Haskell Calculator

Files submitted

Code directory

- README → instructions on how to install and run program
- src directory → the source code
- tests directory → test files we used for testing the program processing instructions from .txt files
- calculator.cabal → cabal information file for the cabal package
- Setup.hs → cabal setup file
- LICENSE - 'No license'

Level of completion

The program fully satisfies the following requirements:

>> Basic requirements (all):

- Addition, subtraction, multiplication, division & extend parsers
- Variable assignment
- Quit command
- History command
- Parsers support multiple digit numbers & whitespace
- Implicit variable 'it'

>> Easy Extensions (all):

- Parser supports negative integers
- Support for functions: abs, mod, power
- Support for floating point numbers

>> Medium Extensions (all):

- Support floating point numbers AND integers & no type-mismatch problem
- Support string operations: addition and multiplication
- Binary search tree for storing & accessing variables
- Command for reading input files (.txt) containing lists of commands
  - Saves result of processing the commands to a _result.txt file
  - Optionally display the result to console
  - Commands processed (from the file) are automatically included in history
- Error handling using *Either*

>> Hard & Very Hard Extensions:

- **Commands for printing and looping**
  - Repeat command
  - Nested repeat command
    - E.g. : `1> :repeat x=x+1 10 times`
      `2> :repeat !1 20 times`
      Will be processed the same as: `:repeat x=x+1 200 times`
  - Also works for commands read from file
- **Right associativity parser problem SOLVED**
  - i.e. initially, a parser command like 2-5+6 would be treated as 2-(5+6) and 10/5*3 would be treated as 10/(5*3)
  - by modifying the parsers, we implemented the correct order of operations according to the mathematical rules of operator precedence
- **Haskeline for improving input**
- **Supporting functions as well as variables**
  - Functions can have multiple arguments
  - Functions can be re-defined during the session

>> Additionally:

- Show !x command, where x is the index of the command in history, will show the command associated with an input line – was also useful for debugging
- Remove <variable_name>, removes variable from the BST
- Program was packaged in a cabal package

Design & implementation

List of commands implemented:

<expression>	Will evaluate an expression	
<var_name> = <expression>	Assignment, will overwrite the value of an existing variable	
:q	Quit command, will exit the calculator	
!x	Repeat command with history index x	
:show !x	Show command stored at history index x	
:process <path>	Process the instructions read from .txt file at <path>	
:repeat !x n times	Repeat command with history index x n times	
:repeat <expression> n times	Repeat processing expression /assignment n times	
:remove <name>	Remove variable name from the list	
:func myfunc(a,b,c..) = expr	Define a function, where **myfunc** = function name; **a,b,c ...** = arguments ; **expr** = the actual expression (does not have to contain any of the arguments)	
:call myfunc(1,7,-9…)	Calling a previously defined function **myfunc**	
<string> + <string>	Will concatenate two strings	Can be used in combination with each other, see screenshots
<string> * 5	Will concatenate 5 <string>	

- We have kept the basic design of the originally provided code, but introduced a **repl_file** function to simulate the **repl** cycle when the commands are being read from an external file.
- Instead of reading the command from console using **getLine**, we feed the command from the list **fileInput** stored within the current state and append the result to **fileOutput** (acts like a buffer), which we write to a _result.txt file when there are no more commands inside **fileInput** and go back to the original **repl** cycle.
- In order to implement the **repeat** command I have added a **to_repeat** field to **State** which keeps track of the command being repeated.
- Thus, we now have multiple lines of execution, depending on whether a command is repeated, whether it is read from a file or from the console. The cycle can be roughly described in the following steps:
    - Start **repl** from **Main**
    - If haven't finished repeating a command then repeat it
    - Otherwise:
        - If there is still something to be processed inside fileInput -> go to **repl_file**
        - Otherwise continue inside **repl**

Our code in the source directory is structured in the following way:

Main.hs –triggers the main loop inside **REPL.hs**, preserves the original code, only added the haskeline library.

Data.hs – here, we included the main types and data type classes: Value and Name and the operations that can be performed on and between them: addition, for instance can be done between all 3. We chose to store numbers in a generic type Value, which allows us to read very large integers without losing precision (which would happen if we read all the numbers a floats)

Parsing.hs – we included all the original parsers provided and marked the ones added by us in the comments, describing their functionality as well.

Expr.hs – contains the definition of the data type describing expressions and all the parsers used to construct them.

BSTVariables.hs – contains the tree data structure, which we used to store and retrieve variables and functions from **state**, as well as the operations involved: insertNode, removeNode, getNode. We decided to use two types of nodes. We have initially implemented the necessary functionality for the **value** type of node and then decided to extend the module, to allow us to store functions as well. All the operations work on both kind of nodes.

Command.hs – contains the description of all commands and the parsers used to construct them. We have gradually re-structured this module when solving the right-hand associativity problem until we arrived at the most readable version.

