Exercise 3

NTNU

TDT4165 fall 2018

1 The power of types: Hoogle

1.1 Type inference

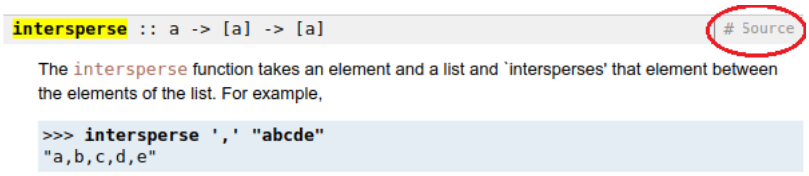
The type system in Haskell is quite powerful, as you might have discovered. Even if we do not include a type signature, we can ask GHC to tell us its type and get an answer. This is called **type inference**.

```
Prelude> import Data.Char
Prelude Data.Char> :t ord
ord :: Char -> Int
Prelude Data.Char> let fun = map ord
Prelude Data.Char> :t fun
fun :: [Char] -> [Int]
```

GHC infers the type of the function **fun** from the function **ord** that is passed to map. Since map has type **(a → b)** → **[a]** → **[b]**, and ord has type **Char** → **Int**, **map ord must have type** **[Char]** → **[Int]**.
For the curious, a description of a type inference algorithm can be found here (not curriculum): `http://www.cs.tau.ac.il/~msagiv/courses/pl14/chap6-1.pdf`

1.2 Hoogle

A consequence of the type system and type inference, is that it is possible to create a database where you can **search for functions based on their arguments and return types**. It is not only possible, it exists!
At `https://www.haskell.org/hoogle/`, you can search through standard libraries using type signature, function name, type name and much, much more. You can also search for keywords. Clicking on the name of the type class or function, then on Source, will bring you to its declaration in source.



2 Bounded parametric polymorphism

Last week, we saw functions like **id :: a** $\rightarrow$ **a** that are polymorphic across *all* types, but we've also seen functions like **show :: Show a** $\Rightarrow$ **a** $\rightarrow$ **String** that are polymorphic across all types that are "showable". Here **Show** is a *type class*, and any type belonging to **Show** must implement the function **show**.

2.1 Type classes and instances

Let us define a function and ask GHC to tell us what types we are dealing with:

```
eq x y
    | x == y    = x
    | otherwise = y

> :t eq
> eq :: Eq a => a -> a -> a
```

The **Eq** type class simply allows checking if two elements are equal or not:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  x == y = not (x /= y)
  x /= y = not (x == y)
```

The last two lines give default implementations of equality (==) in terms of inequality (/=), and vice versa. This means a *minimal complete definition* of **Eq** requires only one of them. For primitive types such as Int, the **Eq** instance is has a low-level implementation. However, for types we have declared ourselves, we may need to implement the instance ourselves. Consider the type constructor **Maybe** defined in exercise 2. We would like **Maybe a** to inherit equality from **a**, meaning that **Just x** equals **Just y** if and only if **x** equals **y**:

```
data Maybe a = Just a | Nothing

instance Eq a => Eq (Maybe a) where
    Nothing == Nothing   = True
    (Just x) == (Just y) = x == y
    _ == _               = False
```

The case of inheriting a type class is very common, and there is a shorthand to avoid writing out the instance explicitly, namely by *deriving*:

```
data Maybe a = Just a | Nothing deriving Eq
```

2.2 List functions

Implement the functions **listSum**, **listProduct**, **listConcat**, **listMaximum** and **listMinimum**, that sums a list, takes the product of a list, concats a list of lists, finds the largest and smallest element in a list, respectively. **Use pattern matching on list and recursion.** Some useful pattern matching syntax:

```
-- [] matches an empty list
example [] = ...
-- here, lst refers to the whole list, x refers to the
    first element, and xs refers to the tail of the list
example lst@(x:xs) = ...
```

3 More higher-order functions: folds

The functions you implemented in task 2, are all ways of reducing a list to a single value, and if you were implementing these as a for-loop you'd have a single state that you'd update on each iteration. This pattern of iteration is captured by a *fold*. Now, lists can be iterated over either from the top or from the bottom (or left to right if you prefer). Laziness makes it more efficient to iterate from the right (ask us about this!), and we'll therefore have a look at **foldr**:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op acc xs = ...

foldr (+) 0 [1,2,3] -- example of a step-wise reduction
==> 1 + (foldr (+) 0 [2,3])
==> 1 + (2 + (foldr (+) 0 [3]))
==> 1 + (2 + (3 + 0))
==> 1 + (2 + 3)
==> 1 + 5
==> 6
```

By how our lists are defined, the head is always what we see first. What makes foldr "folding from the right" is how the actual addition is being done. In the above example, the first addition performed is "3 + 0". The variable `acc` is the *accumulator* and is the state if you were to do a for-loop.

3.1 Foldable

Lists aren't the only data-structure that can be reduced to a single value, and we'll see how it can be done for (binary) trees in the next section. The underlying type class for things that are "foldable" is **Foldable**:

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

It is an immediate generalization of **foldr** for lists, we've only replaced `[a]` by `t a` where `t` is another type constructor. To put this into perspective, remember that as a type constructor `[]` takes a type `a` and constructs the type `[a]` i.e. *the lists with elements in* `a`. Likewise, `Maybe` takes any type `a` and constructs the type `Maybe a`. (Is Maybe foldable?) The `t` in our **Foldable** class is any such type constructor.

In this week's exercise you are asked to implement an instance of **Foldable** for lists, and to define a set of related functions.

