CS2006 H1 – Group Report 180012514 180016546 190021081 19/02/20

Overview

We have been asked to create a calculator in Haskell that supports a wide range of functions expected from an actual calculator; such as variable support, the handling of errors and an active history. All basic requirements have been implemented, meaning that our calculator can handle basic addition, subtraction, multiplication and division as well as having it, quit and set variable functions and an active state that tracks the command history and current values of variables. As for the additional requirements, all of the easy and medium ones have been implemented. This means that our calculator can deal with negative floats and integers, supports additional functions such as power and absolute, makes use of a binary search tree for storing the variables and can read files for inputs. Our calculator also has much better error treatment than needed in the basic requirements. Our use of haskeline means we have also met one hard requirement as the input has been improved.

Design & Implementation

Basic Requirements

The additional operators each needed a corresponding instance of the Expr data type, which were named to be as readable as possible. They then needed methods for evaluation – before implementing the use of the Either type the evaluations used the Maybe type and methods defined in the MaybeHandling module, but since implementing the Either type they use equivalent functions in the EitherHandling module. REPL's process st (Eval e) then needed to call Expr's eval function to evaluate expressions, passing in the at the time empty variables in st. The quit command is simply an entry in Expr's Command data type, that process outputs a string for and then, importantly, doesn't call repl, cutting the loop short and terminating the program.

State was simple enough to implement, though we did find that in order to update both the command history and the collection of variables we needed to create two temporary States (st' and st'') due to updateVars returning a collection of variables and addHistory returning a State: we couldn't make a single state out of an updated command history and an updated variable collection because we didn't have those two components. Being able to update and access the collection of variables was a prerequisite to being able to assign variables. Variable recognition was easy to implement, it just required a lookup function that called be called by eval when evaluating an instance of the new Var Name definition of Expr and for pFactor to return such an instance.

Accessing command history was as simple as a new Command definition (Repeat Int) that process would resolve by calling itself with the desired command from the history list of st. Multiple digit integer recognition was as simple as using Parsing's int primitive. To allow whitespace the user's input is just filtered to remove the ' ' character in the repl function. Similarly, in-line comments are implemented by only trying to parse user input up to a '#' character – the octothorpe and any characters after it are dropped from the input before parsing. The implicit it variable is achieved by calling updateVars with the name "it" for the result of each Eval e command.

Additional Requirements

Easy Extend the parser to support negative integers

The parser already supported negative integers as a result of using Parsing's int primitive (as opposed to the nat primitive, which would only have allowed positive values).

Easy/Medium Support floating point numbers and integers

This was done by simply adding in two new tokens: a float and negative float. References to the 'int' tokens were then replaced in the other files with the new float tokens. An if statement had to then be implemented to remove the decimal point and decimal place from whole numbers to produce integers when only integers are involved. Once the parser could recognise floating point numbers, the next issue was adapting the arithmetic evaluations to work with them. In the case of the div, mod and power functionalities this required either throwing and error message or converting the types – Haskell doesn't support these operations on floats instead requiring values of the RealFrac class. To get around this we use our eitherGetIntegral function to obtain truncated, Integral class values. Truncation necessarily involves the loss of some data, however it seemed like the most elegant solution to the problem as it allows the user to obtain at least an approximate value for all calculations.

Medium Use a binary search tree for storing variables instead of a list of pairs

The BinarySearchTree module has a relatively simple implementation of a binary search tree, defined with a recursive data type (BiTree) that is either a Leaf, or a Node with a left sub-tree, a right sub-tree and a tuple value. The tuple was initially less generic, consisting of a String and an Int, but it was made more generic for reasons that will be explained later. The tree doesn't balance itself, as Dr Brady said that the purpose of the extension wasn't to make the variable lookup faster but to show we could implement a tree structure in Haskell. There is no remove function because there was no reason to remove old variables – repeated definitions overwrite old values without trying to remove them so the dropVar function from the source code has been removed.. As such the tree only supports being inserted into (in a way that preserves ordering but not balance) and to retrieve the value of a variable by recursively descending the tree until either the desired value or a leaf is reached.

The difficulty in implementing the binary search tree came from making it work with QuickCheck – it was necessary to make an arbitrary generator for the BiTree data type. This tutorial [https://www.stackbuilders.com/news/a-quickcheck-tutorial-generators, accessed 19/02/2020] by Juan Pedro Villa from stackbuilders.com was useful but left a lot of room for experimentation to apply its logic to a non-list based data structure. However, I think I was eventually successful in setting up a test (quickCheck prop_ordered from within ghci with BTTesting.hs loaded) that shows that the insert function does maintain order according the first element of the tuple value. The increased genericness of the tuple was due to the difficulty I had in generating a random string without importing a third-party module for that purpose, so that type restriction was removed allowing me to simply use the arbitrary monad from the Test.QuickCheck module.

Medium Implement a command to read input files containing a list of commands, rather than reading from the console. To implement this Haskeline was imported and used. In order to get this to work within our implementation at this point, some of the function decelerations needed to change from *IO ()* to *InputT IO ()*. Furthermore, all *print* and *putStrLn* function calls were changed to the Haskeline *outputStrLn* function. The main method now calls the Haskeline function

runInputT with the default settings, and a *runFile* function was added in the *REPL.hs* class that when called finds and opens the specified file. The *repl* function that was declared in the starting code was modified to adapt to reading in a file – this was as simple as adding a *Just* and *Nothing* case to adapt for the end of the file being reached. A new branch of the process function was also added to meet the new command declaration. *System.Directory* was imported because there was a function that could be called to check to see if the file exists, this was also helpful for the better treatment of errors within the overall program. The file is read in this process. The only problems that could not be fixed were with saving the commands and expressions to the global history so that they could be continued to be used with the normal console input. The old implementation would go back in the history and continue like only one valid command was used – our temporary fix for this was to end the program after the end of file is reached to stop this was occurring – all other implementation for this extension is met.

Medium Implement better treatment of errors: instead of using Maybe and using Nothing to indicate errors, look at the Either type and consider how to use this to represent errors.

The Either type allows for methods to generate error messages as Left values that can be passed up until they reach the IO functions in REPL. Retrieving from Eithers seems quite inelegant – because the left and right values can be different types, it is necessary to use fromLeft and fromRight outside of a do block, supplying a default value in case the call is made on the incorrect type. Despite this, they do work as intended however we struggled to think of error cases that weren't parse errors – the calculator will gracefully (i.e. without terminating the program) display an error message if undeclared variables are referenced but this is the only example of such an error.

Hard We implemented Haskeline in our implementation, and this improved input by allowing backspace and arrow keys to work in the console. This is an improvement from the original implementation because these things would cause weird input and would require us to press enter and do the whole input again.

Other

Although it doesn't appear in the specification, we did include a helper command (":h") that displays all supported operations and commands within the program.

Testing

To test the problem, the majority of testing was done via input files. While implementation of the file handling a file called *testfile* was created, and when testing two more were made. The *basic* file has many commands and operations that include addition, subtraction, multiplication, division, the it command and the repeat (!) command – using integers as well as floats. The *advanced* file contains operations and commands that were added when completing extension requirements (absolute, power, modulus and a help command). Both normal, extreme and exceptional data was tested in these files.

In both the *basic* and *advanced* files, the expected result is expressed as a comment – this was done to make the comparison between expected and actual outputs easier.

Basic:

