: (a -> b) -> Tree a -> Tree b` which applies a function to each value in the tree. The function should return a tree with the same structure as the given tree, but the values should be the results of the function applications.
- The `Functor f` type class consists of one simple function: `fmap :: (a -> b) -> f a -> f b`. Try to understand its type and make `Tree` an instance of this type class.

- Generalise the `intTree2list` function to `tree2list :: Tree a -> [a]` which traverses the tree in depth-first pre-order and returns a list of the values.
- Define a function `foldTree :: (a -> b -> b) -> b -> Tree a -> b` analogously to the function `foldr` from the Prelude (recall from the previous session). **Hint**: you could use `tree2list` and `foldr` in your implementation of `foldTree`.
- Make `Tree a` an instance of the `Eq` type class. **Note**: unlike before, two `Tree`s are equal when they contain the same elements, *regardless of the structure or order*! The number of elements must also be equal.
- Try to make your `Tree a` instance of `Eq` work for types `a` which only have an instance of `Eq a`, but not necessarily of `Ord a`. **Hint**: look up the documentation of the `Data.List` module and have a look under the section *"Set" operations*.
- **Extra**: define a function `tree2listBF :: Tree a -> [a]` similar to `tree2list`, but instead of traversing the tree in depth-first pre-order, it should traverse the tree in *breadth*-first pre-order.

- [**note: this is an excerpt from a previous exam question!**] Given the `Station a b` type, write the following functions:
  - `machine :: [(a, Int)] -> b -> Station a b`, that constructs a `Station` based on its resources-to-target specification.
  - `combine :: [Station a b] -> Station a b`, that combines multiple stations into one large station.

# 2 Sequences

Define a type class `Sequence a` which consists of the functions `next` and `prev` to query the next and previous element in the sequence.

```
Main> prev 't'
's'

Main> next 'z'
*** Exception: no value after 'z'

Main> next (2 :: Int)
3
```

Define the instances of `Sequence a` for `Int`, `Char`, and `Bool`.

Make two subclasses of this type class: `LeftBoundedSequence a` and `RightBoundedSequence a`. The former has a function `firstElem` and the latter has a function `lastElem` to query the first and the last elements in the sequence.

```
Main> firstElem :: Char
'a'

Main> lastElem :: Bool
True

Main> firstElem :: Int
-9223372036854775808
```

Define instances for these two classes for `Int`, `Char`, and `Bool`.
**Hint**: have a look in the `Data.Char` module.

## 3 List Comprehensions

Rewrite the following functions using *list comprehensions*. Give your functions names that don't conflict with the names of the built-in functions.

- `map :: (a -> b) -> [a] -> [b]` applies a function to each argument in the list.
- `filter :: (a -> Bool) -> [a] -> [a]` retains only elements in the given list for which applying the predicate function returns `True`. The function should preserve the original order of the elements.
- `concat :: [[a]] -> [a]` appends a list of lists into one list.

Rewrite the following functions using `concat`, `map`, and `filter`.

- ```
  lc1 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
  lc1 f p as = [f a | a <- as, p a]
  ```
- ```
  lc2 :: (a -> b -> c) -> [a] -> (a -> [b]) ->
         (b -> Bool) -> [c]
  lc2 f as bf p = [f a b | a <- as, b <- bf a, p b]
  ```

- ```
  lc3 :: (Int -> Int -> Int -> a) -> Int -> [a]
  lc3 f n = [f a b c | a <- [1..n]
                     , b <- [a..n], even a
                     , c <- [b..n]
                     , a * a + b * b == c * c]
  ```

# 4 Function Chaining

It is very common in a program to apply multiple functions one after another. For example, to apply `f`, `g` and `h` to a value `x`, one could write:

```
myFunc x = h (g (f x))
```

These parentheses become cumbersome very quickly. The solution to this, is to introduce a higher-order function '`.`' that "chains" two functions after each other.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Using this operator, we can now write `myFunc` much more elegantly:

```
myFunc x = (h . g . f) x
```

This means that we apply `f` to `x`, apply `g` to the result, and finally apply `h` to this result. Note that the parentheses here are necessary, as `h . g . f x` is actually parsed as `h . g . (f x)`, which means something completely different.

Intermediate Haskell programmers don't like to write parentheses, so they have come up with a way to omit these parentheses. The solution is the `$`-function.

```
($) :: (a -> b) -> a -> b
```

This function seems completely pointless as it just represents function application. However, due to how it is parsed, this operator can separate the function and argument without the need for parentheses. We can now rewrite `myFunc` to:

```
myFunc x = h . g . f $ x
```

Notice that the function can be defined by just chaining `f`, `g` and `h` together. The argument `x` is now redundant. Thus, the function myFunc can be defined as:

```
myFunc = h . g . f
```

That is, by evaluating `h` after `g` after `f`.
```

**Assignment** Write a function `applyAll :: [a -> a] -> a -> a` which applies a list of functions to a value, one after the other.

```
Main> applyAll [(+ 2),(* 2)] 5
12

Main> applyAll [(: []). sum, filter odd] [1..8]
[16]
```

Write a function `applyTimes :: Int -> (a -> a) -> a -> a`, which applies a function a given number of times to a value. You should use only two explicit arguments in your code.

```
Main> applyTimes 5 (+ 1) 0
5

Main> applyTimes 4 (++ "i") "W"
"Wiiii"

Main> applyTimes 0 (error "Error!") 3.14
3.14
```

As a variation on this theme, write a function `applyMultipleFuncs :: a -> [a -> b] -> [b]`, which takes an argument and a list of functions, and applies these functions to the given argument.

```
Main > applyMultipleFuncs 2 [(*2), (*3), (+6)]
[4,6,8]
```