CS2006 Practical 1   130010944   140014861   **130011035**   Tutor: Adam Barwell 17/02/15

CS2006 Haskell Practical 1: Calculator

Introduction

As an introduction to the core features of the Haskell language, and the functional programming paradigm, the group was presented with the task of building an interactive calculator within Haskell.

Functionality

This section details in brief the completed aspects of our calculator application:

Basic Requirements

- The minimum requirements to handle addition have been successfully implemented.
- The command “:q” can be used to exit the calculator.
- The parser has been extended so that it now successfully handles addition, subtraction, multiplication and division.
- Variable assignment using a “Set” command is fully supported.
- All three features of the evaluator State; variable values, number of calculations and the command history have been included.
- The parser has been improved so that it supports multiple digit numbers and whitespace.
- The implicit “it” variable is implemented correctly.
- The calculator implements a command for accessing command history, using the “!n” will return the *n*th most recent command.

Additional Requirements

- The parser has been extended to support negative numbers.
- Support of additional functions like abs, mod and power have been added.
- The calculator now supports floats only, and as a result the mod function is no longer supported.

Design

List of files

- Expr.hs - contains the basic expression types, a complete parser and a complete evaluator for expressions.

- Repl.hs - contains the main loop of the calculator application and defines processes that add functionality to the calculator.
- Main.hs - initialises the repl.
- Parser.hs - contains the parsing library.

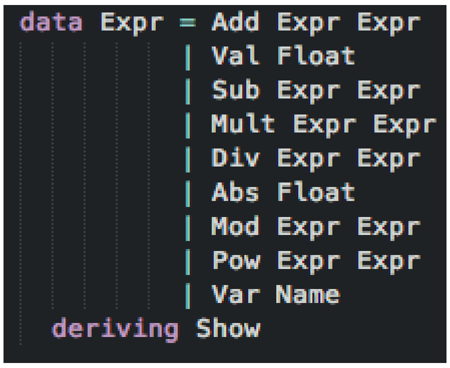
Only Expr.hs and Repl.hs were modified heavily. Parsing.hs was modified to support floating point numbers.
In Expr.hs, we used different aspects of the Parser to make use of negative numbers.

Functions in REPL.hs

Contains processes such as *Set*, *Eval*, *Hist* and *Quit*.
The *addVars* function adds variables when the *Set* command is invoked.
The *updateVars* function updates the value of a variable if it already exists.
Each calculation is stored in the memory (*addHistory)* and can be accessed using ***'!'*** in front of an integer that signifies how recent the command was number (*retrieveHistory)*.
Contains main IO loop so that the program runs until the quit command is encountered.

Implementation

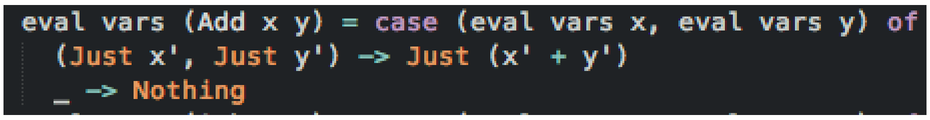
Expr.hs



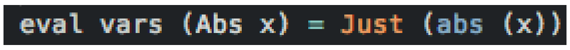
```
data Expr = Add Expr Expr
          | Val Float
          | Sub Expr Expr
          | Mult Expr Expr
          | Div Expr Expr
          | Abs Float
          | Mod Expr Expr
          | Pow Expr Expr
          | Var Name
  deriving Show
```

A data type *Expr* has been defined using the *data* keyword that lists the various operations supported by the calculator.
The various calculator operations are defined in the *eval function* to give the appropriate result. By using the case control structure, the result of the evaluations could be pattern matched, so

that the appropriate result is returned. For example, as shown above, the Add function pattern matches against two integers, at which point it returns the integer total to the process.

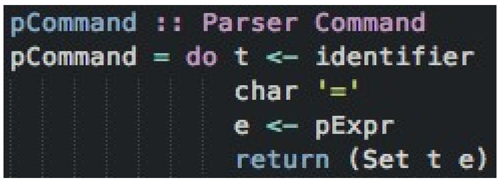

```
eval vars (Add x y) = case (eval vars x, eval vars y) of
  (Just x', Just y') -> Just (x' + y')
  _ -> Nothing
```

Similarly, *Abs*, *Mod* and *Pow* have been implemented.


```
eval vars (Abs x) = Just (abs (x))
```

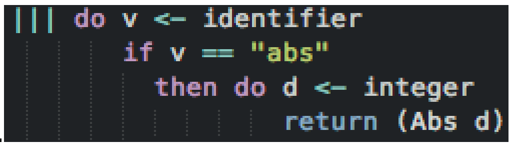
This function returns the absolute value of the integer entered by the user.

The parser evaluates each token separately to identify the type of command the user has entered. For example, if the first token is an *identifier* and the next token is an equal sign, then the *Set* command is invoked which stores the value in the variable.


```
pCommand :: Parser Command
pCommand = do t <- identifier
              char '='
              e <- pExpr
              return (Set t e)
```

Parsing for the *Abs* function was a little different than the rest as the first token of the entered syntax had to be an identifier. If the first token is *"abs"* and the next token is an integer, the *Abs* function is invoked.

In order to prevent the program from throwing exceptions that cause the calculator to stop running, it was necessary to error handle the case where there is a divide by zero. In order to avoid this, a case is introduced, causing Nothing to be thrown as the error, rather than the program crashing.


```
||| do v <- identifier
       if v == "abs"
         then do d <- integer
                 return (Abs d)
```

REPL.hs

This file contains the process function which handles the various commands such as *Set*, *Eval*, *Hist* and *Quit.* These functions are responsible for the actual actions that the calculator takes once it has parsed a command using the pcommand parser. For example, if the pcommand returns an eval command (with the expression to be evaluated), the process takes that command along with the current state, “st”, and updates the state’s history. The next step is a case to pattern match against the result of an evaluation (using eval) of the expression. Eval returns a Maybe Int, which allows us to catch errors. If a Just int is returned, we can simply print the result. If a Nothing is returned, then the program can output an error message.

Along with the process function, there are a number of extra helper functions which allow the calculator to manipulate the data stored within the State data construct. The updateVars function works by simply “consing” the input tuple onto the variable list “vars”. In order to make sure that variables were overwritten if a new definition for a variable was entered by the user, the dropVars function is called. This function uses filter with a lamda function to remove any variables with a matching name. For the sake of convenience, rather than check if the variable is a new one using a conditional, the dropVars function is called every time in the updateVars function.

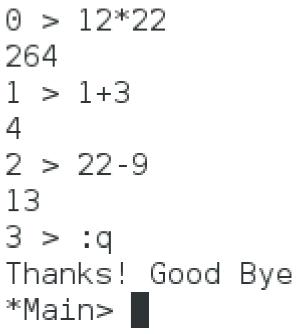
The updateVars function was a necessity when it came to implementation of the variable setting feature. This feature required the implementation of the provided Set command process. The function evaluates the expression, updating the state’s variable list if valid, and printing an error otherwise. During writing of this feature, we learned about the immutability of structures in haskell. In order to update the state’s variable list it was necessary to define a new function addVars which took the state, the updated list and returned a new state with the updated list. The function then recurses, passing the new state (from addVars) to the repl call.

The add history aspect of the REPL.hs involved the updating of the history aspect of the state. Every time a new calculation was entered, it was worked out in Expr.hs, then passed through to REPL, where it was added to the history and displayed for the user. At the same time, it was reasonable to assume that every time a new value was added to the history, a correct calculation had been entered. This means that it was more efficient to update the number of calculations the user had entered at this point. So within the same construct that added to the history, the number of calculations was also incremented.

The retrieve history function was used to retrieve the result to a calculation that had been previously entered. It worked by receiving a number from the user after a ‘!’ token. This number would then be checked to be correct, or not larger than the furthest back value. If it was larger, then the function would go to the furthest back, or earliest, calculation. If all was well, then the function would return the result of the calculation that was the precise number back. The numbers start at 0 for 1 the most recent calculation, and move on as such. Similar to how an array would act, by starting at 0 and moving to be one less than the size.

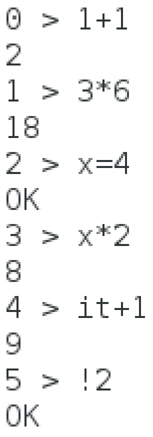
Testing

Test 1: This was to test if the basic requirement of quitting worked.



