Making Sense of Monads

Exercises

Lecture 1 - Introduction

Welcome to Making Sense of Monads! The heart of improving your skills is practicing them, and these exercises aim to help you do just that!

Getting Set Up

The code for this course's exercises lives in a Github repository. You can get a current version of this code in the zip file attached to this lecture. Updates to the exercise code are occasionally pushed out through Github. To get these, you should first email me at james@mondaymorninghaskell.me with your Github name so I can add you as a collaborator to the repository. Then you can run the following commands to get updates:

```
>> git remote add origin \
  https://github.com/MondayMorningHaskell/MakingSenseOfMonads
>> git pull origin main
```

If you are added to the repository, you can also clone it directly instead of unzipping the file:

```
>> git clone \
  https://github.com/MondayMorningHaskell/MakingSenseOfMonads
```

If you use an SSH key for your Github identification, you'll need to use this URL instead to clone and pull updates:

```
git@github.com:MondayMorningHaskell/MakingSenseOfMonads.git
```

Note: If you are added to the GitHub repository, you'll also be able to look at the `answers` branch in case you get stuck.

The code requires you to have the Haskell Platform installed, particularly Stack, so make sure you've done that already.

If you don't, you can install them quite easily. On Mac and Linux, you should be able to run:

```
>> curl -sSL https://get.haskellstack.org/ | sh
```

For Windows, you should download the appropriate installer from the documentation linked above.

To verify everything is installed, you should be able to build the code, and run the basic `run-msm` command like so:

```
>> stack build
```

```
>> stack exec run-msm
```

This should display the "help" console output.

```
>> run-msm
```

Running Exercises

You can run exercises using this `run-msm` command. It is highly recommended you use `stack install` so that you can use the executable without `stack exec`:

```
>> stack install
>> run-msm 1
```

You must pass an argument to `run-msm` to indicate which exercises to run. This should either be the lecture number (1-14) or the name of the lecture, which will also be listed in the exercise document.

For this first lecture, you can run the exercises with either of the following commands:

```
>> run-msm 1
>> run-msm Introduction
```

Running exercises will typically involve compiling your code and informing you of any errors. If your code compiles, it will then run some unit tests. If your unit tests pass, you can move on to the next lecture!

Introductory Exercises

Once you have everything configured, you can start off by doing our introductory exercises. These are intended to motivate some of the use cases we'll have for using monadic functionality in our code. These can be found in the file `src/exercises/Introduction.hs`. You'll want to modify this file to solve the following exercises until the `run-msm 1` command given above is successful.

Not So Convenient

Notice the function `transformString`. Write a new function `transformStringMaybe` that takes a `Maybe` parameter, and outputs a `Maybe String`. If the input is `Nothing`, the output should be too. Otherwise it should follow the same rules as `transformString`.

Triangle of Doom

Next, there are three functions that mess with their input string in different ways. Each takes its input and returns a result wrapped in `Maybe`. Fill in the definition of `transformInput`, which will take a single input `String`. First, it will pass it to `evaluate1`. If that succeeds (is `Just` and not `Nothing`), then it will pass that result to `evaluate2`, and so on. If `evaluate3` succeeds, then you'll return the result. How many `case` statements do you need?

How Did it Fail?

Notice the definition of `transformInputEither`, the corresponding `Either` functions, and the `InputError` type. These act like their counterparts in `Maybe` above, except instead of returning `Nothing` on failure, they return a particular kind of error. Write `transformInputEither` to be like `transformInput`, except using `Either`.

Adding, Maybe?

Fill in `addMaybe`, a function that will return the result of two `Maybe Int` values, as long as they are both `Just`. If either of them is `Nothing`, the result should be `Nothing`. Notice that if you wanted to have this same structure except with multiplication, you would need to write a second function `multiplyMaybe`. We'll see how we can avoid this later in the course.