CS2006 H1 – Group Report 180012514 180016546 190021081 19/02/20

This file contains 43 commands, some of which result in a Parse Error so the actual result in console becomes 37 commands. The console output (left) and expected output (right) is given below:

```
*Main> main
0 > :f tests/basic
Reading file
0 > 2
1 > 0
2 > -5
3 > 10
4 > 11.1
5 > 5
6 > -10
7 > 10
8 > -90.6
9 > 15.5
10 > 0
11 > -10
12 > Invalid history fetch
12 > -10
13 > 6
14 > 705.12006
15 > -2952.5999
16 > 5
17 > 4
18 > Infinity
19 > 0
20 > 0.102
21 > 0.102
22 > 9.898
23 > 0
24 > 10
25 > -4
26 > 15.329897
27 > 249.80168
28 > OK
29 > OK
30 > OK
31 > 11.4
32 > OK
33 > 1.0961539
34 > -179.76923
35 > OK
36 > -179.76923
37 > Parse Error
37 > Parse Error
37 > Parse Error
37 > Parse Error
37 > Bye
*Main>
```

```
1  1 + 1                        # = 2
2  0 + 0                        # = 0
3  -10 + 5                      # = -5
4  5.5 + 4.5                    # = 10
5  10 + 1.1                     # = 11.1
6  10 - 5                       # = 5
7  0 - 10                       # = -10
8  10.4 - 0.4                   # = 10
9  -80.4 - 10.2                 # = -90.6
10 10 - -5.5                    # = 15.5
11 10 * 0                       # = 0
12 5 * -2                       # = -10
13 !12                          # = Invalid history fetch
14 !11                          # = -10
15 1.5 * 4                      # = 6
16 -45.2 * -15.6                # = 705.12
17 14 * -210.9                  # = -2952.6
18 50 / 10                      # = 5
19 8.8 / 2.2                    # = 4
20 10 / 0                       # = Infinity
21 0 / 10                       # = 0
22 -10.2 / -100.0               # = 0.102
23 !20                          # = 0.102
24 10 - it                      # = 9.898
25 it - it                      # = 0
26 10 + 10 - 5 * 2              # = 10
27 1 - 2 + 3                    # = 2 (e-mail from Edwin conf
28 10 / 2 + 100.2 / 9.7         # = 15.3298969072
29 10 - it * -15.642746         # = 249.801683526
30 x = 1                        # = OK
31 y = 10.4                     # = OK
32 z = x + y                    # = OK
33 z                            # = 11.4
34 a = z / y                    # = OK
35 a                            # = 1.09615384615
36 164 * -a                     # = -179.769230769
37 b = it                       # = OK
38 b                            # = -179.769230769
39 +                            # = Parse Error
40 - *                          # = Parse Error
41 \ 3                          # = Parse Error
42 b = !20                      # = Parse Error (Not required
43 :q                           # = Bye
```

For comparisons sake, a table was made to comment on these results.

Command	**Actual Output**	**Correct (Matched against expected)**	**Comments**
1 + 1	2	Match	
0 + 0	0	Match	
-10 + 5	-5	Match	
5.5 + 4.5	10	Match	
10 + 1.1	11.1	Match	
10 - 5	5	Match	
0 - 10	-10	Match	
10.4 – 0.4	10	Match	
-80.4 – 10.2	-90.6	Match	
10 - -5.5	15.5	Match	
10 * 0	0	Match	

5 * -2	-10	Match	
!12	Invalid history fetch	Match	
!11	-10	Match	
1.5 * 4	6	Match	
-45.2 * -15.6	705.12006	705.12	Rounding error
14 * -210.9	-2952.5999	-2952.6	Rounding error
50 / 10	5	Match	
8.8 / 2.2	4	Match	
10 / 0	Infinity	Match	
0 / 10	0	Match	
-10/2 / 100.0	0.102	Match	
!20	0.102	Match	
10 - it	9.898	Match	
it - it	0	Match	
10 + 10 – 5 * 2	10	Match	
1 – 2 + 3	-4	2	BODMAS error
10 / 2 + 100.2 / 9.7	15.329897	15.3298969072	Rounding error
10 – it * -15.642746	249.80168	249.801683526	Rounding error
x = 1	OK	Match	
y = 10.4	OK	Match	
z = x + y	OK	Match	
z	11.4	Match	
a = z / y	OK	Match	
a	1.0961539	Match	
164 * -a	-179.76923	Match	
b = it	OK	Match	
b	-179.76923	Match	
+	Parse Error	Match	
- *	Parse Error	Match	
/ 3	Parse Error	Match	
b = !20	Parse Error	Match	**
:q	Bye	Match	

The rounding errors were ignored because they are not important due to it being so very slightly different (eg .5999 rather than .6). Other than that all other results were as expected except from the command 1 – 2 + 3. This was expected to be 2 yet our calculator returned -4, however this was

something that can be explained in one of the emails sent out – and we were not expected to fix this error due to it being a problem with the starter code.

** - The specification is quite ambiguous in whether this should work or not. Technically as Repeat is a command rather than an expression it shouldn't be able to be stored as a variable, but for completeness it probably makes sense to be able to do this – for how difficult this would be to implement we decided against adding functionality for this.

Advanced:
This file contains 13 commands, some of which result in a Parse Error so the actual result in console becomes 9 commands. The console output (left) and expected output (right) is given below:

```
*Main> main
0 > :f tests/advanced
Reading file
0 > 4
1 > 4
2 > 9
3 > Parse Error
3 > 1
4 > 0
5 > 8
6 > 4
7 > Parse Error
7 > Parse Error
7 > 3.5831811
8 > 2
9 > These are the supported operations:
    x + y : addition
    x - y : subtraction
    x * y : multiplication
    x / y : division
    x % y : modulo (x and y will be truncated)
    x \ y : div (x and y will be truncated)
    | x | : abs
    x ^ y : power of (y will be truncated)
    x = y : variable assignment
    it    : holds the result of the last calulation
These are the non-arithmetic commands:
    !n          : repeats the nth command
    :q          : quits the program
    :f filename : reads the file
    :h          : displays the help message
Inline comments are denoted with '#' and continue to the end of the line
9 > End of file
*Main>
```

```
1   |4|                          # = 4
2   |-4|                         # = 4
3   |-4-5|                       # = 9
4   ||                           # = Parse Error
5   5 % 2                        # = 1
6   3.4 % 1.7                    # = 0
7   2 ^ 3                        # = 8
8   (2+2)*3^|-4%1|               # = 4
9   ^ 2                          # = Parse Error
10  % 2.1                        # = Parse Error
11  1.2 ^ 7.3                    # = 3.5831811
12  5 \ 2                        # = 2
13  :h                           # = Displays help commands
```

Command	Actual Output	Correct (Matched against expected)	Comments
|4|	4	Match	
|-4|	4	Match	
|-4-5|	9	Match	

||	Parse Error	Match	
5 % 2	1	Match	
3.4 % 1.7	0	Match	
2 ^ 3	8	Match	
(2+2)*3^|-4%1|	4	Match	
^ 2	Parse Error	Match	
% 2.1	Parse Error	Match	
1.2 ^ 7.3	3.5831811	Match	
5 \ 2	2	Match	
:h	– displays the output	Match	

All of the tests in the advanced file passed as expected. This file shows that the execution of the program ends when the end of the file is reached.

To test that our Binary Search Tree worked, we made a test class for it, named BTTTesting. This file was then ran and it would output the number of tests passed. Once we saw all 100 tests were passed, we knew our binary search tree had been successfully implemented and we could move on.

```
*BTTesting> :l BTTesting.hs
[1 of 2] Compiling BinarySearchTree ( BinarySearchTree.hs, interpreted )
[2 of 2] Compiling BTTesting        ( BTTesting.hs, interpreted )
Ok, two modules loaded.
*BTTesting> quickCheck prop_ordered
+++ OK, passed 100 tests.
*BTTesting>
```

Evaluation

One annoyance with the calculator is the way that ghci deals with floating point numbers. On rare, but repeatable, occasions, the output for certain values will contain multiple decimal places, to the point of inaccuracy. For example, for the calculation (1-0.9), the answer is 0.100000024 rather than the obvious 0.1. These additional decimal places only seem to occur under certain circumstances, like if there's a number greater than zero before the decimal point or what side of the calculation the floating point is involved. The fact that this issue occurred so rarely, it made it harder to implement a function that correctly rounded the values to a smaller number of decimal places and as we were under a time constraint, said function was not able to be implemented. Another source of error that has been left in is the ordering of BODMAS that results in the parser treating "a+b+c" as "a+(b+c)" rather than "(a+b)+c". However, this issue was addressed in an email sent out by Edwin Brady where he said 'it's fine to leave it as it is for the purposes of this project'. With this consent, we moved on to deal with more additional requirements as time was of the essence. Implementing Haskeline was something we found some difficulty in because it required such a massive change to the program with the new function decelerations. Some problems will stand with the file handling with the commands inside the files only existing in the state for the duration of the file being read.

Conclusion

All in all, we are happy with the work that we have submitted. This was our first time working together as a group and it took time for us to adjust to the differences in each of work styles, but by the end of the deadline we were working in a much more streamlined fashion and getting work done at a faster rate. We look forward to the next practical, where we would be hoping to start of at the same rate that we finished this practical at.

References

QuickCheck used to test that the binary search tree has been correctly implemented: https://hackage.haskell.org/package/QuickCheck