Exercises, Week 46 (10–14 Nov, 2025)

DM580: Functional Programming, SDU

Learning Objectives

After doing these exercises, you will be able to:

- Implement your own functor, applicative, and monad instances.
- Prove basic properties about functors, applicatives, and monads.
- Program with functors, applicatives, and monads.
- Use monads to implement programming language interpreters.

1 Trees are Functors (12.5.1 from the Book)

Define an instance of the `Functor` class for the following type of binary trees that have data in their nodes (only):

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

2 User Validation with Functors and Applicatives

Say we are implementing a validation procedure for user input.

The validation procedure should work with the following `Result` type which, if validation fails, returns a list of validation errors.

```
data Result a
  = Err [String]
  | Result a
  deriving Show
```

The validation procedure should validate the fields of `User` records:

```
data User = User
  { name  :: String
  , email :: String
  , age   :: Int }
```

Your validation should run using the following `main` loop and `validate` function (you can copy-paste the following into a Haskell buffer; see ItsLearning for a copy-paste friendly version of this assignment sheet).

```
validate :: User -> Result User
validate User{..} = User
```

```
  <$> checkName name
  <*> checkEmail email
  <*> checkAge age
  where
    checkName = undefined
    checkEmail = undefined
    checkAge = undefined

main = do
  u <- validate <$> (User <$> (putStrLn "Enter name: "  >> getLine)
                          <*> (putStrLn "Enter email: " >> getLine)
                          <*> (putStrLn "Enter age: "   >> readLn ))
  case u of
    Err es -> do
      putStrLn $ "Validation failed. Error(s):"
                 ++ concat (("\n- " ++) <$> es)
                 ++ "\n"
      putStrLn "Please start over."
      main
    Result x ->  putStrLn "Validation succeeded!\n"
```

In this program:

- `(<$>)` is an alias (defined in Haskell's `Prelude`) for `fmap`
- `(<*>)` is applicative application
- `(>>)` is monadic sequencing; e.g.:

```
λ> :t (>>)
(>>) :: Monad m ⇒ m a -> m b -> m b
```

The validation program should work as follows:

```
λ> main
Enter name:
Ursula von der Leyen
Enter email:
ec-president-vdl@ec.europa.eu
Enter age:
67
Validation succeeded!

λ> main
Enter name:
q
Enter email:
r
Enter age:
-1
Validation failed. Error(s):
```

```
- Name must be at least four characters long.
- Please provide at least one first name and surname.
- Email must be at least five characters long.
- Email must contain exactly one '@' symbol.
- Email must contain at least one '.' symbol.
- Age cannot be negative.

Please start over.
Enter name:
```

Solve the following exercises.

2.1 Functor Instance for `Result`

Implement a functor instance for `Result`.

2.2 Applicative Instance for `Result`

Implement an applicative instance for `Result` that accumulates errors.

2.3 Checking Functions

Implement the missing checking function bodies for `checkName`, `checkEmail`, and `checkAge`.

3 Functor Laws

The following function corresponds to the `fmap` instance for `Maybe` provided in Haskell's `Prelude`.

```haskell
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
```

Prove that the function respects the functor laws.

That is, prove that

1. `mapMaybe id = id`

   or, equivalently, for any type a and any `x :: Maybe a`,

   `mapMaybe id x = x`

2. `mapMaybe f . mapMaybe g = mapMaybe (f . g)`

   or, equivalently, for any types a and b, any `x :: Maybe a`, and any functions `g :: a → b` and `f :: b → c`,

   `mapMaybe f (mapMaybe g x) = mapMaybe (f . g) x`

4 Interpreter

This assignment is loosely based on Phil Wadler's influential paper on "Monads for Functional Programming" (Wadler, 1995). The paper is not a part of the curriculum for the course, and it is not necessary to read the paper in order to solve this assignment. Nevertheless, the paper is fairly accessible and quite informative.

