Homework Assignment 5

Functional and Logic Programming, 2023

Due date: Thursday, July 7, 2023 (07/07/2023)

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 `HW5_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW5_<id>.zip` if submitting alone.
- The zip file should contain **a single file** named `HW5.hs`!
  - Do not submit any other files, like `Deque.hs`, `State.hs`, or `text`
- 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 HW5.hs`.
- For any late submissions, please E-mail your submission directly to ofir.yaniv@post.runi.ac.il.

General notes

- You may not modify the **`import`** statement at the top of the file, nor add new **`import`**s.
  - 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 some cases their definition may not be entire clear just yet!
- You may not add any new `LANGUAGE` pragmas.
- The exercises and sections are defined linearly. 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.

  - In general, you may define as many helper functions, `type`s, and `class`es 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: `Applicative` and `Monad` instances

In this section we will implement `Monad` instances for two data structures we saw in class: `Deque` and `NonEmpty`. Expected behavior:

```
dq1 = 1 `DQ.pushl` (2 `DQ.pushl` DQ.empty)
dq2 = 30 `DQ.pushl` (40 `DQ.pushl` DQ.empty)
toList $ liftA2 (*) dq1 dq2
[30,40,60,80]
toList $ dq1 >>= (\x -> (x * 300) `DQ.pushl` ((x * 400) `DQ.pushl` DQ.empty))
[300,400,600,800]

ne1 = 1 :| [2]
ne2 = 30 :| [40]
toList $ liftA2 (*) ne1 ne2
[30,40,60,80]
toList $ ne1 >>= (\x -> (x * 300) :| [x * 400])
[300,400,600,800]
```

Tips and hints:

- We already supplied the 4 **`instance`**s from `HW4` for `Deque` and the same sans `Monoid` for `NonEmpty`. You might be able to use these implementations to implement `(>>=)`.
  - Many of the functions you implemented in HW4 for `Foldable`, e.g., `toList`, `elem`, are imported from `Data.Foldable`.
- `Monad` of course require an `Applicative` **`instance`**. You can either implement it on your own as an exercise, or use the default implementation shown in class, e.g.,

  ```
  instance Applicative Deque where
    liftA2 f fa fb = fa >>= (\a -> f a <$> fb)
    pure = return
  ```

  Of course, if you do use the default, you can't use either `pure` or `liftA2` in your implementation of `return` and `(>>=)`!
  - If you implement `Applicative`, the behavior of `liftA2` should mimic Cartesian product, as it was shown for lists.

Section 2: `IO` scripts

In this section we will implement a simple script in Haskell. The function `joinGrades` receives three `FilePath`s as input.

1. The first is a file containing lists of students and their respective groups. Each line in the file is in the pattern of `<student_id>,<group_id>`. A student belongs to exactly one group, but a single group can contain one, two, or more students.
2. The second file contains lists of groups and grades. Each line in the file is in the pattern of `<group_id>,<grade>`. Every group can have at most a single grade, but not all groups have grades.

3. The last file is the output file. Each line in the file should be in the pattern of `<student_id>,<grade>`. You should write into it a list of all students and their grades. A student which belongs to a group with no grade should receive the grade 0.

For example, suppose the file `/tmp/groups.txt` looks like this:

```
1234,AB
1235,AB
1236,AB
1237,AC
1238,AD
```

And the file `/tmp/grades.txt` looks like this:

```
AC,99
AB,100
```

Then after running from GHCi `joinGrades "/tmp/groups.txt" "/tmp/grades.txt" "/tmp/output.txt"`, the file `/tmp/output.txt` should contain the following lines (in any order):

```
1234,100
1235,100
1236,100
1237,99
1238,0
```

Notes:

- You can assume all the input is valid.
- You can treat all elements as strings.
- You can use `lines`/`unlines` to handle splitting and joining strings.
- Like always with `IO`, you should write most of your code—parsing lines, matching students, groups, and grades, etc.—without using `IO` at all, and then just wrap things up with `readFile` and `writeFile`.

