Homework Assignment 1

Functional and Logic Programming, 2023

Due date: Thursday, May 4th, 2023 (04/05/2023)

Bureaucracy

- Submission is in pairs, but solo submission is also allowed. We very much suggest you pair-up, as solving this exercises with another person greatly enhances your learning (as well as being more fun!).
- To submit, create a zip file named `HW1_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW1_<id>.zip` if submitting alone.
- Make sure your submission compiles successfully. Submissions which do not compile will receive a 0 grade!
  - We will be using the following command to compile the file: `ghc -Wall -Werror HW1.hs`.

General notes

- The instructions for this exercise are split between this file and `HW1.hs`. This file offers a more high-level overview of the exercise, as well as offering a few hints. The Haskell file details all the required functionality for this assignment.
- You may not modify the **`import`** statement at the top of the file, nor add new **`import`**s.
  - You may however add new `LANGUAGE` pragmas at the top of the file.
  - If you are unsure what some function does, you can either ask HLS or Hoogle.
  - Hoogle also support module lookups, e.g., `Prelude.notElem`.
  - Do be aware however, that some functions from the standard library are more general than what we have learned so far in class.
    - And in some cases their definition may not be entire clear just yet!
- The exercises and sections are defined in a linear fashion. That is, it is a good idea to use previously defined functions (either from the same section or ones before it). It is also a good idea to use functions you saw in class; some of them are already imported, and some of them you would have to define yourself.

- Do not be alarmed by the large amount of functions! Many of them are simple one-liners, and were designed to aid you in solving the more complex functions.
- In general, you may define as many helper functions as you wish.

- Try to write elegant code, as taught in class. Use point-free style, η-reductions, and function composition to make your code shorter and more declarative. Prefer `foldr` over manual recursion where possible, and use functions like `map` and `filter` when appropriate. Although it is not required in the homework assignments, non-elegant code *will* be penalized in the test, so this is a good exercise. HLS and hlint can be very helpful in this.
  - Do note that in some cases, hlint may suggest functions which are not imported, or which you are trying to implement right now!
- If possible, please ask your questions first in Piazza, as this will allow all students to take part in the discussion.

Section 1: Warm-up

This section includes a bunch of simple utility functions for `Maybe` and `Either`. Most of these should be self evidents from the signature alone, but an example usage for each function is detailed below. Hints:

1. Do not confuse `mapMaybe` and `mapEither` with `maybeMap` and `eitherMap` taught in class. However, it might be a good idea to define and use them...
2. Most functions (or at least, most *patterns*) should be a single line!

Example usages

```
fromMaybe 1 Nothing
1
fromMaybe 1 (Just 2)
2
maybe 1 length Nothing
1
maybe 1 length (Just "foo")
3
catMaybes [Just 1, Nothing, Just 3]
[1,3]
mapMaybe (\x -> if x > 0 then Just $ x * 10 else Nothing) [1, -1, 10]
[10,100]

either length (*10) $ Left "foo"
3
either length (*10) $ Right 10
100
mapLeft (++ "bar") (Left "foo")
Left "foobar"
mapLeft (++ "bar") (Right 10)
Right 10
catEithers [Right 10, Right 20]
Right [10, 20]
-- If there are any Lefts, returns the first one encountered
catEithers [Right 10, Left "foo", Right 20, Left "bar"]
Left "foo"
mapEither (\x -> if x > 0 then Right $ x * 10 else Left $ x + 5) [1, 2, 3]
Right [10,20,30]
-- Returns the first Left
mapEither (\x -> if x > 0 then Right $ x * 10 else Left $ x + 5) [1, -1, 2, -2]
Left 4
concatEitherMap (Right . (* 10)) (Right 5)
Right 50
concatEitherMap (Right . (* 10)) (Left 5)
Left 5
partitionEithers [Right "foo", Left 42, Right "bar", Left 54]
([42,54],["foo","bar"])
```

Section 2: Lists and Zips

In this section, we will implement a few useful utility functions for lists. In particular, we implement a few `zip` functions. When working with lists, `zip`s combine elements from both lists, one element at a time, as if we are **zipping** both lists together. For example:

```
zipWith (+) [1, 2, 3] [4, 5, 6]
[5, 7, 9]
```

Usually, when one list is shorter than the other, we simply stop early.

```
zipWith (+) [1, 2] [4, 5, 6]
[5, 7]
```

`unzip` is the reverse operation of zipping.

```
unzip [(1, 2), (3, 4)]
([1, 3], [2, 4])
```

Note: the `zip` function is the very few places where using tuples is the right approach in Haskell!

Section 3: String interpolation

In this section, we will implement a basic string interpolation function. String interpolation, or template strings, is a technique used by modern programming langauges to enable easy string creation. For example, in JavaScript:

```
const x = 1
const y = 2
const z = x + y
`x = ${x}, y = ${y}, x + y = ${z}`
x = 1, y = 2, x + y = 3
```

We will implement a similar function in Haskell, which accepts a list of variables and a template string, and returns the interpolated string. The main entry point is the function `interpolateString`, however, we have split the task for you into multiple helper functions.