Say we are implementing an interpreter for a small language. Since our language has side-effects (failure), and since we expect to be adding more effects, our initial interpreter for the language is monadic, and looks as follows:

```
module Interp where

import Control.Monad

data Expr
  = Lit Int | Plus Expr Expr | Times Expr Expr
  | Tru | Fals
  deriving Show

data Val = B Bool | I Int | Unit
  deriving Show

newtype M a = M { runM :: Either String a }

instance Functor M where
  fmap f m = M $ fmap f $ runM m

instance Applicative M where
  pure = M . Right
  (<*>) = ap

instance Monad M where
  m >>= k = M $ runM m >>= runM . k

err :: String -> M a
err x = M $ Left x

intOf :: Val -> M Int
intOf (I i) = pure i
intOf v = err $ "Expected integer but found: " ++ show v

interp :: Expr -> M Val
interp (Lit i) = pure $ I i
interp (Plus e1 e2) = do
  v1 <- interp e1
  v2 <- interp e2
  I <$> ((+) <$> intOf v1 <*> intOf v2)
interp e = err $ "Error: not implemented!\n  " ++ show e
```

We can run our interpreter by running the monad `M`; e.g.:

```
λ> runM $ interp (Plus (Lit 1) (Lit 2))
Right (I 3)
```

4.1 Implement Cases for `Times`, `Tru`, and `Fals`.

Implement the missing cases of the interpreter to match the following behavior.

```
λ> runM $ interp (Times (Lit 3) (Plus (Lit 1) (Lit 2)))
Right (I 9)
λ> runM $ interp Tru
Right (B True)
λ> runM $ interp Fals
Right (B False)
```

4.2 Extend with Stateful Variables

Our language should also be able to handle assigning to and accessing stateful variables, as well as sequencing expressions.

To this end, we update the syntax of our language:

```
data Expr
  = Lit Int | Plus Expr Expr | Times Expr Expr
  | Tru | Fals
  -- New constructors:
  | Assign String Expr | Access String | Seq Expr Expr
  deriving Show
```

Extend the monad and interpreter to match the following behavior (see below for the definition of `runInterp`):

```
λ> runInterp $ (Assign "x" (Lit 41)) `Seq` (Plus (Access "x") (Lit 42))
Right (I 42)
λ> runInterp $ (Assign "x" (Lit 0))
               `Seq` (Assign "x" (Lit 41))
               `Seq` (Plus (Access "x") (Lit 42))
Right (I 42)
λ> runInterp $ Access "x"
Left "Error: attempted to read uninitialized variable x"
```

To support stateful variables, you will need to extend the monad of your interpreter with a *store* that tracks the state of variables. You are welcome to choose your own representation of monad and store, but here is a recommendation:

```
type Store = String -> Maybe Val

newtype M a = M { runM :: Store -> (Either String a, Store) }
{- Functor, Applicative, and Monad instances elided -}
```

Additionally, you can use the following helpers:

```haskell
overrideSt :: String -> Val -> Store -> Store
overrideSt x v s = \y -> if x == y then Just v else s y

accessSt :: String -> Store -> Maybe Val
accessSt x s = s x

initSt :: Store
initSt = \_ -> Nothing

access :: String -> M Val
access = undefined -- FIXME: implement

assign :: String -> Val -> M ()
assign = undefined -- FIXME: implement

runInterp :: Expr -> Either String Val
runInterp e = fst $ runM (interp e) initSt
```

Now:

1. Adjust your `Functor`, `Applicative`, and `Monad` instances.
2. Add the missing syntax cases for `Assign`, `Access`, and `Seq` to your interpreter.

Your interpreter should use left-to-right order of evaluation.

Hint: It should not be necessary to adjust existing cases of the interpreter. However, the existing monadic operations, such as `err :: String → M a`, may need updating to reflect that `M` has changed.

4.3 Extend with Printing

Extend your interpreter with support for printing strings.

Syntax extension:

```haskell
data Expr
  = Lit Int | Plus Expr Expr | Times Expr Expr
  | Tru | Fals
  | Assign String Expr | Access String | Seq Expr Expr
  -- New constructor:
  | Print String
  deriving Show
```

Recommended monad adjustment:

```haskell
newtype M a = M { runM :: Store -> (Either String a, Store, [String]) }

runInterp :: Expr -> (Either String Val, [String])
runInterp e
  = let (x, _, o) = runM (interp e) initSt
    in (x, o)
```

Now:

1. Adjust the `Functor`, `Applicative`, and `Monad` instances accordingly.
2. Introduce a new operation that represents printing an output:

   ```
   out :: String -> M ()
   ```

3. Write some test expressions in the extended syntax, and what the output of `runInterp` should be.
4. Extend your interpreter with the missing case for the new `Print` constructor.