Forgotten Parameter

Notice the function `deepCall`, which triggers a pretty deep call chain. Unfortunately, you just realized that the last call, `deepestFunc`, requires the additional `IntWrapper` argument ignored in `deepCall` so that it can include it in the addition. Add this argument to each successive function in the call chain to pass it to `deepestFunc`. Then should add the wrapped `Int` value to the final result in that function.

Tangled Costs

You don't have to do anything for this part. Just take a look at the bottom at the functions `func1`, `func2`, `func3`, and `func4`. These are a series of functions that follow a bunch of bizarre rules for computing a string and tracking an arbitrary total “cost” of the operation. The purpose of the example is to demonstrate how complicated it is to keep track of a secondary variable using pure code. We have to constantly return tuples and pass them as parameters even when they really aren't part of the problem we're dealing with. We'll learn a better way to approach this problem in this course!

Lecture 2 - Monoids

These exercises are in `src/exercises/Monoids.hs`. You can run them with either of these commands. Note that the tests will not compile until you fill in some missing type definitions in the exercises!

```
>> run-msm Monoids
>> run-msm 2
```

Operator Exchange

For a warm up, take a look at the functions for `combineStrings` and `combineInts`. They look exactly the same except that the first combines the elements in the tuples using list append `(++)` and the second combines using addition `(+)`. Fill in `combineAll`, a general function that will work for both the `String` type and the `Int` type, given the `Semigroup` instance for `Int` we have in the file.

Newtype Instances

In the lecture, we saw that there are a couple different ways we could potentially write a `Monoid` instance for `Int`. But since we have the addition instance already, we can't write a multiplication instance! (You can only have one instance of a typeclass for each type in scope).

There's a way around this though! Define `newtypes AdditionInt` and `MultiplicationInt`. Then you can write separate `Monoid` and `Semigroup` instances for each of them! (You should also derive `Show` and `Eq` to make the tests work).

Then fill in the general combination function `foldInts` that takes a list of either of these wrapper types and combines them using the right operation.

Maps as Monoids

The `Map` type that stores a mapping from one ordered "key" type to a "value" type. A `Map` is also a `Monoid`! Take a look at the example maps, and then fill in what you expect their combinations to be below. The tests will compare your guess against the built-in `Monoid` instance. When combining maps, you may have to break “ties” in some way. How would you do that? Remember, the solution you pick has to be associative, but not necessarily commutative.

Lecture 3 - Functors

All these questions are in `Lecture3.hs`. Test with `run-msm Functors` or `run-msm 3`!

Functor Simplicity

Remember from lecture 1 how we had the `transformString` function and we needed to write `transformStringMaybe`? Take a look at the code now and see if you can remove the call to `transformStringMaybe`.

Safety First

The definition of `multiplySqrtDouble` uses the `safeSquareRoot` function, which returns a `Maybe`. It uses a `case` statement to evaluate that result. Rewrite the function so that it is a simple, one line function with `fmap`!

.

Put a Functor in Your Functor

Look at `undecidedList`. It is a list containing many `Maybe Int` elements. With that as an example, fill in `plus2`, which should take a list of `Maybe Ints` and add `2` to all of the `Just` elements in that list, leaving the `Nothing` elements untouched.

Storing People

Now take a look at the `CityGovernment` data structure. It is parameterized by the type we use to represent the people in it. Under it, you'll see three different types we use for defining people, as well as a couple conversions between them. Fill in the `convertGovernment` functions that will change the inner type of every person. Do this by defining an instance of `Functor` for the `CityGovernment` type! That way you can just fill in `convertGovernment` with `fmap`.

Measurement Functor

Write a functor instance for the `Metrics` data type. Then fill in the `doubleMetrics` function so that it will double every value in a `Metrics` object, using `fmap`.

Lecture 4 - Applicatives

See `src/exercises/Applicatives.hs`, test with:

```
>> run-msm Applicatives
```

or

```
>> run-msm 4
```

Applicative Ease

Look back to lecture 1 and the `addMaybe` function. Rewrite this using applicative syntax!

