Homework Assignment 2

Functional and Logic Programming, 2024

Due date: Friday, May 10th, 2024 (10/05/2024)

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 `HW2_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW2_<id>.zip` if submitting alone.
  - You do not need special permission to submit alone.
- The zip file should contain **a single file** named `HW2.hs`!
- 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 HW2.hs`.
- You may submit the assignment after the due date even without approval (e.g., excluding reserve duty, serious illness, or other cases covered by the student administration).
  - You will be penalized for **5 points for every late day**.
  - The **maximum** extension allowed by this is **3 days**.
- If you don't know how to implement some function, **do not remove it**! Use `undefined` in the implementation, otherwise your entire submission will fail compilation, which will also result in a 0 grade.
  - This is especially true to for the bonus section!

General notes

- The instructions for this exercise are split between this file and `HW2.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.
  - N.B. HLS has the habit of adding **unnecessary imports** when it doesn't recognize an identifier, **so please double-check this before submitting**!
  - If you are unsure what some function does, you can either ask HLS or Hoogle.
  - Hoogle also supports module lookups, e.g., `Prelude.not`.
  - 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. 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!
- If possible, please ask your questions first in Piazza, as this will allow all students to take part in the discussion.

Section 1: `Maybe` and `Either` utility functions

This section includes a bunch of simple utility functions for `Maybe` and `Either`. Most of these should be self-evident 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

```
concatMaybeMap (\x -> if x > 0 then Just $ x * 10 else Nothing) Nothing
Nothing
concatMaybeMap (\x -> if x > 0 then Just $ x * 10 else Nothing) $ Just 10
Just 100
concatMaybeMap (\x -> if x > 0 then Just $ x * 10 else Nothing) $ Just (-10)
Nothing
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: List functions

In this section we will implement functions for lists.

```
take 0 [1, 2, 3]
[]
take 2 [1, 2, 3]
[1,2]
take 4 [1, 2, 3]
[1,2,3]
take 4 [1..]
[1,2,3,4]
drop 0 [1, 2, 3]
[1,2,3]
drop 2 [1, 2, 3]
[3]
drop 4 [1, 2, 3]
[]
take 5 $ drop 4 [1..]
[5,6,7,8,9]
takeWhile (< 1) [1..]
[]
takeWhile (< 5) [1..]
[1,2,3,4]
take 5 $ dropWhile (< 1) [1..]
[1,2,3,5]
take 5 $ dropWhile (< 5) [1..]
[5,6,7,8,9]

reverse []
[]
reverse [1, 2, 3]
[3,2,1]
rotate (-1) [1, 2, 3]
[1,2,3]
rotate 0 [1, 2, 3]
[1,2,3]
rotate 2 [1, 2, 3]
[2,3,1]
rotate 5 [1, 2, 3]
[2,3,1]
lotate (-1) [1, 2, 3]
[1,2,3]
lotate 0 [1, 2, 3]
[1,2,3]
lotate 2 [1, 2, 3]
[3,1,2]
lotate 5 [1, 2, 3]
[3,1,2]
-- Yields a list, equivalent to the one mentioned in HW1.
fromGenerator $ ((+ 1), (< 0), 0)
[]
fromGenerator $ ((+ 1), (<= 0), 0)
[1]
fromGenerator $ ((+ 1), (< 5), 0)
[1,2,3,4,5]
take 5 $ fromGenerator $ ((+ 1), const True, 0)
[1,2,3,4,5]
```

```
replicate 0 42
[]
replicate 3 42
[42,42,42]

inits [1, 2, 3]
[[],[1],[1,2],[1,2,3]]
take 5 $ inits [1..]
[[], [1], [1,2], [1,2,3],[1,2,3,4]]
tails [1, 2, 3]
[[1,2,3],[2,3],[3],[]]
map (take 3) $ take 5 $ tails [1..]
[[1,2,3],[2,3,4],[3,4,5],[4,5,6],[5,6,7]]
```

Section 3: Zips

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]
```

If we're not interested in stopping early, we have two valid options:

1. Providing a default value.
2. Failing.

```
zipFill 0 'a' [1,2,3] "foobar"
[(1,f),(2,o),(3,o),(0,b),(0,a),(0,r)]
zipFill 0 'a' [1..6] "foo"
[(1,f),(2,o),(3,o),(4,a),(5,a),(6,a)]
take 10 $ zipFill 1 'a' [1..] "foo"
[(1,f),(2,o),(3,o),(4,a),(5,a),(6,a),(7,a),(8,a),(9,a),(10,a)]

zipFail [1, 2] "foobar"
Left ErrorFirst
zipFail [1..] "foobar"
Left ErrorSecond
zipFail [1, 2, 3] "foo"
Right [(1,f),(2,o),(3,o)]
```

`unzip` is the reverse operation of zipping.

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

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

Knight's tour de force

In this section we will implement a version of the Knight's tour problem.

