The University of Hong Kong

COMP3258: Functional Programming

Assignment 3

Deadline: 23:59, April 14, 2017 (HKT)

Notice:

1. Only the following libraries are allowed to import in this assignment.

```
import          Data.Char
import          Data.List
import          Parsing
import          System.IO
```

2. Please submit a single Haskell file, named as A3_XXX.hs, with XXX replaced by your UID, and follow all the type signatures strictly.

3. The last 3 problems can be implemented by one function, you could write a comment in the code to tell me which of those you finished.

1 Expression Trees

In Haskell, a famous data type called `Maybe` is used to represent "a value or nothing". If an expression `e` has type `Maybe a`, then `e` is either `Just a` (just a value of type `a`) or `Nothing`, as demonstrated by the definition below.

```
data Maybe a = Just a | Nothing
```

`Maybe` describes values with a context of possible failure attached. For example, if a function returns `Maybe Int`, it means the function may fail when calculating the result integer.

Problem 1. (5 pts.) Implement a function `lift` which has the type below.

```
lift :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
```

`lift` takes a binary function and two `Maybe` values. If either of the values is `Nothing`, then `lift` returns `Nothing`. Otherwise, `lift` uses the input function to combine them, and returns the result wrapping by `Just`.

Expected running results:

```
*Main> lift (+) (Just 1) Nothing
Nothing
*Main> lift (+) (Just 1) (Just 2)
Just 3
*Main> lift (++) (Just "abc") (Just "def")
Just "abcdef"
*Main> lift (++) Nothing (Just "def")
Nothing
```

Let's consider the following `Expr` data type for expression trees.

```
data Expr
  = Add Expr Expr
  | Sub Expr Expr
  | Mul Expr Expr
```

```
  | Div Expr Expr
  | Mod Expr Expr
  | Val Int
  | Var String
  deriving (Eq, Show)
```

It has addition `Add`, subtraction `Sub`, multiplication `Mul`, division `Div`, modulo `Mod`, together with integer literal `Val` and variable `Var`.

We use an environment `Env` to determine the values for variables:

```
type Env = [(String, Int)]
```

The library function `lookup` could be used for searching in an environment.

Problem 2. (15 pts.) Implement a function `eval :: Env -> Expr -> Maybe Int` to evaluate expression trees. `eval` should return `Nothing` if the divisor is 0 in the division and modulo cases. Also, if a variable cannot be found in the environment, `Nothing` should be returned.

Expected running results:

```
*Main> eval [] (Add (Val 2) (Val 3))
Just 5
*Main> eval [("x", 2)] (Add (Var "x") (Val 3))
Just 5
*Main> eval [("x", 2)] (Add (Var "y") (Val 3))
Nothing
*Main> eval [] (Div (Val 4) (Val 2))
Just 2
*Main> eval [] (Mod (Val 4) (Val 0))
Nothing
```

2 Parsing Expressions

Then let's write a parser for those expression trees. You may want to review Tutorial 5 and the lecture slides when doing this section.

Problem 3. (20 pts.) Implement a function `pExpr :: Parser Expr` for parsing `Expr`s. The grammar is provided as below:

$$
\begin{aligned}
expr &:= term\ op\_term \\
op\_term &:= (\text{'+'} \mid \text{'-'})\ term\ op\_term \mid \epsilon \\
term &:= factor\ op\_factor \\
op\_factor &:= (\text{'*'} \mid \text{'/'} \mid \text{'\%'})\ factor\ op\_factor \mid \epsilon \\
factor &:= \text{'('}\ expr\ \text{')'} \mid integer \mid identifier
\end{aligned}
$$

For now, you can assume the identifiers start with a lower case letter, and may contain any alphabetic or numeric characters after the first one. We will extend it later.

Notice:

- Use the `token` function in `Parsing.hs` to remove leading and trailing spaces.
- Your parser should reflect the left-associativity of the operators. See the second example below.

Expected running results:

```
*Main> parse pExpr "1 + 2"
[(Add (Val 1) (Val 2),"")]
*Main> parse pExpr "1 + 2 + 3"
[(Add (Add (Val 1) (Val 2)) (Val 3),"")]
*Main> parse pExpr "1 + x"
[(Add (Val 1) (Var "x"),"")]
*Main> parse pExpr "1 + x * 3"
[(Add (Val 1) (Mul (Var "x") (Val 3)),"")]
*Main> parse pExpr "1 + x * 3 / 5"
[(Add (Val 1) (Div (Mul (Var "x") (Val 3)) (Val 5)),"")]
```

Problem 4. (5 pts.) Implement a function `runParser :: Parser a -> String -> Maybe a`. `runParser` runs a given parser to parse the **full** string and returns the first result. `Maybe` implys the parser may fail.

Notice:

- Return `Nothing` when the result list is empty.
- Return `Nothing` when the parser only consumes part of the input (the second component of the pair is not empty, see the examples below).

Expected running results:

```
*Main> runParser (char 'a') "a"
Just 'a'
*Main> runParser (char 'a') "ab"
Nothing
*Main> runParser pExpr "1+2"
Just (Add (Val 1) (Val 2))
*Main> runParser pExpr "1++"
Nothing
```

3 MiniExcel using `IO`

In this section we are going to develop a very simple version of Excel. Before continue, paste the `intToColumn` and `columnToInt` functions from Assignment 1 in your code as auxiliary functions.

First, the `Table` type below is used to represent a table. We only consider intergers. In each cell, there's a value of type `Maybe Int`. It's `Just x` if the integer value is `x`, or `Nothing` if the cell is empty (not filled).

```
type Table = [[Maybe Int]]
```

Problem 5. (15 pts.) Implement a function `printTable :: Table -> IO ()` which prints a table.

For a 5 by 5 empty table, `printTable` prints:

```
 |A|B|C|D|E|
1| | | | | |
2| | | | | |
3| | | | | |
4| | | | | |
5| | | | | |
```

- The topmost row contains the column titles.
- The leftmost column contains the row numbers.
- Cells are separated by a single bar `|`.

After filling some numbers in the table, it's printed as:

```
 |A| B|  C| D|E|
1|0|10|   |  | |
2| |  | 34|  | |
3| |  |   |  | |
4| |  |   |40| |
5| |  |100|  | |
```

- Each cell is right-aligned, including the header cells.
- The width of a column is determined by the widest cell in that column. For example, column `C` has width 3 because 100 (position `C5`) is in it.
- Only those cells contain numbers are displayed, empty cells (with `Nothing` in it) are blank.

Notice:

- Please follow the format.
- You can assume the table is at least 1 by 1 (i.e. has at least one row and one column).

Expected running results:

```
*Main> printTable [[Nothing]]
 |A|
1| |
*Main> printTable [[Just 1, Nothing], [Nothing, Just 2]]
 |A|B|
1|1| |
2| |2|
*Main> printTable [[Just 1, Nothing, Nothing],
                   [Nothing, Nothing, Just 3],
                   [Just 1, Just 2, Just 3]]
 |A|B|C|
1|1| | |
2| | |3|
3|1|2|3|
```

Problem 6. (20 pts.) Implement a interactive program `main :: IO ()`.

The program firstly asks the user to input number of rows and number of columns. It keep asking the user when the input is invalid (not a positive number).

Example 1:

```
*Main> main
Number of rows: 3
Number of columns: 3

>
```

Example 2:

```
*Main> main
Number of rows: 5
Number of columns: a
Number of columns: abc
Number of columns: 4

>
```

After that, it prints a prompt `>` and wait for a command. Please use `putStr "\n> "` for the prompt (it will print a newline before the `>` symbol and a space after it).

Then the program enters a loop, like the GHCi REPL (Read-Eval-Print-Loop). It takes a command, executes it, and then asks for a new command and repeats again, until the `quit` command is entered.

You need to support 3 types of command:

1. `table` to print the table.
2. Value assignments for cells such as `C1 = 10` and `C2 = C1 + 3 * A2`.

   - All the operations in the first section should be supported.
   - After an command `cell_name = expression`, it should echo `cell_name = value` where the value is the evaluated result of the expression. See examples below.
   - After an assignment, the value will be displayed at the specific position for command `table` next time.
   - The later assignment override the old value in the cell.

3. `quit` to quit the loop.

Example 1:

```
*Main> main
Number of rows: 4
Number of columns: 4

> table
 |A|B|C|D|
1| | | | |
2| | | | |
3| | | | |
4| | | | |

> A1 = 100
A1 = 100
```

```
> A2 = A1 + 10
A2 = 110

> table
 |  A|B|C|D|
1|100| | | |
2|110| | | |
3|   | | | |
4|   | | | |

> C3 = A1 + A2 + 500
C3 = 710

> table
 |  A|B|  C|D|
1|100| |   | |
2|110| |   | |
3|   | |710| |
4|   | |   | |

> quit
```

Example 2:

```
*Main> main
Number of rows: 3
Number of columns: 3

> A1 = 10
A1 = 10

> table
 | A|B|C|
1|10| | |
2|  | | |
3|  | | |

> A1 = A1 * 2
A1 = 20

> table
 | A|B|C|
1|20| | |
```

```
2|  | | |
3|  | | |

> quit
```

If any illegal case occured, just print `Error`. Those cases include:

- Cell name out of range.
- Illegal command.
- The expression on the right hand side evaluates to `Nothing`.

See this example:

Example 3:

```
*Main> main
Number of rows: 3
Number of columns: 3

> no such command
Error

> Z100 = 10
Error

> A1 = 1 % 0
Error

> quit
```

Hints:

- In some cases the prompt doesn't show up correctly, you may need to add the following lines at the beginning of your `main` function.

  ```
  hSetBuffering stdin LineBuffering
  hSetBuffering stdout NoBuffering
  ```

- You may need to modify the parser of identifiers before to support variables that start with an upper case letter (which is the column title).

- You may need to build a simple parser for parsing the commands.

Problem 7. (5 pts.) Support a new type of command `del cell_name` which clear a cell by setting the value to `Nothing`. The program should print `Deleted cell_name`, or `Error` if no such cell.

Example:

```
*Main> main
Number of rows: 3
Number of columns: 3

> A1 = 10
A1 = 10

> table
 | A|B|C|
1|10| | |
2|  | | |
3|  | | |

> del A1
Deleted A1

> table
 |A|B|C|
1| | | |
2| | | |
3| | | |

> del Z1
Error

> quit
```

Problem 8. (10 pts.) Support auxiliary variables that start with a lower case letter. They can be added, deleted, and used in the expressions. Add a new command `vars` which prints all these variables, in alphabetical order (Library function `sort` can be used here).

Example 1:

```
Number of rows: 3
Number of columns: 3

> a = 10
```

```
a = 10

> b = a * 2
b = 20

> C1 = b + a
C1 = 30

> table
 |A|B| C|
1| | |30|
2| | |  |
3| | |  |

> vars
a = 10
b = 20

> del b
Deleted b

> vars
a = 10

> quit
```

Example 2:

```
*Main> main
Number of rows: 3
Number of columns: 3

> b = 10
b = 10

> a = 20
a = 20

> c = 30
c = 30

> aa = 10
aa = 10
```

```
> vars
a = 20
aa = 10
b = 10
c = 30

> quit
```

If the expression refers to an undefined variable, print `Error` as before.

Example 3:

```
*Main> main
Number of rows: 3
Number of columns: 3

> a = 10
a = 10

> vars
a = 10

> c = d
Error

> quit
```

Code style and submission (5 pts.)

All functions should be implemented in a single Haskell file, named as A3_XXX.hs, with XXX replaced by your UID. Your code should be well-written (e.g. proper indentation, names, and type annotations) and documented. Please submit your solution on Moodle before the deadline.

Plagiarism

Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.