Eval.hs – contains the fundamental function in our program, which evaluates all expressions. We have defined the arithmetic and string operations inside Data.hs , leaving eval to deal with the simple task of recursively applying the relevant operators to expressions and outputting either a String or a Value for each. Here, the use of the **Either** construct offers us flexibility, allowing us to provide more detailed and relevant error messages to the user.

REPL.hs – contains the main 'cycle' of the program: read instruction (from console or file) , process inside **process** function, take appropriate action (display result & update state), go back to reading.

Challenging parts

- Implementing **:repeat** !x, where !x also represents a repeat command. (Perhaps due to the state of the code at that time) it was (surprisingly) tricky to discover why it did not initially work and to come up with the mechanism for multiplying the number of times to repeat within each of the nested commands.

- Switching between **repl** and **repl_file** cycles and checking for the state of to_repeat within the **process** function caused a lot of bugs, until we re-factored the code, introducing the **outputMsg** function that does the check for where to output the result: to console or **fileOutput** buffer. We also reduced the places where the value of **numcalcs** can be changed (currently only inside **repl** and **repl_file**) removing some bugs.

- Solving the right-associativity problem was the most challenging (and most rewarding) part. It took a while to come up with the idea of a 'subtractable' expression and switching between pAddition and pSubtraction parsers. After that, the problem was basically solved, because the same can be applied to higher precedence operations like Mult,Div,Abs etc.
  - In solving this I started by re-writing pExpr to parse expressions like 1-5+6 correctly and then gradually re-written and re-factored the remaining parsers, until we could finally group them into pHigherPriority and pLowPriority.

- Implementing functions was also an interesting task.
  - We thought about it like this:
  - User defines: function_name (agrn) = expr
  - User calls: function_name (argv)
  - The key to implement this is to evaluate the expr in the same way we would evaluate a normal expression in our program, except that when evaluating a function we combine argn and argv into a new tree **vars'** of variables and pass this tree in the function call : **eval vars' expr**, which will use the normal look-up mechanism for variables and replace them with their values.

Summary of provenance

As a group, we started working by making a list of all the requirements, and tried to divide the work into 3 parts.

Personal contribution:

- Assuming all numbers are read as floats, I have implemented the basic and further operations, written the implementation for **eval** function
- Implementation of **process** function for all basic commands **except** *Quit* and *Hist i*.
- Error handling using Either (rather than Mabye) inside **eval** and **process** functions
- Implementation of parsing and processing additional commands like: **remove variable**, **show** command given history index, **repeat** command (+ supports nested repeats), **process file** (with storing and displaying the results) & extended State appropriately
- Error handling of the commands listed above
- Implicit variable **it** & preventing user from removing it
- Implemented the parsing of variables (with support for displaying - <var name>)
- Modified and re-arranged all the parsers (except those related to reading numbers and string operations) in order to solve the right-associativity problem & to re-factor the code
- Implemented the **Node** data structure and the operations **insertNode, removeNode, getNodeValue** and used them in **updateVars, dropVar** and **eval** functions respectively.
- Written the parser for reading function declarations, using the added parsers **commaseparated** and **arguments** inside Parsing.hs

(Personal) modifications to the original code: highlighted in green in the list above.

Testing

We created the test suite together as a group and then I tested the program with instructions read from the files in the tests directory, while another member tested the program with commands read from the console, as well as the :func and :call commands. The tests are presented in the tables below and in the screenshots of the sample session in our program. We share the same testing evidence within the group.

Commands typed at the console

Test case number	Type of input	Input	Expected output	Actual output (all passed)
1	normal	5+2	7	
2	normal	x = 20	OK	
3	normal	y = x + 3.0	OK	
4	normal	x	20	
5	normal	y	23	
6	normal	20 / 4.2	4.761905	
7	normal	5 ^ 2.3	40.51641	
8	normal	“ok” * 4+15	“okokokok15”	
9	normal	:repeat :repeat :repeat x=x+1 5 times 10 times 2 times	OK	
10	normal	x	120	
11	normal	:remove x	OK	
12	normal	x	'x not found.'	
13	normal	:func double(x)=x*2	OK	
14	normal	:call double(2.4)	4.8	
15	normal	5/2	2	
16	normal	it	2	
17	exceptional	:process abcde	Failed to read “abcde”	
18	exceptional	5/0	Can't divide by 0.	
19	exceptional	2^-1.0	Invalid exponent.	