Safety First (Part 2)

Fill in `sumOfSquareRoots`, so that it returns the sum of the square roots of its inputs. Use `safeSquareRoot` to check for negative inputs, and return `Nothing` if either is negative.

A Comprehensive Applicative

Remember list comprehensions from module 3? Take a look at `myApplicativeList`, which calculates the “distance number” (for our arbitrary definition of distance) of every pair of numbers from the two lists. Rewrite this function using applicative syntax!

Generate All Results

Given a list of operations and two lists, generate all combinations of those operations with each pair of numbers from the two lists.

Why Do You Build Me Up Just to Let Me Down?

Examine `Profile`, a type with many different fields. We have a function taking all the necessary parameters, but all of them are `Maybe` values! If all of them end up being `Just` values, you should return the full data structure. Otherwise you should return `Nothing`. See if you can use applicative syntax to write this function without resorting to any case statements or pattern matching!

Functors and Applicatives

Consider the `Functor` instance you made for `CityGovernment` in the last set of exercises. Consider: would it ever make sense to make an `Applicative` instance for this type?

Lecture 5 - Monads

See `src/exercises/Monads.hs`, test with:

```
>> run-msm Monads
```

or

```
>> run-msm 5
```

No More Triangular Code!

Think back to the `tranformInput` function from lecture 1. If you did things the simple way, you probably needed three case statements, and they were arrayed in an awkward triangle. Rewrite that function, only now use the bind operator and the `Maybe` monad! You should be able to avoid using any case statements!

How did it Fail (Part 2)

Now fill in `transformInputEither` again, using the `Either` monad!

Safety First (Part 3)

Fill in the definition of `sqrtAndMultiply`. Use the monad bind operator to apply `safeSquareRoot` to the input and then multiply it by 10 using `multiplyIfSmall`.

More Fun with Lists

Fill in the function `tripleProducts`. The function itself is straightforward. You want every pairwise product of the first two lists, and then you want to take each of those products, and get every pairwise product with the third list. We've done similar things with list comprehensions before. But now, you should do it using the list monad and the bind operator! Remember, under the list monad, you can pass a function that takes only a single input and produces a list!

Add and Negate

Fill in `addAndNegate`. This takes a list of inputs and produces a new list that adds 1, 2, and 3 to each input. Then for the final result, it also includes the negation of every resulting value. Look at the example for the expected order of results.

Lecture 6 - Do Syntax

See `src/exercises/DoSyntax.hs`, test with:

```
>> run-msm DoSyntax
```

or

```
>> run-msm 6
```

Rewrite with Do!

All five of the functions from the last set of exercises are here again. Re-implement them all using do-syntax instead of the bind operator!

Unbinding the Chains

Take a look at `wayTooManyBinds`. It's a monadic action that only uses the bind operator to pass its internal state. But it gets very complicated! We have an abundance of pass-through parameters. Rewrite the expression using do-syntax and observe how much simpler it is!

Lecture 7 - Reader and Writer Monads

See `src/exercises/ReaderWriter.hs`, test with:

```
>> run-msm ReaderWriter
```

or

```
>> run-msm 7
```

Deep Call, Simplified

Remember our friends `deepCall` and `deepestFunc` from lecture 1? Rewrite them once again to add the `IntWrapper` argument, only now do it with a `Reader` monad instead of an extra parameter!

Localization

Take a look at the `processList` function. It's a `Reader` function with a list of numbers as its state that returns an `Int`. Now fill in the definition of `differentOutcomes`. This is similarly a `Reader` action that takes a list of integers. We want a tuple of five integers, where the first result is the result of `processList` on the full list, the second is `processList` when dropping the first element, the third comes from dropping two elements, and so on. Use the `drop` function which simply gives the empty list if there aren't enough elements. Use `local` to call into the `processList` function with these different combinations!

Getting the Groceries

Observe the `GroceryItem` type. Each item has a string, a weight value (`Int`) and a `GroceryCost`.

