14 Foldable and Traversable

Exercise 14.1 (*Warm-up:* Type derivation, `PolyTypes.hs`). Derive the *most general type* of the following polymorphic functions.

```
lift0 f x           = f x
lift1 f g1 x        = f (g1 x)
lift2 f g1 g2 x     = f (g1 x) (g2 x)

deMorgan quantor p   = not . quantor (not . p)
```

Check your answers using GHCi!

Tip: `deMorgan` is certainly more difficult than the rest; it may help to remember that `\f . g = \x→f (g x)`, and that the type of `not` is `Bool → Bool`.

Exercise 14.2 (*Warm-up:* folding and traversing, `Demo.hs`).
Consider the following expressions:

```
expr1 = Just True
expr2 = Just False
expr3 = Nothing
expr4 = Just 42
expr5 = []
expr6 = [1..5]
expr7 = [True,True,False]
expr8 = Map.empty
expr9 = Map.fromList [("Rinus", 7.5), ("Peter", 8.2), ("Ralf", 6.8)]
```

1. Since lists, `Maybe`, and the `Data.Map` type are *foldable*, we can use `foldr` on all of these expressions! What happens for each of these expressions if we compute:
   - `foldr (+) 0 expr`
   - `foldr (&&) True expr`
   - `foldr (||) False expr`
2. Because type constructors are instances of `Traversable`, and `IO` and `Maybe` are `Applicative`, we can also try the following expressions:
   - `traverse (\x→if x >= 6 then Nothing else Just x) expr`
   - `traverse print expr`
   - `traverse (\x→putStr "enter replacement for " >> print x >> getLine) expr`

   What do these compute?

(Practical note: for monads, `mapM` and `traverse` are functionally equivalent.)

Exercise 14.3 (*Warm-up:* instances, `Instances.hs`, `Result.hs`).

1. Remember the `Result` data type:

   ```
   data Result a = Okay a | Error [String]
   ```

   Show how `Result` be made foldable by creating the instance `Foldable Result`. Either define `foldr` or `foldMap` (as you prefer). We have already provided a `Traversable` instance. As a reminder, the types of these functions will be:

   ```
   foldr    :: (a → b → b) → b → Result a → b
   foldMap  :: (Monoid m) ⇒ (a → m) → Result a → m
   traverse :: (Applicative f) ⇒ (a → f b) → Result a → f (Result b)
   ```

   You can re-use anything from Exercise 11.5. Note that since `Result` is container holding either zero or one value, it may feel that there is not much to "fold" or "traverse" here!

2. Consider the data type used for binary search trees:

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

   What would be the type of `foldMap`? Create an instance `Foldable` for `Tree` by giving the definition of `foldMap`.

   There are multiple ways to fold a tree, choose the one that makes the most sense to you. (You can also try to define `foldr`, but you will discover that its definition is less nice.)

3. By implementing `Foldable`, you get a lot of functions for free that you had to implement explicitly in Exercise 4.3 and 4.4. Use these to test your instance of `Foldable Tree` by trying to compute:

   ```
   >>> length assistants
   9
   >>> elem "Marc" assistants
   False
   >>> foldr1 (\x y→x++", "++y) assistants
   "Bram, Cassian, Jen, Mario, Patrick, Quinten, Rico, Sander, Willem"
   ```

   The order in which these names will depend on your implementation of `foldMap`.

4. Create the instance `Traversable` for `Tree` by defining `traverse`; this also requires you to create an instance `Functor Tree`.

   Since you don't know which `Applicative` is being used in the definition of `traverse`, you are forced to use `pure`, `(⟨*⟩)` and/or `(<$>)` to produce results of the correct type.

   Again, the `IO` and `Maybe` monads are easy ways to test your instance; e.g. try:

   - `traverse putStrLn assistants` to print the assistants listed in the binary tree.
   - `traverse (\name→putStrLn ("Who should succeed "++name++"?") >> getLine) assistants` to create a new tree. This also highlights a caveat: is the output of this operation guaranteed to always be *binary search tree*?

Exercise 14.4 (*Mandatory*, Traversing expressions, `TraverseExpr.hs`).
We have worked with several expressions, represented by a datatype such as

```haskell
data Expr var = Var var | Lit Integer | Op BinOp (Expr var) (Expr var)
data BinOp    = Add | Sub | Mul | Div
```

In this case the `Expr` is parameterized by the type of variables, and we use the same contructor for all binary operations.

1. Create instances of `Functor`, `Foldable`, and `Traversable` for `Expr` by giving definitions of `fmap`, `foldMap` and `traverse`.

   As a reminder, this is essentially the same exercise as finding non-trivial instances for the following function types:

   ```haskell
   fmap     :: (a → b) → Expr a → Expr b
   foldMap  :: (Monoid m) ⇒ (a → m) → Expr a → m
   traverse :: (Applicative f) ⇒ (a → f b) → Expr a → f (Expr b)
   ```

   Explain in plain and simple English what it *means* to fold an expression. Is all information used or are some bits ignored?

2. Using functions from the `Foldable` class, define a function `allVars :: (Ord a) ⇒ Expr a → [a]` that outputs a list of all the unique variable names used in an expression; e.g.

   ```haskell
   ⋙ allVars (Op Add (Op Mul (Var "x") (Var "y")) (Var "x"))
   ["x","y"]
   ```

   An overview of all available functions in the `Foldable` class can be found here: https://hackage.haskell.org/package/base/docs/Data-Foldable.html.

3. In a compiler it often makes sense to use integer indices for variables, instead of strings, because comparing integers is much faster and they are easier to use as indices in an array.

   To represent a mapping from variables names to indices we can use *association lists* or `Data.Map`. We have also needed this concept in Exercise 7.6 to map values to Huffman codes, and Exercise 11.6, where it was to associate variables with values. One way to do this is to first scan the expression for all variables, give every unique variable its own index, and then use this table to perform the substitution.

   But in this case, we would like to scan the expression *just once*: the mapping from variables to indices can be constructed *while* replacing names with indices at the same time.

   In pseudo-code:

   ```
   table := []
   for every (Var name) in Expr:
       if name not in table:
           index := sizeof table
           add "name->index" to the table
       replace (Var name) with (Var table[name])
   ```

   Handling such a *stateful* programming task is of course a job for the *state monad*.

So, start by writing a *state monad action*:

```
renameVar :: String → State [(String,Int)] Int
renameVar name = do ...
```

the converts *a single variable* into an integer index. If the variable is already indexed, its index should be returned; otherwise a new index must be generated and added to the state as well.

Here we are using the state monad from the `transformers` library. The type `State` reprsents a stateful computation, and has two arguments: the type of the state, and the result of the computation. (If you are stuck with the state monad, see Hint 1)

(If you want to use `Data.Map`, you can also use `renameVar :: String → State (M.Map String Int) Int`

4. Define a function `indexVars :: Expr String → Expr Int` **that uses** `renameVar` and `traverse`, to convert all variables in an expression from strings to integers in the manner defined by the pseudo-code above:

   ```
    ⋙ indexVars (Op Add (Op Mul (Var "x") (Var "y")) (Var "x"))
   Op Add (Op Mul (Var 0) (Var 1)) (Var 0) --- your indices for "x" and "y" may differ
   ```

   Tip: you can use a helper `renameAllVars :: Expr String → State [(String,Int)] (Expr Int)`, and then "run" that in the state monad using `evalState :: State s a → s → a` to obtain `Expr Int` from the state monad.

Exercise 14.5 (*Extra:* Traversal-like functions, `TraverseExpr.hs`, `Lenses.hs`).
(This exercise explores how traversals and lenses work. For the "user perspective" see Exercise 14.7)
In exercise 14.4 we saw a way to traverse all the *variables* in an expression tree making an instance of `Traverse`. We can make this more explicit by giving a it more specific name:

```
traverseVars :: (Applicative f) ⇒ (a → f b) → Expr a → f (Expr b)
travelseVars = traverse
```

But expressions (like many data structures) contain more than just one type of object. In this case, we have binary operations and literals too!

1. Define a function

   ```
   traverseLits :: (Applicative f) ⇒ (Integer → f Integer) → Expr a → f (Expr a)
   ```

   That instead of traversing all *variables*, traverses all *literals* in an expression *(Tip: start by copying the definition of* traverse *from Exercise 14.4 and modifying it slightly).*

2. Define a function

   ```
   traverseBinOps :: (Applicative f) ⇒ (BinOp → f BinOp) → Expr a → f (Expr a)
   ```

   that traverses all *binary operations* in an expression.

3. The three functions we have: `traverseVars`, `traverseLits`, `traverseBinOps` all are examples of *traversals*, and in fact we can give their type as:

```
traverseVars   :: Traversal (Expr a) a
traverseLits   :: Traversal (Expr a) Integer
traverseBinOps :: Traversal (Expr a) BinOp
```

The file `Lenses.hs` contains the function:

```
listOf :: Traversal s a → s → [a]
```

which uses a *traversal* to collect the traversed items in a list.

Import this file, and use the function `listOf` to produce a list of all variables, a list of all literals, and a list of all binary operators in an expression:

```
⋙ listOf traverseVars (Op Add (Lit 42) (Op Mul (Lit 5) (Var "x")))
["x"]
⋙ listOf traverseLits (Op Add (Lit 42) (Op Mul (Lit 5) (Var "x")))
[42,5]
⋙ listOf traverseBinOps (Op Add (Lit 42) (Op Mul (Lit 5) (Var "x")))
[Add,Mul]
```

4. The file `Lenses.hs` also contains the function:

   ```
   mapOf :: Traversal s a → (a → a) → (s → s)
   ```

   that instead of *viewing* an expression, will modify it.

   Use it, and the appropriate traversal, to perform the following computations:

   - Multiply every literal in an expression by 23.
   - Add an underscore to every variable name.
   - For every multiplication and addition, flip the order of the operands.

Exercise 14.6 (*Extra*: Using foldable, `Folds.hs`).

1. Use `foldr` to implement a function `mySum :: (Foldable t, Num a) ⇒ t a → a` that calculates the sum of the elements of a container. *Tip: whenever you are intimidated by an abstract looking type,* 'start simple!'*, and first create* `mySum :: [a] → a`*.*

2. Use `foldr` to implement a function `myLength :: (Foldable t) ⇒ t a → Int` that calculates the number of elements in a container.

3. The mean value of a container is the sum of the elements divided by the number of elements. In Haskell we can implement this as

   ```
   myMean :: (Foldable t) ⇒ t Integer → Double
   myMean xs = fromInteger (mySum xs) / fromInteger (myLength xs)
   ```

   This function makes two passes over the container. Re-implement it using `foldr` to make only a single pass over the container.

   Compare the performance of your one-pass version with the definition given above (remember you can get timing information by using `:set +s`)—it should be twice as fast.

(Optional/extra: this exercise can also be made by using `foldMap` *instead of* `foldr`*; in that case finding the solution lies in finding the appropriate* `Monoid` *instance to use, see Hint 2)*

Exercise 14.7 (*Extra*: using lenses, Lenses.hs, Person.hs).
While the machinery that goes into building lenses is quite abstract (because it has to be very general), actually using them is much more intuitive and they are popular in "real world programs". Consider again the Person database from Exercise 2.6:

```
data Person = Person { name::String, age::Int, favouriteCourse::String }

elena, peter, pol :: Person
elena  =  Person {name="Elena", age=33, favouriteCourse="Functional Programming"}
peter  =  Person {name="Peter", age=57, favouriteCourse="Imperative Programming"}
pol    =  Person {name="Pol",   age=36, favouriteCourse="Object Oriented Programming"}
```

Accessing the fields of such a record is an excellent job for lenses.

1. We have already provided a lens for the name of students:

   ```
   name' :: (Functor f) ⇒ (String → f String) → (Person → f Person)
   name' f p = (\x→p{ name=x }) <$> f (name p)
   ```

   or, more simply, we can write the type as:

   ```
   name' :: Lens Person String
   ```

   As a reminder, the notation `p{name=x}` means to produce a new record where the field `name` gets the value `x`, but all other fields are copied from `p`. It is syntactic sugar for:

   ```
     Person {name=x, age=age p, favouriteCourse=favouriteCourse p}
   ```

   or equivalently:

   ```
     Person x (age p) (favouriteCourse p)
   ```

   Copying the pattern for `name'`, create additional lenses:

   ```
   age'       :: Lens Person Int
   favCourse' :: Lens Person String
   ```

   (Practical note: this is a purely mechanical job—after this exercise you can create lenses for every record type by just repeating this boilerplate. GHC extensions exist that automate this.)

2. We can now use these lenses:

   ```
     >>> view name' pol
     "Pol"
     >>> view favCourse' pol
     "Object Oriented Programming"
     >>> set favCourse' "Hacking in C" pol
     Person {name = "Pol", age = 36, favouriteCourse = "Hacking in C"}
   ```

   Use `mapOf` and the appropriate lens to increment the age of Frits by 2 years.

3. Using `mapOf` and lens composition involving `each` and/or `when`, create expression for the following tasks:

   - increment the age of each student by two;

- promote all the students (prefix `"dr ."` to their name);
- promote all students named "Frits";

4. Using `foldMapOf`, create a function `sumOf :: (Num a) ⇒ Traversal s a → s → a` that sums all elements found by a traversal. This can be used to create a traversal-based averaging function:

   ```
   meanOf :: (Fractional n, Integral a) ⇒ Traversal s a → s → n
   meanOf trav x = fromIntegral (sumOf trav x) / fromIntegral (lengthOf trav x)
   ```

   (Extra: You can also try to define a more efficient version of `meanOf` in the spirit of Exercise 14.6—this will require defining it in terms of `foldMapOf` directly and using Hint 2).

5. Using `meanOf` and lens composition, create an expression computing the average age of all students; then create an expression computing the average age of all students whose favourite course is Functional Programming.

6. Using lenses, promote all students whose favourite course is Functional Programming.

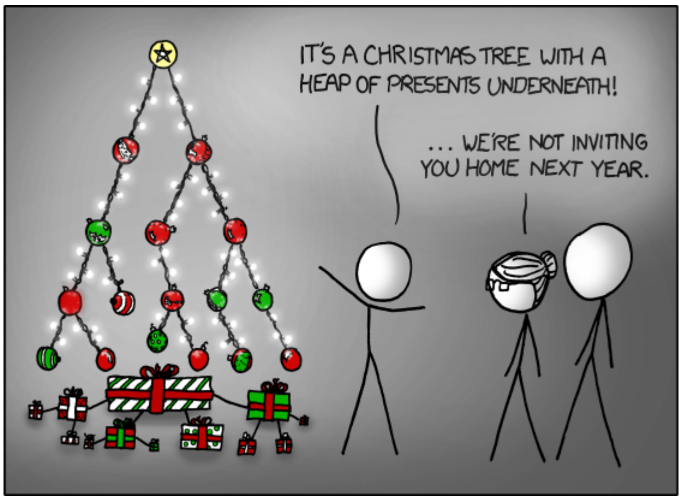
Exercise 14.8 (*Extra:* Finding instances for types, `FindDefs.hs`).
For each of these two function types, find an implementation matching its type:

```
bindSt  :: (a → s → (b,s)) → (s → (a,s)) → (s → (b,s))
andThen :: ((a → r) → r) → ((b → r) → a → r) → (b → r) → r
```

Then aswer the following questions:

- Although these types look intimidating, do they resemble functions you have seen before?
- Are these types the *most general type* for the definitions you found? You can check your answer by removing the type declaration from the functions and using `:type` in GHCi.

Enjoy the holidays!


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

https://xkcd.com/835

Hints to practitioners 1.
Remember that in addition to `return :: a → State s a`, there are the *state monad actions*:

```
  get    :: State s s
  put    :: s → State s ()
```

that can be used to *retrieve* and *update* the state inside the monad. (See Hint 1)

As an example on how to use `get` and `put`:

```
reverseVars :: State [(String,Int)] Int
reverseVars = do vars ← get
                 put (reverse vars)
                 return (length vars)
```

is a *state monad action* that returns the number of variable bindings known in the state monad, but as a side effect also reverses all the variable bindings.

Hints to practitioners 2.
During the lecture, `myLength` and `mySum` were already given away (using `foldMap` and monoids):

```
mySum :: (Num a, Foldable t) ⇒ t a → a
mySum = getSum . foldMap Sum

myLength :: (Foldable t) ⇒ t a → Int
myLength = getSum . foldMap (\_ → Sum 1)
```

I.e. this uses `foldMap` and the `Sum` monoid (defined in `Data.Monoid`).

To create `myMean`, you can make use of the fact that the following monoid instance exists in `Data.Monoid` as well:

```
  instance (Monoid a, Monoid b) ⇒ Monoid (a, b) where
    (x1,y1) <> (x2,y2) = (x1<>x2, y1<>y2)
    mempty             = (mempty, mempty)
```