Hint: It should not be necessary to adjust existing cases of the interpreter. However, existing monadic operations may need updating.

5 Monads are Applicatives

Any monad is also an applicative functor.

As evidence of this fact, the module `Control.Monad` provides the following function:

```
ap :: Monad m ⇒ m (a -> b) -> m a -> m b
```

Implement your own version of this function; e.g.:

```
myAp :: Monad m ⇒ m (a -> b) -> m a -> m b
```

Hint: use monadic bind (or `do` notation) to sequence the arguments of the function.

6 Going Deeper

The following assignments help deepen your knowledge of the concepts considered in the other assignments. You are recommended to only do these assignments once you are done with the remaining assignments.

6.1 Prove that `myAp` is Lawful

Prove that your `myAp` function implemented earlier respects the applicative laws; i.e., prove that each of the following four laws are true.

1. Applicative identity law:

   ```
   pure id `myAp` x        =  x
   ```

2. Applicative homomorphism law:

   ```
   pure (g x)              =  pure g `myAp` pure y
   ```

3. Applicative interchange law:

   ```
   x `myAp` pure y         =  pure (\g -> g y) `myAp` x
   ```

4. Applicative composition law:

   ```
   x `myAp` (y `myAp` z)   =  (pure (.) `myAp` x `myAp` y) `myAp` z
   ```

Hint: the first three applicative laws (identity, homomorphism, and interchange) can be proven using the first two monad laws from the book (left unit and right unit; see also below). To prove the last applicative law (composition), you need also the third monad law (associativity).

Monad Laws

1. Monad left unit law:

   ```
   pure x >>= f      =  f x
   ```

2. Monad right unit law:

   ```
   mx >>= pure       =  mx
   ```

3. Monad associativity law:

   ```
   (mx >>= f) >>= g  =  mx >>= (\x -> (f x >>= g))
   ```

6.2 More Interpretation

6.2.1 Exception Handling

Extend your interpreter with support for raising and handling exceptions.

Syntax extension:

```
data Expr
  = Lit Int | Plus Expr Expr | Times Expr Expr
  | Tru | Fals
  | Assign String Expr | Access String | Seq Expr Expr
  | Print String
  -- New constructors:
  | Throw | Catch Expr Expr
  deriving Show
```

Recommended monad adjustment:

```
newtype M a = M { runM :: Store -> (Either String (Maybe a), Store, [String]) }
```

1. Adjust the `Functor`, `Applicative` and, `Monad` instances accordingly.
2. Introduce new operations that represent throwing and catching exceptions:

   ```
   throw :: M a
   throw = undefined -- FIXME: implement

   catch :: M a -> M a -> M a
   catch m1 m2 = undefined -- FIXME: implement.  If `m1` raises an exception,
                           --   then run `m2`; otherwise return the value
                           --   that `m1` returns.
   ```

6.2.2 Extend with Logical and Conditional Expressions

Our language should also be able to handle logical and conditional expressions; i.e.:

```haskell
data Expr
  = Lit Int | Plus Expr Expr | Times Expr Expr
  | Tru | Fals
  | Assign String Expr | Access String | Seq Expr Expr
  | Print String
  | Throw | Catch Expr Expr
  -- New constructors:
  | And Expr Expr | Or Expr Expr | If Expr Expr Expr
  | LtI Expr Expr | EqI Expr Expr
  deriving Show
```

Extend the interpreter to handle the missing cases.

The intended behavior of ‘LtI‘ (integer less than) and ‘EqI‘ (integer equality) is to compare integer values.

The ‘And‘, ‘Or‘, and ‘If‘ expressions should have their usual behavior.

No monad adjustments should be necessary to support this extension.

6.2.3 Extend with Support for While Looping

Syntax extension:

```haskell
data Expr
  = Lit Int | Plus Expr Expr | Times Expr Expr
  | Tru | Fals
  | Assign String Expr | Access String | Seq Expr Expr
  | Print String
  | Throw | Catch Expr Expr
  | And Expr Expr | Or Expr Expr | If Expr Expr Expr
  | LtI Expr Expr | EqI Expr Expr
  -- New constructor:
  | While Expr Expr
  deriving Show
```

Hint: use recursion in Haskell to implement looping. No monad adjustment necessary.

References

P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52, London, 1995. Springer. URL `https://doi.org/10.1007/3-540-59451-5_2`. Alternative version: `https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf`.