Homework Assignment 3

Functional and Logic Programming, 2024

Due date: Thursday, May 23rd, 2024 (23/05/2024)

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 `HW3_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW3_<id>.zip` if submitting alone.
  - You do not need special permission to submit alone.
- The zip file should contain **a single, top-level file** (no folders!) named `HW3.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 HW3.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 `HW3.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, so all students can take part in the discussion.

Special instructions

- This assignment is a little more free-form than the previous ones. This gives you greater freedom in how you implement the functions, and play around with functional design. If you get stuck, remember it's better to ask a human than a bot for help.
- In questions where you can assume the input is valid, it's fine to use `error`. This isn't very good design, but it's fine for an assignment.

Section 1: Tree serialization and deserialization

In this section, you will implement a couple of functions to serialize and deserialize binary trees. You are free to implement these as you see fit, so long as the following requirement holds:

- `deserialize . serialize` $\equiv$ `id`

In other words, calling `deserialize` after `serialize` always returns the original tree. Notes:

- The **exact** tree should be returned, i.e., it should not just contain the same values, but also in the exact **same position** in the tree.
  - `Tree (Tree Empty 1 Empty) 2 Empty` is **not** the same tree as `Tree Empty 1 (Tree Empty 2 Empty)`!
- These trees are not necessarily **search** trees, i.e., they can be unordered.
- The tree is not necessarily balanced, and it can contain any valid `Int`, including zero and negative numbers.
- You can assume the input to `serialize` is always a finite tree.
- In the implementation of `deserialize`, you may assume the input is always valid, i.e., it is the output of calling *your* `serialize` function on some tree.

Section 2: Infinite lists

In this section we will study infinite lists more in depth. Unlike regular Haskell lists, which may or may not be infinite, we will deal with exclusively infinite lists. Unlike regular lists `[a]`, which may or may not be infinite, we will define a list which is **necessarily** infinite.

```
-- (:>) Is the constructor in this case
data InfiniteList a = a :> InfiniteList a
infixr 5 :>
```

Since regular lists can also be infinite, it's possible to convert an infinite list to a regular list.

- Which gives us access to all the existing list functions!

```
sample :: InfiniteList a -> [a]
sample = take 10 . itoList
smallSample :: InfiniteList a -> [a]
smallSample = take 5 . itoList

sample $ irepeat 3
[3,3,3,3,3,3,3,3,3,3]
```

Some examples of the functions you should implement:

```
smallSample $ irepeat 1
[1,1,1,1,1]
smallSample $ iiterate (\ x -> x * x + x) 1
[1,2,6,42,1806]
sample naturals
[0,1,2,3,4,5,6,7,8,9]
itake 5 naturals
[0,1,2,3,4]
sample $ idrop 5 naturals
[5,6,7,8,9,10,11,12,13,14]
sample $ imap (* 3) naturals
[0,3,6,9,12,15,18,21,24,27]
sample $ iprepend [1,2,3] naturals
[1,2,3,0,1,2,3,4,5,6]
```

`ifilter` can result in a "finite" (or even empty!) list if a finite number of elements satisfy the predicate. Since the type itself is still an `InfiniteList`, this can cause programs to never halt:

```
sample $ ifilter even naturals
[0,2,4,6,8,10,12,14,16,18]
smallFilter = ifilter (< 5) naturals
smallSample smallFilter
[0,1,2,3,4]
sample smallFilter -- Will never halt!
```

Similarly, `iconcat` can also produce a "finite" list:

```
sample $ iconcat $ iiterate (map (+1)) [1,2,3]
[1,2,3,2,3,4,3,4,5,4]
smallConcat = iconcat $ [1] :> [2] :> [3] :> [4] :> [5] :> irepeat []
smallSample smallConcat
[1,2,3,4,5]
sample smallConcat -- Will never halt!
```

Unlike the finite version, `ifind` doesn't return `Nothing`, but it will never halt either if it can't find an element satisfying the predicate

```
ifind (> 10) naturals
11
ifind (< 0) naturals -- Will never halt!
```

Lastly, you should implement the following two functions for generating numbers. You are free to implement them however you wish, with the following requirements:

- `integers` and `rationals` should contain all possible `integers` and `rationals`, respectively.

  - Every integer number should appear at **some point** in `integers`, and likewise for rational numbers and `rationals`.
  - Or mathematically, for every $i \in \mathbb{Z}$, there should be an $n \in \mathbb{N}$ such that: `(itoList integers !! n) == i`, and likewise for $\mathbb{Q}$ and `rationals`, respectively.
    * `(!!) :: [a] -> Int -> a` is the list indexing operator.

- `integers` should have no repeats, but `rationals` can have repeats.
  - You can check for repeats in a **finite** list by comparing it with its `nub` version.

```
[1,3,2] == nub [1,3,2]
True
[1,3,2,3] == nub [1,3,2,3]
False
```

- The order of either list doesn't matter.

```
ifind (== 0) integers
0
ifind (\ x -> x > 20 && x < 22) integers
21
ifind (< (-20)) integers -- Any number smaller than -20 will do
-50
integerSample = itake 1000 integers
nub integerSample == integerSample
True
ifind (\x -> x < 9 % 2 && x > 10 % 3) rationals
-- It doesn't have to be this particular rational
4 % 1
ifind (\x -> x < -51 % 47 && x > - 52 % 43) rationals
(-6) % 5
```

Guidance:

- Use the `(%)` function to construct a `Rational` number, and `numerator` and `denominator` to extract the numerator and denominator from a `Rational`.
  - `(%)` already handles fracture reduction.

```
5 % 10
1 % 2
```

- `negate` works on any kind of number, including `Rationals`!

Bonus: $\mathbb{Q}$ with no repeats

For a bonus, you can implement `rationals'`, which like `rationals` contains all the rationals, but **without repeats**!
Hints:

- Take a look at Calikin—Wilf tree for inspiration, but don't forget you also need to support zero and negative numbers!
- Although your solution does not have to directly use trees, regular `Trees` can also be infinite. A function like `treeterate :: (a -> (a, a)) -> a -> Tree a` can be defined and implemented to generate such an infinite tree.

```
rationals'Sample = itake 1000 rationals'
nub rationals'Sample == rationals'Sample
True
ifind (\x -> x < 9 % 2 && x > 10 % 3) rationals
4 % 1
```

Section 3: Stack machines

Instructions

In this section, we will implement a simple stack machine parser. The stack machine will have the following instructions (instructions are case-**sensitive**):

- `PUSH n`—Pushes the integer `n` onto the stack.
- `POP`—Pops the top element from the stack.
- `SWAP`—Swaps the two top most elements on the stack.
- `DUP`—Duplicates the top element on the stack.
- `ADD`—Pops the top two elements from the stack, adds them, and pushes the result back onto the stack.
- `SUB`—Pops the top two elements from the stack, subtracts them (the bottom element from the top element), and pushes the result back onto the stack.
- `MUL`—Pops the top two elements from the stack, multiplies them, and pushes the result back onto the stack.
- `DIV`—Pops the top two elements from the stack, divides them (the top element is the numerator), and pushes the result back onto the stack.

Error handling

- If there was any error in parsing the instructions, i.e., the strings do not form valid instructions, you should use the `ParseError` constructor with the offending line.
- If the stack is empty when trying to `pop` an element or perform an operation, use `StackUnderflow` with the failing instruction and `Nothing`. If the stack contained a single element `e`, `Just e` should be used instead of `Nothing`.
- If trying to divide by zero, use `DivisionByZero`.

If there were no errors, you should just return the final stack.

Examples

Below are some examples. For clarity, each example contains a pseudo-file with a list of instructions, and the expected output.

```
PUSH 1
PUSH 2
Bad!
No!
Left (ParseError {line = "Bad!"})
```

For reference, the above would be represented as `"PUSH 1\nPUSH 2\nBad!\nNo!"`.

```
PUSH 2
POP
POP
Left (InstructionError
       (StackUnderflow {instruction = "POP", stackValue = Nothing}))
```

```
PUSH 0
PUSH 1
DIV
Left (InstructionError DivisionByZero)
```

```
PUSH 2
PUSH 1
SWAP
Right [2, 1]
```

```
PUSH 2
PUSH 4
MUL

PUSH 5
SUB
PUSH 3
Right [3, -3]
```

Guidance

- You can use the `readMaybe` function to parse strings into integers.

  ```
  readMaybe "1234" :: Maybe Integer
  Just 1234
  readMaybe "foo" :: Maybe Integer
  Nothing
  ```

  - `readMaybe` depends on the `Read` type class, and has the type signature:

    ```
    readMaybe :: Read a => String -> Maybe a
    ```

  - The compiler can infer the correct type, e.g., if you use it in combination with `maybeMap SomeConstructor` (where `SomeConstructor` is a constructor accepting `Int`), or you can explicitly define a helper function:

    ```
    readInteger :: String -> Maybe Int
    readInteger = readMaybe
    ```

- You can split a multi-line string using the imported library function `lines`.

  ```
  lines "foo\nbar\nbazz"
  ["foo","bar","bazz"]
  ```

  - Similarly, you can use `unlines` to convert a list of strings into a single string.

    ```
    unlines ["foo","bar","bazz"]
    "foo\nbar\nbazz\n"
    ```

- You should ignore empty lines.
- Split into smaller functions, as taught in class. For example, you might want to define an ADT for `Instruction`, and then define the following functions:

  ```
  parseInstructions :: String -> Maybe Instruction
  runInstruction :: [Int] -> Instruction -> Either StackError [Int]
  ```

- Remember lifting! It's easier to write functions whose input is a non-error type, and then lift them to work on `Maybe` and `Either` types.

  - Examples of lifting functions, from class and from the previous assignment:

```haskell
-- From the 3rd lecture chapter:
maybeMap :: (a -> b) -> Maybe a -> Maybe b
eitherMap :: (a -> b) -> Either e a -> Either e b
-- From Tutorial 3:
maybeToEither :: a -> Maybe b -> Either a b
-- From the previous assignment:
concatMaybeMap :: (a -> Maybe b) -> Maybe a -> Maybe b
concatEitherMap :: (a -> Either e b) -> Either e a -> Either e b
```

- To combine error handling with list traversal, you may wish to define and implement the following functions:

```haskell
-- If any mapped element yields Nothing, return Nothing.
maybeMapList :: (a -> Maybe b) -> [a] -> Maybe [b]
-- Like foldl, this function combines the *first element with the seed*.
-- If any element yields Left, return Left.
foldlEither :: (b -> a -> Either e b) -> b -> [a] -> Either e b
```

  - Using a bit of Type Tetris, you can easily compose all these functions to implement the `parseAndRun` function.