Exercises, Week 45 (3–7 Nov, 2025)

DM580: Functional Programming, SDU

Learning Objectives

After doing these exercises, you will be able to:

- Program with record types in Haskell (Section 1)
- Write programs with interactive input and output in Haskell, using the `IO` monad (Section 2)
- Implement your own definition of sequencing for a model of the `IO` monad (Section 3)

If you do not have time to finish all assignments, make sure to finish some from each section.

Since the book does not say much about record types, Section 1.1 provides a brief tutorial on them.

We recommend that you do the section-numbered exercises in this order:

$$1.2 \to 2.1 \to 2.2 \to 1.3 \to 3 \to 1.4 \to 4.1 \to (4.2).$$

Note that 4.2 covers material that is not a part of the curriculum for the course, but that provides motivation for the kind of programming Haskell's approach to interactive input/output supports.

Note also that the source file (`ex5.org` which is in plain text format—use any text editor to open it) for this exercise set is available on ItsLearning for ease of copy-pasting code snippets.

1 Record Types in Haskell

See 1.2 onwards for exercises on record types. Section 1.1 provides a brief tutorial on record types.

1.1 A Brief Tutorial on Record Types

A record type in Haskell is a regular data type constructor that comes with predefined functions for projecting argument values from constructed data. Record types are typically helpful to use for data types whose constructor(s) have many parameters.

1.1.1 Declaring Record Types

For example, the following declares a record type called `Triple`:

```
data Triple a b c = Triple
  { proj1 :: a
  , proj2 :: b
  , proj3 :: c }
  deriving Show
```

We call `proj1 :: a`, `proj2 :: b`, and `proj3 :: c` the *fields* of the record.

1.1.2 Constructing Record Types

We can construct `Triple` values similarly to how we construct regular data types; e.g.,

```
exampleTriple1 :: Triple Int Bool ()
exampleTriple1 = Triple 42 True ()
```

Or we can name the fields in any order when we construct the value:

```
exampleTriple2 :: Triple Int Bool ()
exampleTriple2 = Triple { proj3 = (), proj1 = 42, proj2 = True }
```

1.1.3 Field Accessors

Unlike regular data types, we get *accessor functions* (sometimes called *projection functions*) for each field of the record; e.g.,

```
tripToPair12 :: Triple a b c -> (a, b)
tripToPair12 t = (proj1 t, proj2 t)
```

1.1.4 Record Updates

When working with records, it is often useful to inhabit the fields of a new record value with the field values of an existing record value. *Record updates* do just this.

For example, the following function takes as input a `Triple a b Int` value, and returns as output a new `Triple` whose fields are the same as the input, except for the third field.

```
tripIntReset3 :: Triple a b Int -> Triple a b Int
tripIntReset3 t = t { proj3 = 0 }
```

Note that, just like plain data types in Haskell, record types are *immutable*. The record update syntax conceptually constructs a new record value. (In practice, the underlying bit-level representation uses sharing to prevent expensive duplication of data. Since data is immutable, this sharing is unobservable, except for lower runtimes and memory footprints.)

For example,

```
λ> let r = Triple "foo" True 42; r' = tripIntReset3 r in (proj3 r, proj3 r')
(42,0)
```

1.1.5 A Note on Multi-Constructor Record Types

Rule of thumb: use record types mainly for single-constructor data types.

It is, in principle, possible to declare record types with multiple constructors. However, multiple constructor record types often give rise to *partial* projection functions.

For example, consider:

```
data TripOrQuad a b c d
  = Trip
    { proj1 :: a
    , proj2 :: b
```

```
    , proj3 :: c }
  | Quad
    { proj1 :: a
    , proj2 :: b
    , proj3 :: c
    , proj4 :: d }
```

While Haskell accepts this record type, the projection function `proj4` has type `proj4 :: TripOrQuad a b c d → d` and will fail if we apply it a `Trip` value!

For example:

```
λ> proj4 (Trip 1 2 3)
*** Exception: No match in record selector proj4
```

A rule of thumb is therefore: use record types mainly for single-constructor data types.