Commands read from file

Test case	File name	Description / comment	Expected Output	Actual output
0	test0_emptylines	Some lines are empty & some contain just spaces	Nothing to process	As expected
1	test1_spaces	Contains 3 valid commands with random spaces within them	Neglects spaces, stores the correct results of the operations inside ..\\test1_spaces_result.txt	As expected
2	test2_normal	Contains normal input	Writes the expected (see below) result into ..\\test2_normal_result.txt	As expected
3	test3_exceptional	Contains exceptional input	Writes the expected (see below) result into ..\\test3_exceptional_result.txt	As expected

Case 0, the results of processing ..\\test0_emptylines.txt :

```
? > :process d:\tests\test0_emptylines.txt

        Reading file ... "d:\\tests\\test0_emptylines.txt"

        Nothing to process inside:
        "d:\\tests\\test0_emptylines.txt"
```

Case 1, the contents of ..\\test1_spaces_result.txt after processing:

```
        CALC:   Results of processing file:
        "d:\\hask1_mine\\src15\\test1_spaces.txt"
27>     USER:   1+6
27>     CALC:   7
28>     USER:   258+5
28>     CALC:   263
29>     USER:   c=78
29>     CALC:   OK
```

Case 2, contents of ..\\test2_normal_result.txt after processing:

INPUT	COMMENT / EXPECTED RESULT	TEST RESULT, green = OK, red = FAIL
1+2+5	Support basic addition of integers	
10+18.5 – 15.5	Float and integer mixed operation	
1-5+6	Testing operator precedence: (1-5)+6 and not 1-(5+6) as the original parser did	
1-5-6	Testing operator precedence, as above	
10*11*20	Testing multiplication of integers	
12.5*4	Multiplication of float and integer	
10/2.5	Division of float and integer	
999999999999… (very long number)	Displays number correctly	
123456789123	Checking that parser reads both integers and floats Value displayed should be the same as the original and not in float format	
|100-650|	Abs	
30%24%5	Mod	
16^2	Power	
x=2	Assignment, single value	
x	Displaying value of variable	
z=x^(100/100*5)+|200-700|/5^3%750-32	Assignment, expression	
z	Correct result of expression above	
x=0	Initialization	
:repeat x=x+1 10 times	Should work as if command was input from console	
it	‘it’ should hold the result of last operation, i.e. 10	
x	Checking that x was incremented 10 times, previous operation	
:remove x	Should remove from the BST	
x	Should display: Variable not found	
y=-18	See below:	
-y	‘-‘ operator applied to variables	
123 +”abc"	“123abc”	
"abc"*0	Empty string	
"abc"*5	“abcabcabcabcabc”	
"hey"*3+"hurray"	“heyheyheyhurray”	

Case 3, contents of ..\\test3_exceptional_result.txt after processing:

INPUT	Comment / expected result	Test result, Green= ok Red = fail
illegal*symbols=18	Command not found (CNF)	
CapitalLetter=18	CNF, variables must start with lower case	
x=unknown + 78	Variable not found	
:repeat x=???	CNF	
:repeat (:q) 10 times	CNF, first argument must be an expression or history command, e.g. !8	
x=2	See below:	
!x	CNF, trying to pass a variable as a history index	
:repeat x=x+5 0 times	Cannot repeat command 0 times.	
:repeat x=x+5 -5 times	CNF, invalid number of times	
NO SUCH COMMAND	CNF	
!-10	Invalid history index	
!100000	Invalid history index	
:show !-10	Invalid history index	
:process d:\hask1\src12\test.txt	Cannot process two files simultaneously, valid path	
:process d:\%%%^^nosuchpath	Cannot process two files simultaneously, invalid path	
:remove unknown	Tries to remove it but doesn't crash if variable not found	
:q	No effect. Must type command at console to quit.	
x=100/0	Division by 0, exception caught	
10^(-2)	Invalid exponent	
10^-2	Invalid exponent	

Case 4, contents of ..\\test4_functions_result.txt after processing:

INPUT	Comment / expected result	Test result, Green= ok Red = fail
DEFINING FUNCTIONS		
:func addOne(x)=x+1	Normal input	
:func multSeven(x)=7*x	Normal input	
:func manyargs(a,b,c,d)=(100+a) - 50*b + c - d	Normal input	
:func sameArgs(a,a,b,b)=a+b	Should use the value of first a and first b when calculating result	
:func norags()=15	Allow defining	
:func argsNotUsed(x,y)=26*0+5	Allow defining	
a=10, b=15 :func twoArgs(a,b)=a+b	Should use value of arguments a,b and not of global variables a,b	
:func dividebyzero(x)=x/0	Should display error message	
:func invalid???name(x)=18	Should not allow defining	
:func wrong declaration = wrong	Should not allow defining	
:func 1+6-15^8	Should not allow defining	
:func	No such command	

CALLING FUNCTIONS		
INPUT	Comment / expected result	Test result, Green= ok Red = fail
:call addOne(7)	8	
:call addOne(-7)	-6	
:call multSeven(7)	49	
:call multSeven(-2.5)	-17.5	
:call manyargs(0,2,100,50)	50	
:call manyargs(1,2)	Wrong number of parameters	
:call sameArgs(1,2,3,4)	4 (assigns a=1, b=3)	
:call noargs()	15	
:call noargs(18)	Wrong number of parameters	
:call argsNotUsed(100,200)	5	
:call twoArgs(100,200)	300 (if function had used the global variables, would have been 25)	
:call twoArgs(a,b)	Does not support variables as arguments	
:call dividebyzero(7)	Cannot divide by zero – error message	
:call nosuchfunction()	Function not found	
:call invalid command	No such command	
:call	No such command	

Screenshots of a sample session in our program: