The University of Hong Kong

COMP3258: Functional Programming

Assignment 2

Deadline: 23:59, Nov 17, 2017 (HKT)

Notice:

1. Only the following libraries are allowed to be imported in this assignment. `Parsing` is the code for Lecture that can be found in the Moodle.

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

2. Please submit a single Haskell file, named as A2_XXX.hs, with XXX replaced by your UID, and follow all the type signatures strictly. **No need** to submit `Parsing.hs`. In the case that Moodle rejects your .hs file, please submit it as A2_XXX.zip, which includes only one file A2_XXX.hs.

3. If only partial functionality is supported for REPL part, please give one line comment explaining which commands are supported.

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.

```haskell
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. We will make use of `Maybe` to deal with expression trees.

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

```haskell
data Binop = Add | Sub | Mul | Div | Mod
  deriving (Eq, Show)

data Expr
  = Bin Binop Expr Expr
  | Val Int
  | Var String
  deriving (Eq, Show)
```

It might be a little bit different from the expression tree you have seen, because we are going to reuse `Binop` later. `Binop` stands for binary operations, including addition `Add`, subtraction `Sub`, multiplication `Mul`, division `Div`, modulo `Mod`. `Expr` contains all binary operations over expressions, together with integer literal `Val` and variable `Var`.

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

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

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

Problem 1. (10 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 [] (Bin Add (Val 2) (Val 3))
Just 5
*Main> eval [("x", 2)] (Bin Add (Var "x") (Val 3))
Just 5
*Main> eval [("x", 2)] (Bin Add (Var "y") (Val 3))
Nothing
*Main> eval [] (Bin Div (Val 4) (Val 2))
Just 2
*Main> eval [] (Bin 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 4 and the lecture slides when doing this section.

Problem 2. (20 pts.) Implement a function `pExpr :: Parser Expr` for parsing `Expr`s. The grammar is provided as below:

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

You can assume the identifiers start with a lower case letter, and may contain any alphabetic or numeric characters after the first one.

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

Problem 3. (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 (Bin Add (Val 1) (Val 2))
*Main> runParser pExpr "1++"
Nothing
```

3 Compilation

Instead of defining `eval` directly, we can give expressions meaning by compiling expressions onto a simple stack machine.

Consider that the stack machine supports the following instructions:

```haskell
data Instr = IVal Int | IBin Binop | IVar String
  deriving (Eq, Show)
```

Instructions contains integer literal `IVal`, variable `Var`, and binary operations `IBin`. The evaluation of instructions are based on a simple stack, which is modeled as a list of integers:

```haskell
type Stack = [Int]
```

Instruction `IVal i` will push `i` into the stack, and `IVar x` will push the value of variable `x` (if it is in the environment) into stack. `IBin op` will pop two values from the stack, and apply the binary operator to them, and then push the result into the stack.

A program is a list of instructions.

```haskell
type Prog = [Instr]
```

Problem 4. (10 pts.) Implement a function `runProg :: Prog -> Env -> Maybe Int`. `runProg` runs a program with an empty stack at the beginning. It should end with exactly one number in the stack. In those cases, return `Just value` where `value` is the only number in the stack. Return `Nothing` if evaluation fails:

- Not enough numbers to apply for binary operations.
- A divisor is 0 in the division and modulo cases.
- A variable cannot be found in the environment.
- After evaluation, the stack has less or more than one number.

Expected running results:

```
*Main> runProg [IVal 1, IVal 2, IBin Add] []
Just 3
```

```
*Main> runProg [IVal 2, IVal 4, IBin Div] []
Just 2
*Main> runProg [IVal 3, IBin Add] []
Nothing
*Main> runProg [IVal 2, IVal 4, IBin Mul, IVal 1] []
Nothing
*Main> runProg [IVal 2, IVar "x", IBin Sub] [("x", 3)]
Just 1
```

Problem 5. (10 pts.) Implement a function `compile :: Expr -> Prog` that compiles an expression into a program that can be run in the stack machine.

```
*Main> compile (Bin Sub (Val 2) (Val 1))
[IVal 1,IVal 2,IBin Sub]
*Main> compile (Bin Sub (Val 2) (Var "x"))
[IVar "x",IVal 2,IBin Sub]
*Main> compile (Bin Sub (Val 2) (Bin Add (Val 1) (Var "x")))
[IVar "x",IVal 1,IBin Add,IVal 2,IBin Sub]
```

With all those functions defined so far, you should be able to verity that the two approaches always give the same result:

- Direct evaluation over an expression
- First translate the expression into a program on a simple stack machine, and then run the program.

```
*Main> let Just x = runParser pExpr "2-1"
*Main> x
Bin Sub (Val 2) (Val 1)
*Main> eval [] x
Just 1
*Main> compile x
[IVal 1,IVal 2,IBin Sub]
*Main> runProg (compile x) []
Just 1
```

4 Optimization

The compilation in practice can be very complicated. In order to produce efficient machine programs, there are usually many optimization heuristics. One of the simplest heuristics is *Constant folding*, namely, calculation between constants is calculated directly during compilation time instead of at runtime.

Problem 6. (10 pts.) Implement a function `optimize :: Expr -> Maybe Expr` that optimizes an expression according to the following rules:

- Multiplication between any expression `e` and `0` is simplified to `0`.
- Addition between any expression `e` and `0` is simplified to `e`.
- Subtraction an expression `e` by `0` simplified to `e`.
- Division or Modulo by `0` returns `Nothing`.
- Any evaluation between constants are calculated directly.

Expected running results:

```
*Main> optimize $ Bin Add (Var "x") (Bin Sub (Val 2) (Val 1))
Just (Bin Add (Var "x") (Val 1))
*Main> optimize $ Bin Add (Val 3) (Bin Sub (Val 2) (Val 1))
Just (Val 4)
*Main> optimize $ Bin Add (Val 3) (Bin Mul (Var "x") (Val 0))
Just (Val 3)
*Main> optimize $ Bin Add (Var "x") (Val 0)
Just (Var "x")
*Main> optimize $ Bin Add (Var "x") (Val 1)
Just (Bin Add (Var "x") (Val 1))
*Main> optimize $ Bin Div (Val 3) (Val 0)
Nothing
```

You can verity that the evaluation result of an optimized expression should be the same as the evaluation result of the original expression.

5 REPL: Read-Eval-Print Loop

In previous programs, we are required to pass an environment to the evaluation process manually, which is quite inconvenient. Also there is no way for us to change the value of a variable during the calculation. In this section we are going to develop a very simple REPL to compute the value of an expression. A REPL is an interactive program, which reads the input, executes the input, prints the result and waits for the next input. For example, `GHCi` is the REPL for Haskell.

The REPL stores values of variables in `Env`, therefore we can directly refer to those variables during calculation. To hide the details about IO, you can paste the following code to the assignment:

```haskell
main :: IO ()
main = do
  hSetBuffering stdin LineBuffering
  hSetBuffering stdout NoBuffering
  repl []

repl :: Env -> IO ()
repl env = do
  putStr "\n> "
  line <- getLine
  dispatch env line

dispatch :: Env -> String -> IO ()
dispatch = error "TODO"
```

If you enter `main` in `GHCi`, it enters the REPL with an empty environment. The REPL prints a prompt `>` and waits for the command from the user.

```
*Main> main

>
```

You job is to implement the `dispatch` method that takes the current environment and the command from the user, executes the command, prints the result, and goes back to REPL. The following functions are provided to take care of IO things for you:

```haskell
quit :: IO ()
quit = return ()

loop :: String -> Env -> IO ()
loop str next = do
  putStrLn str
  repl next
```

When you call the function `quit` from `dispatch`, it will exit the REPL. You can pass a `String` and current environment to `loop`, which will print the `String` for you and continue to REPL with the environment you provided.

Problem 7. (30 pts.) You need to support 6 types of commands. If any illegal case occured, just print `Error` and continue to REPL. For example, illegal command.

Hints:

- You may need to build a simple parser for parsing the commands.
- Use `\n` in string to print a blank line.

1. `quit` to quit the loop.

Example 1:

```
*Main> main

> no such command
Error

> quit
*Main>
```

2. Support variable assignment `let var = expression`, such as `let a = 10` and `let b = a + 10`.

   - `expression` is the `Expr` in Problem 1.
   - If the expression on the right hand evaluates to `Nothing`, then print `Error`, and keep the environment unchanged.
   - After command `let x = expression`, it should echo `x = value` where the value is the evaluated result of the expression. See examples below.
   - A new assignment creates an new entry in the environment and the variable can be used in any later command.
   - The later assignment override the old value in the environment.

Example 2:

```
*Main> main

> let a = 10 + 10 * 2
a = 30

> let b = a * 2
b = 60

> let b = 5 / 0
Error

> let d = c
Error

> let b = 10
b = 10

> let e = b + 5
e = 15

> quit
*Main>
```

3. `env` to print the current environment in alphabetical order (Library function `sort` can be used here)

- Notice that new assignment overrides the old one, so there shouldn't be duplicated entries for a same variable.
- Print empty string `""` if environment is empty.
- The format to print is `var = value` per line. Don't print extra blank line after the final line.

Example 3:

```
*Main> main
> env


> let a = 10
a = 10

> let b = a / 2
```

```
b = 5

> env
a = 10
b = 5

> let a = 5 + 1 * 2
a = 7

> env
a = 7
b = 5

> quit
*Main>
```

4. `var x` to enquiry the value of `x` in the environment.

- Print `x = value` for enquiry.
- If the variable is not in environment, print `Error`.

Example 4:

```
*Main> main

> let a = 10 + 20
a = 30

> var a
a = 30

> var b
Error

> let a = 3 * 4
a = 12

> var a
a = 12

> quit
*Main>
```

5. `del x` to delete `x` from the environment.

- Print `Deleted var` where `var` is the real variable name.
- Print `Error` if the variable does not exist in the environment.

Example:5

```
*Main> main

> let a = 10

> var a
a = 10

> del a
Deleted a

> var a
Error

> del b
Error

> let b = 10
b = 10

> quit
```

6. Direct evaluation of any expressions from first section, for example, `1 + 2`, `x * 2`.

- Print `ans = value`, where value is the calculated result.
- All the operations in the first section should be supported.
- The expression can make use of existing variables in the environment.
- If the expression on the right hand evaluate to `Nothing`, then print `Error`.

Example 6:

```
*Main> main

> 1 + 2 + 3 + 4
ans = 10

> 1 + 2 * 2
ans = 5

> let a = 10
```

```
a = 10

> a - 1 - 2
ans = 7

> 5 / 0
Error

> b + 1
Error

> let b = 2
b = 2

> a * b
ans = 20

> quit
```

Code style and submission (5 pts.)

All functions should be implemented in a single Haskell file, named as A2_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. In the case that Moodle rejects your .hs file, please submit it as A2_XXX.zip, which includes only one file A2_XXX.hs. Please at least make sure your file can compile!

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.