1.1.6 Final Remarks

For more on record types, see

`https://en.wikibooks.org/wiki/Haskell/More_on_datatypes`

The exercises below cover how to program with records in Haskell.

1.2 Book Lending

Consider the following record type.

```haskell
data Book = Book
  { title  :: String
  , author :: String
  , year   :: Int
  , loaned :: Bool
  } deriving Show
```

Use record update syntax (see 1.1.4) to define a function `checkout :: Book → Book` that toggles the loan status of a book.

1.3 Library Cataloging

Using the same `Book` record type, implement a cataloging function that, given a library (i.e., a list of books) returns an *association list* that associates each author with the set of books in the library that the author wrote.

```haskell
groupByAuthor :: [Book] -> [(String, [Book])]
```

The association list should contain no duplicate author names. For example, consider the following library:

```haskell
myLibrary :: [Book]
myLibrary
  = [ Book "Dune"                "Frank Herbert"    1965 False
```

```
    , Book "Dune Messiah"          "Frank Herbert"    1969 False
    , Book "Foundation"            "Isaac Asimov"     1951 False
    , Book "Foundation and Empire" "Isaac Asimov"     1952 False
    , Book "I, Robot"              "Isaac Asimov"     1950 True
    , Book "The Martian"           "Andy Weir"        2011 True
    , Book "Project Hail Mary"     "Andy Weir"        2021 False
    , Book "2001: A Space Odyssey" "Arthur C. Clarke" 1968 False ]
```

An example output of calling `groupByAuthor myLibrary` is given below. Note that the order of the authors and books in the association list is not essential. It is essential that each author is associated with each book that they wrote.

```
λ> groupByAuthor myLibrary
[ ("Arthur C. Clarke",
    [ Book { title  = "2001: A Space Odyssey"
           , author = "Arthur C. Clarke"
           , year   = 1968
           , loaned = False } ])
, ("Andy Weir",
    [ Book { title  = "The Martian"
           , author = "Andy Weir"
           , year   = 2011
           , loaned = True }
    , Book { title  = "Project Hail Mary"
           , author = "Andy Weir"
           , year   = 2021
           , loaned = False } ])
, ("Isaac Asimov",
    [ Book { title  = "Foundation"
           , author = "Isaac Asimov"
           , year   = 1951
           , loaned = False }
    , Book { title  = "Foundation and Empire"
           , author = "Isaac Asimov"
           , year   = 1952
           , loaned = False }
    , Book { title  = "I, Robot"
           , author = "Isaac Asimov"
           , year   = 1950
           , loaned = True } ])
, ("Frank Herbert",
    [ Book { title  = "Dune"
           , author = "Frank Herbert"
           , year   = 1965
           , loaned = False }
    , Book { title  = "Dune Messiah"
           , author = "Frank Herbert"
           , year   = 1969
           , loaned = False } ]) ]
```

If you want to pretty-print your output akin to what's shown above, you can either write your own pretty-printer, or use the one provided in Appendix A.

1.4 Bank Accounts

Consider the following record type.

```
data Account = Account
  { holder :: String
  , balance :: Double
  } deriving Show
```

1.4.1 Depositing, Withdrawing, Transferring, and Balancing

Implement the following functions.

`deposit :: Double → Account → Account` which should increment the balance.

`withdraw :: Double → Account → Maybe Account` which decrements the balance, or returns `Nothing` in case of insufficient funds.

`transfer :: Double → Account → Account → Maybe (Account, Account)` which transfers money from the first account to the second, returning the updated set of accounts; or `Nothing` if there is insufficient funds.

`totalBalance :: [Account] → Double` which returns the total balance of a set of accounts.

1.4.2 Transactions

Let us introduce a record for transactions:

```
data Transaction = Transaction { from :: String, to :: String, amount :: Double }
```

Using this, implement a function `applyTransaction :: Transaction → [Account] → Maybe [Account]` which finds the accounts with `holder` names matching the `from` and `to` fields of the transaction (you can assume that no two accounts have the same holder field), and applies the transaction to yield an updated set of accounts, if the transaction succeeds.