3.2 Binary trees

The goal of this section is to understand how we may represent binary trees (called `Tree2` below) in Haskell and make our Tree type an instance of **Foldable**. Before reading on, try to define your own binary tree type, and please ask if you have no clue! (There are many ways, e.g. should the branches have values?)

First of all, a *binary* tree is one where branches split into *two*, and you could think of our representation of lists as "unary trees" (`Tree1` below) which only have one, continuous branch. With this analogy:

```
data List a = Cons a (List a) | Nil -- same as a : [a] | []
data Tree1 a = Branch a (Tree1 a) | Leaf1 a
data Tree2 a = Branch (Tree2 a) a (Tree2 a) | Leaf a
```

Now, as with lists, we can make sense of summing all the values of a **Tree2**, or taking their product, maximum or minimum—even concatenating them makes sense. This should be an indication that **Tree2** can have a **Foldable** instance. As a hint, your implementation should produce the following reductions:

```
foldr (+) 0 (Leaf 1)  ==> 1 + 0
foldr (+) 0 (Branch (Leaf 1) 2 (Leaf 3))
==> foldr (+) (2 + foldr (+) 0 (Leaf 3)) (Leaf 1)
==> 1 + (2 + foldr (+) 0 (Leaf 3))
==> 1 + (2 + (3 + 0)) ...
```

The skeleton for this task can be found in src/Tree.hs. Make the tree an instance of **Foldable**. Writing the three predefined functions might help you reason about this task, but they are not mandatory.

4 The Num type class

4.1 Num Complex

A Complex type is already defined in the source file. It is a member of the Show and Eq type classes. Make it an instance of the Num type class by implementing the functions below. This is the minimal complete definition of Num.

```
(+), (*) :: Num a => a -> a -> a
abs, signum :: Num a => a -> a
fromInteger :: Num a => Integer -> a
(-) :: Num a => a -> a -> a
or
negate :: Num a => a -> a
```

This is the minimal implementation of a type for it to be included in the Num type class. An explanation of the functions and rules for complex numbers, are outlined below.

4.2 (+)

Addition of two complex numbers.

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

4.3 (*)

Multiplication of two complex numbers.

$$(ac - bd) + (ad + bc)i$$

4.4 abs

Absolute value of a complex number.

$$|a + bi| = \sqrt{a^2 + b^2}$$

4.5 signum

The sign of a complex number.

$$sign(z) = \frac{z}{|z|}$$

4.6 fromInteger

Conversion from an Integer to the complex representation of this integer. For a Double, **fromInteger 7** will return **7.0**.

4.7 (negate | (-))

Implement either negate (unary minus), which returns the negation of the number it receives, **or** (-) that performs subtraction.

$$(a - bi) + (c - di) = (a - c) + (b - d)i$$

5 Making your own type classes

5.1 Record syntax

Using record syntax, we can create types with named fields.

```
--record syntax
data Person = Person { firstName :: String
                     , lastName :: String
                     , socialSecNr :: Int
                     , zipCode :: Int
                     } deriving (Eq, Show)
```

Above, we have defined a type **Person**, using record syntax. The fields are easy to access.

```
Prelude> let person1 = Person {firstName = "Jane", lastName
    = "Doe", socialSecNr = 1234567890, zipCode = 7034}
Prelude> firstName person1
"Jane"
```

The person type defined above, is a derived instance of Show and Eq. (==) will then result in a comparison of every field and False if they are not all equal. This is unfortunate, since Jane might move or change her name. To amend this, we write our own instance of Eq.

```
instance Eq Person where
    (==) p1 p2 = socialSecNr p1 == socialSecNr p2
--now Jane can move all she wants and still be the same
   person
Prelude> let samePers = person1 {zipCode = 7042}
Prelude> samePers == person1
True
```

5.2 Move type class

Create a type class Move that is a subclass of Pos. A movable type should be able to be moved and has a position where it belongs.

5.3 Car

Create a type **Car**, using record syntax, and make it an instance of Eq, Pos and Move.

5.4 Key

Create a type **Key**, using record syntax, and make it an instance of Eq, Pos and Move.

5.5 free

Implement a function **free :: Move a ⇒ a → Bool** that checks if an object is at the place it belongs.

5.6 carAvailable

Implement a function **carAvailable :: Car → Bool** that checks if the car and its keys are free.

5.7 distanceBetween

Implement a function **distanceBetween :: Pos a ⇒ a → a → (Position or Int)** which calculates the distance between two positions. You are free to choose your own definition of *distance*.

6 Theory questions

- When implementing **treeMaximum** in src/Tree.hs, do you have to return a Maybe? Why or why not?
- After making **Tree a** an instance of **Foldable**, we can sum it by calling **sum $ Branch (Leaf 56) 2 (Leaf 23)** (:l src/Tree.hs and try it out!). This also applies to **maximum** and several other functions. How do we get these functions "for free"?
- Can we make Complex an instance of Ord? The minimal complete definition of Ord is **compare** — **(<=)**, meaning that you have to implement either compare or (<=). Use Hoogle to inspect these functions in more detail, if needed. Provide the reasoning behind your answer.
- Load your file and write **negate (Complex 3 4)**, if you implemented (-) in 2.7. If you implemented negate, type **(Complex 3 4) - (Complex 2 7)**. What happens? Why does this happen?
- Try to make a Pos instance of the type synonym Position. What kind of error message do you get? From the error message, do you think it's possible to achieve this?

- You used record syntax when creating your types, and you might have run into issues if you tried to name fields identically. If not, try to run the following code:

```
data Employee = Employee { name :: String
                         , employeenr :: Int }

data Student = Student { name :: String
                       , studentnr :: Int }
```

  What kind of error message do you get? Why do you get this error message? Hint: Remove one of the entries, load the file and ask for the type of a field. What's the scope of the entry?