- The main entry point is the `tour` function, which will return a list of moves in an open tour (any tour would do), or `Nothing` if no tour is possible.
  - `KnightPos 0 0` is the top-left corner of the board.
  - Moving right increases `x`, moving down increases `y`.
  - For `KnightMove`, the first word signifies moving 1 in that direction, and the second word signifies moving 2 in that direction.
    - So `LeftBottom` means moving 1 left (`x` **decreases** by 1) and 2 down (`y` **increases** by 2), or from `KnightPos 2 3` to `KnightPos 1 5`.
  - You can assume the board is valid, i.e., it is at least 1x1, and that the starting position is within the board.
  - For boards of size 5×5, and a starting position of (0, 0), here is an example of one possible solution:

```
tour (Board 5 5) (KnightPos 0 0)
Just [RightBottom,TopRight,TopLeft,RightBottom,BottomRight,BottomLeft,
      TopLeft,RightTop,TopRight,RightBottom,LeftBottom,TopLeft,LeftTop,
      TopRight,BottomRight,LeftBottom,BottomLeft,LeftTop,TopRight,TopRight,
      LeftBottom,RightBottom,TopLeft,BottomLeft]
```

- You should also implement a couple of functions for translating `[KnightMove]` into a `[KnightPos]` and vice-versa. Note that you don't have to use these functions in your solution of `tour`.
  - To make things easier(?), `translate` can't fail, meaning it can support negative positions.

```
translate (KnightPos (-2) (-1)) [TopLeft, TopRight]
[KnightPos {x = -4, y = -2},KnightPos {x = -2, y = -3}]
translate (KnightPos 20 10) [BottomRight]
[KnightPos {x = 22, y = 11}]
```

  - While `translate'` also supports negative positions, it can fail—returning a `Left`—if it's impossible to move from one position to the next via a knight move, e.g., `[KnightPos 0 0, KnightPos 2 2]`.

```
translate' [KnightPos 0 0, KnightPos 1 2]
Right [RightBottom]
translate' [KnightPos 0 0, KnightPos 1 2, KnightPos 0 0, KightPos 2 1]
Right [RightBottom, LeftTop, BottomRight]
translate' [KnightPos 0 0, KnightPos (-1) (-2)]
Right [LeftTop]
translate' [KnightPos 0 0, KnightPos 2 2]
Left (InvalidPosition (KnightPos { x = 2, y = 2 }))
```

- Lastly, for a bonus of 10 points, you can implement `mark`, which marks knight positions on a board by step number.
  - For example, the mark for the above solution, after the appropriate translations, is

```
[[0,3,14,9,20],
 [13,8,19,2,15],
 [18,1,4,21,10],
 [7,12,23,16,5],
 [24,17,6,11,22]]
```

  - Note that the positions do not have to belong to a valid tour.
  - Any missing non-visited positions should be marked as `(-1)`.
  - If any position is outside the board bounds, return `Left InvalidPosition` with the first invalid position.
  - Also, if any position is visited more than once, return `Left InvalidPosition` with the first repeated position.

```
mark
  (Board 2 2)
  [KnightPos 0 0, KnightPos 1 1, KnightPos 0 1, KnightPos 1 0]
Right [[0,3],[2,1]]
mark
  (Board 2 2)
  [KnightPos 0 0, KnightPos 1 1]
Right [[0,-1],[-1,1]]
mark
  (Board 1 2)
  [KnightPos 0 0, KnightPos 1 1, KnightPos 0 1, KnightPos 1 0]
Left (InvalidPosition (KnightPos { x = 1, y = 1 }))
mark
  (Board 2 2)
  [KnightPos 0 0, KnightPos 1 1, KnightPos 1 0, KnightPos 1 1, KnightPos 0 0]
Left (InvalidPosition (KnightPos { x = 1, y = 1 }))
```

Hints:

- Use backtracking.
- The board does not have to be square
- For a board of size 1×1, the only possible tour is the knight standing still, so the result should be `Just []`.
- For boards larger than 6×6, it might take a while to compute the result. We will not check your implementation for performance, but it should be able to solve a 6x6 board in a reasonable time (less than 30 seconds).
- As a utility, the pre-defined `allKnightMoves` function returns a list of all enum values, similar to Java's `.values()` method for **`enum`**s.

- Consider defining and implementing a function to check if a move is valid from a given position: `Board -> KnightPos -> KnightMove -> Bool` or even `Board -> KnightPos -> KnightMove -> Maybe KnightPos`.
- You can use many of the utility functions you implemented in the previous sections.
- For the bonus function, you can use the following code to create a 5×5 board for example filled with `(-1)`:

```
initLists :: [[Int]]
initLists = replicate 5 $ replicate 5 (-1)
```

And the following code (in `ghci`) to print a `[[Int]]`:

```
toPrettyString = Prelude.unlines . map (Prelude.unwords . map Prelude.show)

putStrLn $ toPrettyString intLists
```