2 IO in Haskell

These exercises are based on Chapter 10 of the book. If you haven't read the chapter, it is a good idea to do so before doing the exercises.

2.1 Adder (10.10.4 from the Book)

Define an action `adder :: IO ()` that reads a given number of integers from the keyboard, one per line, and displays their sum. For example:

```
λ> adder
How many numbers? 5
1
```

```
3
5
7
9
The total is 25
```

Hint: start by defining an auxiliary function that takes the current total and how many numbers remain to be read as arguments. You will likely need to use the library functions `read` and `show`.

2.2 Sliding Window

Implement an action `sliding :: IO ()` that continuously reads inputs, and then prints the three most recently seen inputs.

```
λ> sliding
Window: []
Next number? 7
Window: [7]
Next number? 3
Window: [3,7]
Next number? 3
Window: [3,3,7]
Next number? 1
Window: [1,3,3]
[...]
```

Hint: look at how we added game state (turn tracking) to our hangman game in the lecture of week 5 (in `Week5_postlecture.hs` on ItsLearning).

3 A Mathematical Model of `IO`

In the lecture, we implemented our own model of the `IO` monad, by modeling it as a function that mutates the state of the world.

Your task is to implement the following `bind` function which lets us sequence together `IOish` operations:

```haskell
bind :: IOish a -> (a -> IOish b) -> IOish b
bind m k = undefined
```

The rest of this exercise text recalls what we went through in the lecture, and provides hints on how to implement the function.

Recall that `IOish` is defined as a function that conceptually mutates the state of the world. For our purposes, we model the world as a list of inputs, and a list of outputs:

```haskell
data World = World { inputStrings  :: [String]
                   , outputStrings :: [String] }
  deriving (Show, Eq)
```

The initial state of the world is constructed by giving a list of inputs:

```haskell
initWorld :: [String] -> World
initWorld inputs = World inputs []
```

Our model is then given by the following data type:

```haskell
newtype IOish a
  = IOish { runIOish :: World -> (Either String a, World) }
```

Because Haskell does not support mutation, and because we do not want our model to use the circular explanation that "mutation is mutation", we model world mutation as a function that, given an input world, produces an output world. The idea is that the output world yielded by `IOish` functions are fed forward; i.e., future `IOish` computations take as input the worlds computed by previous `IOish` computations.

To interact with the world, we build functions that represent an abstract interface that lets us mutate the underlying world. We implemented these functions together in the lecture, matching the interface functions provided by Haskell for the `IO` monad:

```haskell
pureish :: a -> IOish a
pureish x = IOish $ \w -> (Right x, w)

putStrLnish :: String -> IOish ()
putStrLnish str = IOish $ \w ->
  ( Right ()
  , w { outputStrings = str : outputStrings w } )

getLineish :: IOish String
getLineish = IOish $ \w ->
  if null (inputStrings w)
  then (Left "Tried to access empty list of inputs", w)
  else ( Right (head (inputStrings w))
       , w { inputStrings = tail (inputStrings w) } )
```

For example:

```
λ> runIOish getLineish (initWorld ["Hello"])
(Right "Hello",World {inputStrings = [], outputStrings = []})
```

We can sequence together `IOish` computations by explicitly unpacking and invoking the underlying functions:

```haskell
explicitExample :: IOish ()
explicitExample = IOish $ \w ->
  case runIOish getLineish w of
    (Left err, w') -> (Left err, w')
    (Right str, w') -> runIOish (putStrLnish str) w'
```

However, writing such functions punches a hole in our `IOish` abstraction. What we want instead is to write `IO` computations like the ones found in Chapter 10 of the book, where we used `do` notation to sequence together operations.

The `bind` function lets us do so:

```
bind :: IOish a -> (a -> IOish b) -> IOish b
bind m k = undefined
```

The type of `bind` says, if we have an initial computation (i.e., a world-mutating function) that returns `a`, and we have a *continuation* which, given an `a` produces a computation `b`, then we can construct a computation that returns the `b` directly. The way to implement `bind` is to first run the initial computation, and then invoke the continuation on the result, feeding the updated world from the initial computation forward to the continuation.

Your `bind` function should allow us to ”play” hangman, by providing inputs representing an interactive sequence of game interactions.

That is, using these functions from the lecture:

```
hangmanish :: IOish ()
hangmanish =
  putStrLnish "Think of a word:" `bind` \_ ->
  getLineish `bind` \word ->
  putStrLnish "Try to guess it:" `bind` \_ ->
  playish word

playish :: String -> IOish ()
playish word =
  putStrLnish "? " `bind` \_ ->
  getLineish `bind` \guess ->
  if guess == word then putStrLnish "You got it!!"
  else putStrLnish (match word guess) `bind` \_ -> playish word

match :: String -> String -> String
match xs ys =
  [if elem x ys then x else '-' | x <- xs]
```

We can run `hangmanish` to play the game, like we did in the lecture, but now using our model of the world:

```
λ> runIOish hangmanish (initWorld ["odense", "foo", "bar", "dense", "odense"])
( Right ()
, World { inputStrings = []
        , outputStrings = [ "You got it!!"
                          , "? "
                          , "-dense"
                          , "? "
                          , "------"
                          , "? "
                          , "o-----"
                          , "? "
                          , "Try to guess it:"
                          , "Think of a word:" ] } )
```

4 Going Deeper

4.1 TicTacToe (Chapter 11)

Work through Chapter 11 of Programming in Haskell, to implement an unbeatable TicTacToe game.

Then work through the exercises at the end of the chapter.

4.2 Software Transactional Memory (OPTIONAL Exercise)

This exercise concerns material that is not part of the curriculum for the course.

Software transactional memory (STM) provides a clean solution to the problem of concurrent programming with shared memory—a problem that is highly non-trivial, yet of utmost importance in modern programming where multi-core execution is the standard.

To illustrate the problem, say we have two processes p_1 and p_2 which both read from a shared variable $x = 0$, and which both increment it. We should expect that running p_1 and p_2 would cause a final memory state where $x = 2$, since the variable is incremented by both processes. However, p_1 might read the value of x first. Then process p_2 might read the same datum and overwrite the memory with the result of incrementing it; i.e., $x = 1$. Meanwhile, process p_1 still thinks the value of the memory location is 0! So it writes what it thinks is an incrementation of the memory, namely 1 again.

Most concurrent programs with shared memory involve critical memory regions which must be accessed and mutated in an *atomic* manner. That is, accessing and mutating those memory regions should behave as though processes access it in *sequence*.

Different programming languages provide different solutions to this problem. For example, Java relies on `synchronized` fields and methods, which, behind the scenes, rely on *locks* which ensure that critical memory can only be accessed by a single process at a time. However, locks can be difficult to work with, can be a source of inefficiency, and risks *deadlocks* where two processes are stuck mutually waiting for the other process to release its lock.

STM avoids many of the pitfalls with locks, making it easier for programmers to write concurrent programs. While STM has been implemented in many languages, its implementation in Haskell is particularly clean, as it leverages Haskell's type system to rule out programming mistakes that could lead to bugs in STM implementations found in other languages.

In this exercise, you are encouraged to read and work through Sections 3 and 4 of Simon Peyton Jones' paper describing STM (Peyton Jones, 2007).

To work through it, you need the `stm` library. You can install this, using Haskell's package manager, `cabal`.

Assuming you have `cabal` installed, run the following in your terminal (not GHCi):

```
$ mkdir stmplayground
$ cd stmplayground
$ cabal init
```

The `cabal init` shell command will guide you through setting up a default `cabal` file.

Once the `cabal` file has been set up, you should have a folder containing the following:

```
$ ls -l
total 24
drwxr-xr-x@ 3 cbp  staff    96 Oct 31 11:46 app
-rw-r--r--@ 1 cbp  staff   116 Oct 31 11:46 CHANGELOG.md
-rw-r--r--@ 1 cbp  staff  1515 Oct 31 11:46 LICENSE
-rw-r--r--@ 1 cbp  staff  2470 Oct 31 11:46 stmplayground.cabal
$ ls -l app
total 8
-rw-r--r--@ 1 cbp  staff  48 Nov  1 10:45 Main.hs
```

Open `stmplayground.cabal` and find the line that says:

```
    build-depends:    base ^≥4.21.0.0
```

Modify this to include the stm package:

```
    build-depends:    base ^≥4.21.0.0, stm
```

Save and close the file.

Now, when you navigate to `stmplayground/app/Main.hs`, you should be able to program with STM in Haskell. You can test it by running the following experiment which should allow you to observe the difference between using concurrency with plain mutation and concurrency with STM's `atomically` block.

```haskell
module Main where

import Control.Concurrent
import Control.Concurrent.STM
import Control.Monad
import Data.IORef

main :: IO ()
main = do
  counter <- newIORef 0 -- ordinary mutable variable in IO

  -- Spawn 10 threads, doing 10000 iterations of reading the
  -- counter and then incrementing it.  Because the threads
  -- execute non-atomically, they sometimes overwrite each
  -- others' modifications to the state, leading to results
  -- other than the expected 10*10000 = 100000.
  replicateM_ 10 $
    forkIO $ replicateM_ 10000 $ do
    n <- readIORef counter
    writeIORef counter (n + 1)

  threadDelay 1000000 -- wait for threads to finish

  final <- readIORef counter
  putStrLn $ "[NON-ATOMICALLY] Final counter value: " ++ show final
```

```
  -- Now let's do the same, but using STM
  counter' <- newTVarIO 0

  replicateM_ 10 $
    forkIO $ replicateM_ 10000 $ atomically $ do -- note: atomically!
      n <- readTVar counter'
      writeTVar counter' (n + 1)

  threadDelay 1000000

  final' <- readTVarIO counter'
  putStrLn $ "[  ATOMICALLY  ] Final counter value: " ++ show final'
```

To load the experiment into GHCi, run the following shell command:

```
$ cabal repl
```

You can now call functions in `Main.hs` to get results such as these:

```
λ> main
[NON-ATOMICALLY] Final counter value: 20000
[  ATOMICALLY  ] Final counter value: 100000
λ> main
[NON-ATOMICALLY] Final counter value: 20004
[  ATOMICALLY  ] Final counter value: 100000
λ> main
[NON-ATOMICALLY] Final counter value: 100000
[  ATOMICALLY  ] Final counter value: 100000
λ> main
[NON-ATOMICALLY] Final counter value: 60009
[  ATOMICALLY  ] Final counter value: 100000
```

References

S. Peyton Jones. Beautiful concurrency. In G. Wilson, editor, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 24, pages 549–556. O'Reilly Media, Sebastopol, CA, 2007.

A Appendix: Pretty-Printer for Books

```haskell
import Data.List (intercalate)

prettyLib :: [(String, [Book])] -> String
prettyLib xs = "[ "
               ++ intercalate "\n, " (map (prettyPair 1) xs)
               ++ " ]"
  where
    prettyPair :: Int -> (String, [Book]) -> String
    prettyPair i (auth, bs) =
        "(" ++ show auth ++ ",\n"
        ++ indent (i+1) ++ "[ "
        ++ intercalate (indent (i+1) ++ ", ") (map (prettyBook (i+2)) bs)
        ++ indent (i+1) ++ "])"

    prettyBook :: Int -> Book -> String
    prettyBook i b = "Book"
        ++                  " { title  = " ++ show (title b)  ++ "\n"
        ++ indent i ++ "    , author = " ++ show (author b) ++ "\n"
        ++ indent i ++ "    , year   = " ++ show (year b)   ++ "\n"
        ++ indent i ++ "    , loaned = " ++ show (loaned b) ++ " }\n"

indent :: Int -> String
indent n = replicate (n * 2) ' '

demo :: IO ()
demo = putStrLn $ prettyLib $ groupByAuthor myLibrary
```