We want to organize these into different bags and have that as the outcome of the `processGroceries` function. But while we're at it, we want to track the total cost of the groceries. For each bag used, you should assess an extra `GroceryCost` of one dollar. Change the type signature of `processGroceries` to return the cost using the `Writer` monad and update the helper functions accordingly. You'll need to make a `Monoid` instance for `GroceryCost`.

Undoing IO

Take a look at the `computeIO` function. This uses the IO monad (which we'll learn about in a couple lectures). All it does is a few computations on integers, but it has print statements logging what it is doing each time.

Fill in the type signature and definition of `computeWriter` below. Instead of using IO (which we want to avoid because the side effects are limitless), this should use the `Writer` monad to track the output as a list of strings.

Lecture 8 - State Monad

See `src/exercises/State.hs`, test with:

```
>> run-msm State
```

or

```
>> run-msm 8
```

Suddenly Writable

Take a look at `getResults`. It is a `Reader` function that runs a couple simple computations on integers. One of these involves a call to `local`, which as you recall has no effect on the stored value. Change these functions to instead use the `State` monad. Then instead of merely calling `local`, you should permanently update the stored `Config` value by applying the `doubleConfig` function.

Computing Cost

First fill in the definition for `applyOpCount`. This should return the result of running the `applyOp` function but also add the `opCost` of the operation to the `State Int` value. Then use this function to fill in `applyAndCountOperations`, which will compute the total cost AND the final result of applying all the operations. (Recursion is your friend here!)

Grocery List with State

For exercise 3, we've copied over the previous starter code from Lecture 6 on the grocery list problem.

Feel free to replace it with your solution from Lecture 6, except that you should now change your solution to use the `State` monad instead of the `Writer` monad!

The State of Stones

In a variant of the game Nim, we have a pile of stones. On their turn, a player must take at least one stone. But they can take no more than half the stones in the pile. The player who takes the last stone loses the game. We have a couple data types surrounding how to play this game and how moves are encoded. We also provide the function `playNimGame` that calls into the state monad.

Your job is to write the function `processNimMoves`. This should determine the result of a game, based on the moves in the game and the current state. Once again, you'll need to use recursion! Also, don't worry about handling edge cases! For instance, there will always be enough moves in the list to finish the game (take the pile down to 0).

Lecture 9

See `src/exercises/IO.hs`, test with:

```
>> run-msm IO
```

or

```
>> run-msm 9
```

Since these exercises deal with user input, you can also run this particular module by itself by using the executable `run-io`! Note however that you'll need to re-build and/or re-install your code after each modification in order to do this. You'll also need to rebuild before you run the test commands above!

```
>> stack build
>> stack exec run-io
```

or

```
>> stack install
>> run-io
```

Running the command like this can help you figure out if you're getting the correct output format more easily.

In these exercises, you're going to implement a series of small, IO related items. You'll mostly be filling in the `main` function in the exercise module!

1. To start, print `"Hello, World!"`, using `putStrLn`.
2. Then, print, `"Running from directory: "` followed (on the same line) by the "current" directory the program is running in.
3. Next, find the home directory on your system and print `"Home directory is..."`, followed by the filepath of that directory.
4. After that, "list" the home directory and print the number of files and directories it contains. For example: `"Home directory contains 10 sub-paths"`.

Finally, fill in the function `readSingleNumber`. This should prompt the user to enter a number by printing "Please enter a number." Then retrieve an integer entered by the user. If it can be read properly as an integer (use `readMaybe`), print `"Received x"`, where x is the number. If it cannot, print `"Could not read that as an integer."` and return `Nothing`.

From your main function, call `readSingleNumber` twice. If both inputs can be parsed, print their sum with `"The sum of these is x."` Otherwise, print `"Sum is not possible."`

Lecture 10

See `src/exercises/Files.hs` test with:

```
>> run-msm Files
```

or

```
>> run-msm 10
```

There are a few different functions to implement for these exercises. You can implement all of these using the `Handle` abstraction, but some of them can also be written without it. Try each of these both ways! Note that running these exercises will create files in your "home" directory and then subsequently delete them.