- A utility method for splitting a string on the **first occurrence** of a given character (`splitOn`)
  - If the character does not exist, return `Nothing`.
  - If the character exists, returns the prefix and the suffix of the string, without the character.

  ```
  splitOn 'x' "foobar"
  Nothing
  splitOn 'x' "fooxbar"
  Just ("foo", "bar")
  splitOn 'x' "foox"
  Just ("foo", "")
  splitOn 'x' "fooxfooxfoo"
  Just ("foo", "fooxfoo")
  ```

- `parseTemplate` splits a template string into separate plain and interpolated strings. If the template string is invalid, returns `Nothing`

- You may assume the characters '$', '{', and '}' only appear in the context of a string variable.
- If they appear in any other context, the string is considered to be invalid, and the parse function should return `Nothing`.

```
parseTemplate "Hello${world}!"
Just [PlainString "Hello",Variable "world",PlainString "!"]
parseTemplate "Hello${!" -- Unclosed variable
Nothing
parseTemplate "Hello$!" -- Invalid $
Nothing
```

Hint: Don't forget `String`s are really just `[Char]`, and you can pattern match on `Char` values:

```
parseTemplate ('$' : rest) = ???
```

- `assignTemplate` accepts the variable list—in the form of `[(String, String)]`, where the first element in the pair is the variable name and the second is the variable value—and the result of `parseTemplate`, and returns either the name of the first missing variable, or the fixed string.

```
assignTemplate [] []
Right ""
assignTemplate [] [PlainString "Hello!"]
Right "Hello!"
parsed = [PlainString "Hello ", Variable "name", PlainString "!"]
assignTemplate [("name", "Simon")] parsed
Right "Hello Simon!"
-- Variables are case sensitive!
assignTemplate [("Name", "Simon")] parsed
Left "name"
-- Return the first missing variable
assignTemplate [] [Variable "x", Variable "y"]
Left "x"
```

  - Hint: Use the `lookup` function from the standard library. Its (simplified) signature is:

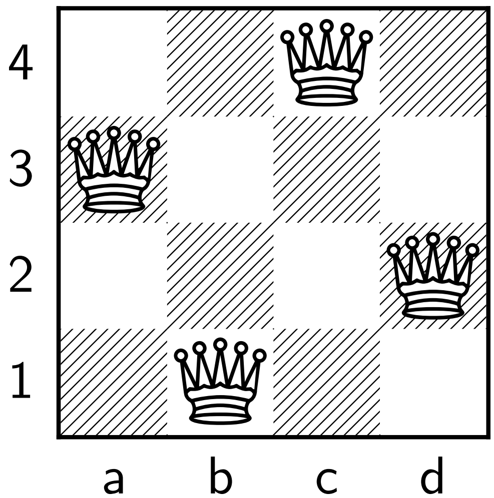
```
lookup :: String -> [(String, a)] -> Maybe a
```

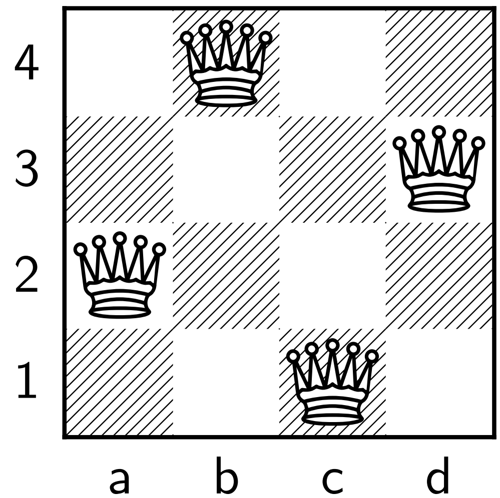
- The final function `interpolateString` combines the two previously defined functions. Note the different error types! `parseTemplate` returns `Nothing` on failure, `assignTemplate` returns `Left String`, and `interpolateString` returns `Left Error`. Use the functions from the first section and ones shown in class to be get around this.

```
interpolateString [("name", "Simon")] "Hello ${name}!"
Right "Hello Simon!"
interpolateString [("name", "Simon")] "Hello $name!"
Left InvalidTemplate
interpolateString [("Name", "Simon")] "Hello ${name}!"
Left (MissingVar "name")
```

Section 4: The queens puzzle

In this section we will solve the generalized case of the "Eight queens puzzle" for n queens. For $n < 1$, there are no solutions. For $n = 1$ there is only one trivial solution. For $n = 2$ or $n = 3$ there are no solutions. For $n = 4$, there are only two possible solutions:





For $n = 5$ there are 10 solutions. For $n = 6$ there are 4 solutions. (For the general list, see Number of ways of placing n-nonattacking queens on an n X n board. Note that for $n > 8$ the function may take a very long time to compute all solutions!)
As before, we split the problem into multiple utility functions. Note that in `splits`, the order of returned list elements is important, whereas in `permutations` and `queens` it is not important.

- The `queens` function should return all possible solutions, where each solution is a list of the (0-based) column of the queen in its i^{th} row.

```
queens 1
[[0]]
queens 3
[]
queens 2
[]
-- This solution order matches the diagram above.
queens 4
[[2,0,3,1],[1,3,0,2]]
```

- Hint: Each queen will necessarily be on a different column; perhaps the other functions in this section can help you create a list of all possible queen coordinates?

- Food for thought (unrelated to this exercise): Would it matter if `queens` returned a list of rows instead of a list of columns? Why or why not?