Section 3: Guessing game

In this section we will implement a very simple guessing game: The user picks a number between some minimum and maximum value (inclusive), and the program tries to guess the correct number. To do that, the program should choose some number in the range between the minimum and maximum, and ask the user if the number they chose is lower, greater, or equal to the number. You can assume the user will always answer with either `l/e/g` followed by the `ENTER` key, so you can read a single char using `getLine` and pattern match on `"l"/"g"/"e"`. Finally, the function should return the number the user chose.

Example running of `guessingGame 0 2000` from `GHCi`:

```
*HW5> guessingGame 0 10
Please pick a number between 0 and 10
Is the number less than, equal, or greater than 5? (l/e/g)
l
Is the number less than, equal, or greater than 2? (l/e/g)
e
2

*HW5> guessingGame 0 2000
Please pick a number between 0 and 2000
Is the number less than, equal, or greater than 1000? (l/e/g)
g
Is the number less than, equal, or greater than 1500? (l/e/g)
g
Is the number less than, equal, or greater than 1750? (l/e/g)
l
Is the number less than, equal, or greater than 1625? (l/e/g)
g
Is the number less than, equal, or greater than 1687? (l/e/g)
g
Is the number less than, equal, or greater than 1718? (l/e/g)
g
Is the number less than, equal, or greater than 1734? (l/e/g)
l
Is the number less than, equal, or greater than 1726? (l/e/g)
g
Is the number less than, equal, or greater than 1730? (l/e/g)
l
Is the number less than, equal, or greater than 1728? (l/e/g)
g
1729
```

Section 4: `State` evaluation

In this section we will implement an important feature for the calculator we saw in the third chapter: the ability to assign values to new variables. The code discussed in class for expressions is already provided for you in `Calculator.hs`. Our goal is to support calculations like the following:

```
x = 1 + 3 -- x is assigned 4
y = (3 * x) / 2 -- y is assigned 6
x = 42 + y -- x is assigned the value of 48
z = (4 * y) - x -- z is assigned the value of -24
```

After running all these computations, we will end up with an integer value for every identifier: `x = 48`, `y = 6`, and `z = -24`. The function `runCalculator` receives a list of variable names and expressions, and goes over them one by one. For example, we can encode the above computations as:

```
[ ("x", Plus (Literal 1) (Literal 3))
, ("y", Division (Mult (Literal 3) (Identifier "x")) (Literal 2))
, ("x", Plus (Literal 42) (Identifier "y"))
, ("z", Minus (Mult (Literal 4) (Identifier "y")) (Identifier "x"))
]
```

As we discussed, expressions can fail for multiple reasons: we could divide by zero, or reference an undefined variable.

```
[ ("x", (Literal 0))
, ("y", Plus (Literal 3) (Identifier "a")) -- Whoops
, ("z", Division (Literal 42) (Identifier "x") -- Whoops
]
```

To make our function `runCalculator` feature complete, we also want to keep track of all the errors we encountered: the number of divisions by zero, and the number of times each variable was missing, reported by to the result of `evaluate`. Note that if an assignment fails because of an invalid expression, we will **remove** the previous assignment of the identifier.
Hint: Define a helper function with the signature of `(String, Expression) -> State Result ()`, and use a combination of `traverse` and `execState`.
Example run of the program:

```
-- Already defined in State.hs
execState :: State s a -> s -> s
execState = snd .: runState

list =
  [ ("x", Literal 0)
  , ("z", Division (Literal 42) (Identifier "x"))
  , ("x", Plus (Literal 3) (Identifier "a"))
  -- x will be missing here, since it failed above.
  , ("y", Plus (Identifier "x") (Identifier "x"))
  , ("x", Plus (Literal 3) (Literal 4))
  , ("y", Plus (Identifier "x") (Identifier "x"))
  -- An assignment can reference itself.
  , ("x", Mult (Identifier "x") (Identifier "x"))
  ]

runCalculator list
-- New lines don't matter for the result:
Result {
  finalValues = fromList [("x",49),("y",14)],
  missingVariables = fromList [("a",1),("x",1)],
  divisionByZero = 1}
```