Retrieve All the Lines

Fill in `getAllLines`, which takes a file and returns the full contents of that file, except divided into a list of strings, where each element is a single line of the file.

Only the First!

Next fill in `readFirstLine`. This function should only retrieve the first line of a file.

Saving Home

In Lecture 9, we used `listDirectory` to get a list of all the sub-paths in our "home" directory. In the `saveSubpaths` function, you should do this again, except that you should write the results of this function out to the `FilePath` given as an input to the function (this argument is NOT the home directory itself). Before writing the listing results, you should write the name of the directory itself at the top of the file.

How Many Lines?

For the function `appendNumberOfLines`, you should take an input file, and determine how many lines currently exist in the file. Then you should write an additional line in the file, saying `"There are now x lines!"`, where x is the **new** number of lines in the file (so the original number + 1).

For this function, you'll be doing multiple operations on the same file. So you'll definitely want to use the **Strict** library for part of it! It's imported in a qualfied manner at the top, so you can use functions like `S.readFile` or `S.hGetContents` to read files strictly.

Lecture 11

See `src/exercises/Transformers.hs`, test with:

```
>> run-msm Transformers
```

or

```
>> run-msm 11
```

As with lecture 9, you can run this exercise as a separate executable, so you can try custom inputs. Once again, you'll need to re-build and/or re-install your code after each modification in order to do this and each time you want to run the tests. This executable will begin by asking for a number of users to try registering, after which you'll enter the information for that many users.

```
>> stack build
>> stack exec run-transformers
```

or

```
>> stack install
>> run-transformers
```

Indicating Errors with IO

At the top of the exercise file, we've written out the example from the lecture of reading items in from the terminal. Right now, the functions work and return `Nothing` properly. But this won't tell the user what went wrong. For a warm up, modify these functions to print the given error message specifying what went wrong in each case.

1. `readEmail - "Not a valid email address!"`
2. `readPassword - "Password isn't long enough!"`
3. `readAge - "Age is not an integer!"`

Computing Cost Redux

Recall the "Computing Cost" exercise. We've replicated the `Op` type, and added a `Show` instance. Rewrite `applyOpCount` and `applyAndCountOperations` from your old code, but now use a monad transformer over `IO`. In `applyOpCount`, you can then print the operation that is being used.

A Mix of Monads

Take a look at the type synonym we have for `BigMonad`. It should look familiar from the lecture. We use a `Reader`, `Writer`, and `State` monad, all parameterized by integers, all on top of the IO monad.

Fill in the definition of `mixNewNumber`, which takes a single integer. First, it should print the message `"Received {i}"`, where `{i}` is the input parameter. Then, it should read the integer from the `Reader` and add it to the input. Then it should `tell` this sum to the `Writer` (which will add it). Then, if the sum is even, it should modify the `State` by adding twice the sum. If the sum is odd, it should subtract the sum from the `State`. Finally, return three times the sum value.

Simplifying the Pattern

The pattern of `Reader/Writer/State` is common enough that there is a specific construct for it, using `RWST` as the type name! Fill in `mixNewNumberPattern`, having the same functionality except that the function uses the `RWS` shortcut. With `RWS`, you won't have to use `lift`, except once in order to call into `IO`. Note though, that you'll need to use the qualified functions from the `RWS` module above in order to use the normal monadic functions. (e.g. `RWS.get`, `RWS.ask`, etc.) We need to import that module as `qualified` here, as otherwise its functions will conflict with the normal functions.

Lecture 12

See `src/exercises/Laws.hs`. You'll want to use commands like:

```
>> run-msm Laws
```

or

```
>> run-msm 12
```

Breaking the Law

Think back to the `CityGovernment` type from lecture 3. In the file we've replicated the type and a nearly complete `Functor` instance. Now instead of filling in the `interimMayor` with something logical, set it so that it's always `Nothing` after `fmap`. You can run the tests now, and they should fail!

Examine `stringIdentityCheck`. This is a test function using the `QuickCheck` library that seeks to verify that the identity law holds for our functor instance. Look closely to see where we are actually verifying the identity property! The details of this library are beyond the scope of this course, but it's useful to know that there's a tool out there to verify abstract properties of our code.

Fix your `Functor` instance and run the tests to make sure they pass. Then move onto the next part.

Checking our Math

Recall our `AdditionInt` type from Lecture 2. We've copied that over with reasonable instances for `Monoid` and `Semigroup`. Following the example of `personIdentityCheck`, fill in `additionIdentityCheck` and `additionAssociativityCheck`. These predicate functions should verify the identity and associativity laws of monoids, respectively.

Your predicates should state that two different expression values (created from the function inputs) are equivalent.

In the `tests` expression towards the bottom, uncomment the two lines with these functions and verify that the tests still pass.

Disorder of Operations

Take a look at a similar `newtype SubtractionInt`. Can you write proper instances for `Monoid` and `Semigroup` that use subtraction as the operation? If not, what monoid rules does

it break and how? Verify whether these operations work (or don't work) by filling in `subtractionIdentityCheck` and `subtractionAssociativityCheck` to match the test functions you wrote above for addition. Then uncomment them in `main` and run again! (It's OK if these tests "fail"!)

Lecture 13 - Parsing

See `src/exercises/Parsing.hs`, test with:

```
>> run-msm Parsing
```

or

```
>> run-msm 13
```

To test out your basic parsing skills, we're going to write some parsing functions that can handle JSON input data. There are several different test groups for this set of exercises. Unlike previous exercises, we actually list the test cases at the bottom of the file, so you can enable and disable different sets as you go along. Just modify the lines in the `testCases` expression. Some of them are commented out to begin with, but you should uncomment them as you go along.

Basic Parsers

To start out, let's write parsers for the simple JSON cases: strings, numbers, booleans and null. Fill in the parsing functions according to these basic rules:

1. `stringParser`: Any set of characters between two quotation marks. For these exercises you can ignore the edge cases of a string which contains quotation marks itself.
2. `numberParser`: Any integer or decimal number. An integer is just a series of digit characters. A decima lmust have the decimal point "." and then it has a series of digits either before or after (or both). Your parser should handle negative numbers. Do not worry about scientific notation (e.g. `2.3e-9`)
3. `boolParser`: Any string that matches `true` or `false` in a case-insensitive manner.
4. `nullParser`: Any string that matches `null` in a case-insensitive manner.

Now fill in `valueParser` so that it will `try` each of these options in turn. At this point you can run the tests and they should be able to pass the `basicValue` test group, as well as the four test groups for the individual parsers.

Arrays

Now fill in `arrayParser`. This should parse a "left" bracket (`[`) and then a comma-separated list of other JSON values, followed by a "right" bracket (`]`). You'll want to use `valueParser` from above. You should allow any number of spaces between brackets, elements, and commas.

Hint: the `sepBy` combinator (described in the code comments) is your friend here!

At this point, your code should be able to pass the `basicArray` test group. But you won't be able to pass the `nestedArray` group. Since one JSON array can be nested within another, you should add `arrayParser` as another option to `valueParser`. Then you'll pass that test group!

Objects

For the last step, fill in `objectParser`. This will be similar to `arrayParser`. Instead of brackets though, you'll use curly braces (`{}`). And instead of simply having the values themselves, each value will be preceded by a string key and a colon. The key should follow the same rules as the `stringParser` above. It must begin and end with quotation marks but can have any kind of characters in between those, except other quotation marks.

Hint: Make a helper function to parse a string key, the colon, and the JSON value.

Once again, you should also add `objectParser` as an option to `valueParser`. This will help you pass the final test cases!