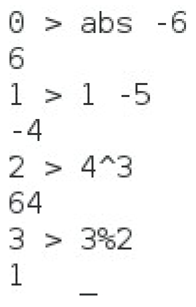
```
0 > 12*22
264
1 > 1+3
4
2 > 22-9
13
3 > :q
Thanks! Good Bye
*Main> █
```

Test 2: This was to test if the basic requirement of being able to access command history worked.



```
0 > 1+1
2
1 > 3*6
18
2 > x=4
OK
3 > x*2
8
4 > it+1
9
5 > !2
OK
```

Test 3: This was to test if the extension of negative numbers and the abs, mod and power functions worked.



```
0 > abs -6
6
1 > 1 -5
-4
2 > 4^3
64
3 > 3%2
1   _
```

Due to the large extent of all possible tests, it would be impossible to test every case. So we included a variety of options, showing off that every aspect works. From positive to negative numbers, along with the ability to support both single and multiple digit numbers. There was a wide variety displayed in the tests, though as previously stipulated, to test every possible solution would be impossible, so we used a smaller number of edge cases as examples but tested as we made our way through the practical.

Problems

It took us a while before we were able to manipulate the state data structure to manipulate the history function. The initial problem was that we did not realise that the history command was a function, then once we had solved that minor inconvenience, we struggled to find the correct locations to call the functions which manipulated the history from. However, there is still a slight problem in that when the function retrieves from history, it also adds to history, increasing the size of history with a duplicate of what was retrieved.
Also we discovered a small problem with how we had implemented our Evals in the process function. This in turn would cause the quit function to fail temporarily due to the recursive aspects of the program. This too was fixed by manipulating the location of the code which solved the problem.

Individual Reports

130010944

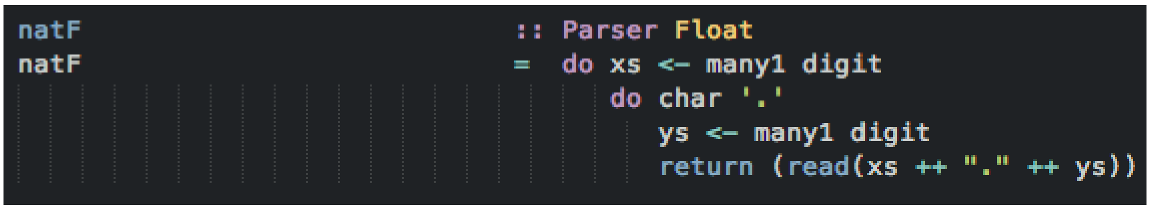
I have made a number of contributions to the program. I was responsible for the implementation of support for add and the other basic operators. I also implemented the Set feature that allows the user to define their own variables, along with the variable storage in the state. This naturally lead to an easy implementation of the implicit “it” variable. It simply required that the update section of the Set process be swapped into the Eval process as well, except that instead of supplying the updateVars function with a custom variable name, the name “it” was passed in.

I assisted 130011035 with the implementation of the recent command functionality through pair programming. Because I was responsible for the processes initially, I had the best understanding of the process function, and I was able to assist by adding the “Hist” command, which utilised the retrieveHistory function which 130011035 wrote.

I also optimised the Quit feature, by switching it into a quit command, rather than a section of the REPL loop. This makes this aspect follow the design of the program more naturally. I was also responsible for the error handling surrounding divide by zero exceptions. However, my error handling produces a generic message, not necessarily related to division by zero. If we had implemented the Either type, it would have been easier to introduce a specific error message, by perhaps returning our own Error type, rather than Nothing.

140014861

I have added support for operations such as abs,mod and power. I had implemented command history on my own with a few bugs. The output wasn't as expected sometimes. 130010944 managed to fix the issue and solve the problem.
I implemented the support of floating point numbers. This was done by mainly changing parsing.hs. The following parsing style also prevents a string with multiple decimal points to be considered as a float.



```
natF                           :: Parser Float
natF                           =  do xs <- many1 digit
                                     do char '.'
                                        ys <- many1 digit
                                        return (read(xs ++ "." ++ ys))
```

A few minor changes Expr.hs and REPL.hs to support Float instead of Int were made.
Adding this functionality broke the support for mod since mod only works for integral values.

<u>130011035</u>

Quit Function: I initially implemented the quit function. Before we discovered a problem with the layout, we had a different way to quit. It involved an if which would check if the input included a ':q' then it would print a message and end, otherwise it would run the loop in the REPL file. By changing it we moved towards a more functional design.

The number of calculations: I worked on the number of calculations aspect, and managed to get it to increment every time a new value was added to the history which should happen every time a new calculation is entered.

The command history: I implemented the ability to add to the history. So every time a new calculation was entered, it would add it to the history part of state. I also implemented the ability to retrieve the calculation that the user requests using the !n aspect. However, I struggled to find the part of the program that I should use to call the functions. So there were aspects of pair programming as we solved the problem, and in the end 130010944 managed to solve the problem in it's entirety.