As noted in the pre-lecture write-up, make sure to use `Data.Aeson.KeyMap` and `Data.Aeson.Key` when constructing the final object. These are both imported `qualified as K`. You'll particularly want to use `K.fromText` to convert a `Text` into a `Key`, and `K.fromList` to build the `KeyMap` that is associated with the `Object`.

Bonus Challenges

As an extra challenge, you can try to tackle some of the edge cases we ignored above. Try to modify `stringParser` so that it can handle escaped quotation marks (`\"`) within the input. Then modify `numberParser` so that it can handle scientific notation. To run these tests along with your code, you can uncomment the `bonusTests` line in the `main` function.

Lecture 14 - GEDCOM Parsing

See `src/exercises/GEDCOM.hs`, test with:

```
>> run-msm GEDCOM
```

or

```
>> run-msm 14
```

This is the final challenge! There are several parts to this. There are almost 100 test cases in these exercises. Like the last lecture, you can modify the testCases expression to limit which test cases are run. All except the last set are active to begin with. If you don't want to deal with all the error output, you can comment out tests in the `main` function and then reactivate them as you go along.

Individual Attributes

To start, observe the type `IndividualAttribute`. Each of these constructors represents a possible line we can parse for an individual. Fill in the functions associated with parsing each of these attributes, based on the format from the slides. You should, in general, allow for multiple spaces between elements on the same line, and you should parse in a case insensitive way. These functions all have a fairly generic type signature (`MonadParsec Void String`) so that they will work with any Parsec-like monad you specify later. Each of these parsers should consume the newline character at the end of the line (use the `eol` parsing function)!

1. `nameParser`: Parse an individual's given name (which may have spaces within it, but should not end in a space), as well as their family name, which should be between forward slashes. Names should only have alphabetical characters.
2. `sexParser`: Parse the `SexOption` for an individual
3. `famCParser`: Parse the family tag for which this individual is a "child". Do not return the `"@"` symbols as part of the tag. So for the example from the slides, you would return `"F1"`, not `"@F1@"`. Tags should only have alphanumeric characters (no spaces).
4. `famSParser`: Same as above, except the family for which this individual is a "spouse".

Now fill in the function `buildIndividualFromAttributes`. This should take a list of attributes and return an `Individual`, as long as the required fields are present. If they are not, return `Nothing`.

There are a couple helper functions, `trim` and `hspace` that can help you with these and other parts of the problem. The `trim` function will remove spaces from the end of a string. Then `hspace` will parse *horizontal* space characters (i.e. it will not consume newline characters).

Building an Individual

Using the tools from above, fill in `parseIndividual`. This will return a `String` for the individual's identifier tag, as well as the `Individual` itself constructed from the attributes. You'll need to parse the "definition" line for the individual in this function and then rely on your attribute parsers. Note that the return type of this function is **not** a `Maybe` value. If you cannot build the individual from the attributes, you can use the function `fail` to indicate that the parse has failed. This function takes a string as an argument.

Families

Follow the same process you used from parsing individuals to parse `Family` objects. Once again, there are attribute functions:

1. `parseHusband`
2. `parseWife`
3. `parseChild`
4. `parseMarriage`

And then you can combine them in `buildFamilyFromAttributes`, and use that result in `parseFamily`. The main difference is that there are no "required" fields for a family, so there is no need to deal with `Maybe` cases.

After you fill in this function, you should pass all the originally active tests. You can now update the `main` function and uncomment the final test group.

Bringing it Together

Now fill in the function `parseElement`. The type signature for this function is left up to you! It should either parse an individual or it should parse a family, trying both options. No matter which option it chooses, it should "incorporate" the new element in such a way that you'll ultimately construct the `FamilyTree` at the end. Pick a monad combination that helps you to do this! You can import any monads you like at the top, but there are some ideas there to get you started!

The last function to fill in is `parseFamilyTree`. This function takes an input `String` and returns a `Maybe FamilyTree`. You can return `Nothing` if there are any parse errors. Use `parseElement` as appropriate!

You can test your work now. Once you pass the tests, you've completed the challenge and the course!