CS2006 Practical 2 Report
Tutor: Ognjen Arandelovic
200001408, 20010973

Repository link: https://lr225.hg.cs.st-andrews.ac.uk/

Overview:

In this practical we were asked to develop a program that would read input instructions (both from terminal and files), interpret them, and execute them. The program would be able to assign and store variables, and deal with simple calculations and conditionals. We worked on this only as a group of 2, as our third group member is no longer studying at this university.

Note: we have included files to test the ‘read’ function on, and a README detailing how to run the program and quickchecks on cabal.

Design:

The program uses a read, evaluate, print loop to execute commands. There is a list of commands that acts as a backlog to be executed, which is added to when performing loops and reading files. The first conditional of the repl function checks if there are any commands in the back log, and if so executes them by passing the 1st command to process. If there are no commands in the backlog (length == 0), then user input from the terminal is taken and parsed, before being passed into process as long as there is not a parsing error.

The vast majority of parsing occurs in Expr.hs, where the user’s input is passed into the pCommand parser which deals with parsing commands such as read, repeat, print, assignations etc. Most of these make use of pExpr, which is a parser that deals with expressions (+,-,% etc.). pTerm deals with the parsing of multiplication and division, while pFactor deals with the parsing of doubles, strings, ints as well as other operations.

Notable design choices made when implementing the parser include multiplying numbers by -1 when they are preceded by a minus sign, and treating double value as two lots of digits separated by a decimal point, which are then concatenated to form a double. We came to the conclusion that despite reading in doubles and ints separately we should treat them both as

doubles (by converting ints to doubles) as this would save a lot of code when performing operations. This is due to the pattern matching that occurs in the eval function, which would require a lot of cases for every single expression (e.g. Int and Double, Int and Int, Double and Int etc).

The eval function evaluates calculations by pattern matching to the correct data type ‘Expr’ (which we expanded to hold all required expressions). These make use of ‘Either’ which returns Right (Value) when an expression is successfully evaluated and Left (“specific error type”) otherwise. Most of eval is self explanatory, such as adding the two inputted values in the Add function. They all make use of cases to ensure that Values are valid, by recursively calling the eval function. ToInt uses read to transform the string to an int, however then reformats it to a double for the reasons described above, while ToString calls show to convert an integer to a string then returns this as a StrVal. We decided that calling toString on a string would just return a string rather than throwing an error, this was done by adding a case in eval where a string is passed into it.

In order to deal with doubles and strings we created a new data type as suggested and changed eval so that all mathematical operations acted on doubles instead of int. This required the toInt method to actually convert the input strings into doubles instead of integers, so we decided to truncate all doubles that could be to integers when they were output and exclusively work on doubles inside the program. We also made the required modification to maybe, switching the program to use either instead which allowed us to provide more specific error messaging depending on where the problem occurred.

We decided that it would be useful to make a change to the parsing file to include new ‘derived primitives’: isnotquote and isnotspace. This served the purpose of allowing us to easily determine where a string began and ended without having to rewrite the checking code each time.

Our implementation allows for the construction of conditionals. To implement this we created a new parser method called pBoolean which took in two expressions and a comparator so that we could evaluate it. We would then take in two commands of which one would be performed depending on how the boolean was evaluated. We also created a looping conditional which allowed for an instruction to be evaluated a specified number of times.

The process function in REPL has multiple implementations depending on the Command passed into it. Set takes a variable and an expression and if valid, uses the updateVars function to add the variable-value pair to the list. updateVars checks if a variable is already stored in LState using elem, if so it makes use of map to find that variable and update it with the new value, otherwise it just appends the variable-value pair to the list.

Print checks the type of expression to be printed, if it is a string it can merely call putStrLn to print this, however if it is a int it needs to convert it to a string before doing so. For doubles, we first check whether it is actually a double, which it is then converted to a string and printed, or if it is an integer (x.0) then it is converted to an integer so that it is printed without the trailing zeros. Input gets the user's next line of input before calling process on it and then re-calling repl to continue the program.

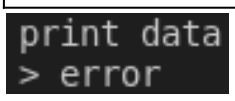
Read reads the file with the name passed in using readFile, then splits the commands into a list of commands by calling lines on it. If the file didn't contain any commands then an error message is printed before continuing the program, otherwise the list of commands is added to the backlog which will then be iterated through.

If takes a boolean expression, and two commands for the 'then' or the 'else' sections, the boolean operator in string form is then matched to the haskell equivalent and the entire expression is evaluated. If true, then the 'then' command is passed into process, otherwise the 'else' command is.

Repeat works similarly to read in that it adds the command to the backlog, this is added n times where n is the number specified by the user. There is an error message printed if n is less than 0, and a base case was included due to the recursive nature of the function.

Manual Testing:

Test	Pre-requisites	Expected output	Actual output
Print rejects invalid expressions	Program running	Error message output	Success see below



Test	Pre-requisites	Expected output	Actual output
Print outputs specified expression	Program running	Provided expression output	Success see below

```
print 1
> 1
```

Test	Pre-requisites	Expected output	Actual output
Print evaluates expression before printing	Program running	Expression output has been correctly evaluated	Success see below

```
print 2 + 2
> 4
```

Test	Pre-requisites	Expected output	Actual output
Print works with strings	Program running	Provided string is output correctly	Success see below

```
print "Hello"
> Hello
```

Test	Pre-requisites	Expected output	Actual output
‘=’ successfully stores the expression provided after into the variable specified before	Program running	Provided expression output	Success see below

```
x = 1
print x
> > 1
```

Test	Pre-requisites	Expected output	Actual output
‘=’ can store strings	Program running	Provided string output	Success see below

```
x = "Test"
print x
> > Test
```

Test	Pre-requisites	Expected output	Actual output

'+' works with both integers and doubles	Program running	Expression evaluated correctly for both data types	Success see below

```
print 1 + 1.1
> 2.1
```

Test	Pre-requisites	Expected output	Actual output
'+' rejects strings	Program running	Error message output	Success see below

```
print "2" + "2"
> addition error
```

Test	Pre-requisites	Expected output	Actual output
'-' works with both integers and doubles	Program running	Expression evaluated correctly for both data types	Success see below

```
print 1-1.5
> -0.5
```

Test	Pre-requisites	Expected output	Actual output
'-' rejects strings	Program running	Error message output	Success see below

```
print "2" - "2"
> subtraction error
```

Test	Pre-requisites	Expected output	Actual output
'*' works with both integers and doubles	Program running	Expression evaluated correctly for both data types	Success see below

```
print 1.6 * 2
> 3.2
```

Test	Pre-requisites	Expected output	Actual output
'*' rejects strings	Program running	Error message output	Success see below

```
print "2" * "2"
> multiplication error
```

Test	Pre-requisites	Expected output	Actual output
‘/’ works with both integers and doubles	Program running	Expression evaluated correctly for both data types	Success see below

```
print 1.2 / 2
> 0.6
```

Test	Pre-requisites	Expected output	Actual output
‘/’ rejects strings	Program running	Error message output	Success see below

```
print "2" / "2"
> division error
```

Test	Pre-requisites	Expected output	Actual output
‘%’ works with both integers and doubles	Program running	Expression evaluated correctly for both data types	Success see below

```
print 1.3 % 2
> 1.3
```

Test	Pre-requisites	Expected output	Actual output
‘%’ rejects strings	Program running	Error message output	Success see below

```
print "2" % "2"
> modulo operation error
```

Test	Pre-requisites	Expected output	Actual output
‘^’ works with both integers and doubles	Program running	Expression evaluated correctly for both data types	Success see below

```
print 1.2 ^ 2
> 1.44
```

Test	Pre-requisites	Expected output	Actual output
‘^’ rejects strings	Program running	Error message output	Success see below

```
print "2" ^ "2"
> exponentiation error
```

Test	Pre-requisites	Expected output	Actual output
‘++’ works with strings	Program running	Expression evaluated correctly	Success see below

```
print "Te" ++ "st"
> Test
```

Test	Pre-requisites	Expected output	Actual output
‘++’ rejects non-strings	Program running	Error message output	Success see below

```
print 2 ++ 2
> concatenation error
```

Test	Pre-requisites	Expected output	Actual output
‘toString’ converts non-strings to strings	Program running	Value converted correctly	Success see below

```
print "v" ++ toString(1)
> v1.0
```

Test	Pre-requisites	Expected output	Actual output
‘toString’ ignores strings	Program running	String re-output	Success see below

```
print toString("test")
> "test"
```

Test	Pre-requisites	Expected output	Actual output
‘toInt’ converts strings to integers	Program running	String converted correctly	Success see below

```
print toInt("2")
> 2
```

Test	Pre-requisites	Expected output	Actual output
‘toInt’ rejects non-strings	Program running	Error message output	Success see below

```
print toInt(1)
> integer conversion error
```

```
print toInt(1.1)
> integer conversion error
```

Test	Pre-requisites	Expected output	Actual output
Input stores user input in variable	Program running	Option to give input provided	Success see below

```
x = input
1
```

Test	Pre-requisites	Expected output	Actual output
Value stored by input is correct	Program running	Value output correctly	Success see below

```
x = input
1
print x
> > 1
```

Test	Pre-requisites	Expected output	Actual output
Input stores user input as string (so does not evaluate expression)	Program running	Expression output as input	Success see below

```
x = input
2 + 2
print x
> > 2 + 2
```

Test	Pre-requisites	Expected output	Actual output
Read rejects empty file	Program running, file exists	Error message output	Success see below

```
read test.txt
> Error: Empty File
```

Test	Pre-requisites	Expected output	Actual output
Read takes the name of a file and performs instruction contained within	Program running, file exists	Instruction executed correctly	Success see below

```
test.txt
  1   print "Hello World"
```

```
read test.txt
> Hello World
```

Test	Pre-requisites	Expected output	Actual output

Read performs all instructions contained in a file	Program running, file exists	Instructions executed correctly	Success see below

test.txt

```
x = 10
y = 2
z = x ^ y
if z > 35 then z = z - 1 else z = z + 1
print z
```

```
read test.txt
> 99
```

Test	Pre-requisites	Expected output	Actual output
If correctly evaluates the booleans provided	Program running	Booleans evaluated correctly	Success see below

```
if 1 == 1 then print "True" else print "False"
> True
```

Test	Pre-requisites	Expected output	Actual output
If correctly evaluates booleans containing variables	Program running	Booleans evaluated correctly	Success see below

```
x = 1
if x == 2 then print "True" else print "False"
> > False
```

Test	Pre-requisites	Expected output	Actual output
If correctly evaluates the expressions contained within it	Program running, file exists	Instructions executed correctly	Success see below

```
x = 1
if x == 1 then x = x + 1 else print "Done"
print x
> > > 2
if x == 1 then x = x + 1 else print "Done"
> Done
```

Test	Pre-requisites	Expected output	Actual output
Repeat rejects specified number of operations below 0	Program running	Error message output	Success see below

```
repeat -1 {print 1}
> Error: Number of repititions must be > 0
```

Test	Pre-requisites	Expected output	Actual output
Repeat does not perform operation when number of repetitions is specified as 0	Program running	No change	Success see below

```
repeat 0 {print 1}
```

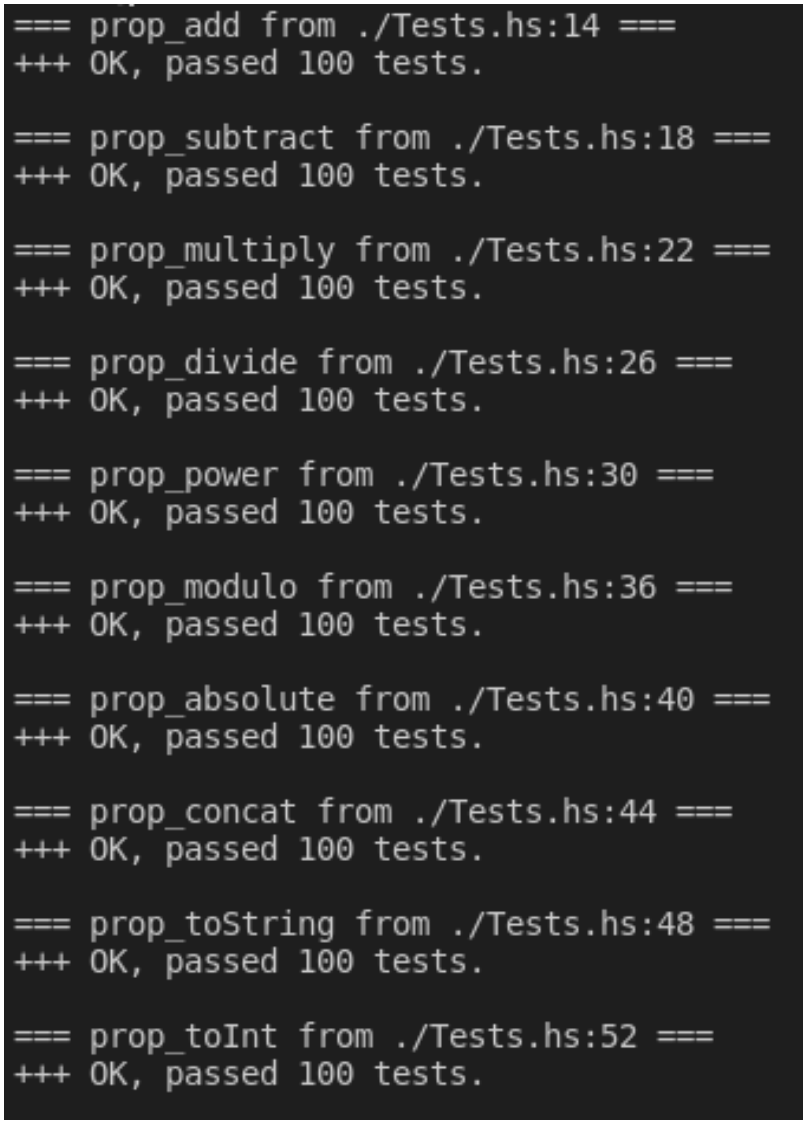
Test	Pre-requisites	Expected output	Actual output
Repeat evaluates expression correctly the number of times specified for it to do so	Program running	Instruction evaluated correctly the correct number of times	Success see below

```
repeat 2 {print 1}
> > 1
1
```

Test	Pre-requisites	Expected output	Actual output
Quit terminates the program	Program running	Exit message and program terminates	Success see below

```
quit
> Exiting...
gw79@pc8-035-l:~/Documents/CS2006/Haskell 2
```

QuickCheck:
Note - In the prop_power test, any tests which are mathematically impossible are skipped as although both our implementation and the haskell ghci calculator return NaN, it is impossible to compare these values



Evaluation:
We completed all basic, easy, and medium requirements except replacing the name value tuple list with another data structure. We also partially completed the hard requirement to create a loop command. However, there were a few bugs in our implementation: when performing calculations the floating point error causes some calculations to return marginally incorrect results; our toString method outputs converted numbers in double form and our repeat command does not allow for multiple instructions (separated by a semi-colon)

to be performed. Given more time we would hopefully be able to fix this last issue, however the marginally incorrect results for double calculations is a problem prevalent in many languages.

Conclusion:
The most difficult part of this practical for us was time management given that we were working in a reduced group size. If we had had more time (or were not missing a group member to share the workload) we would have attempted the remaining requirements, especially switching to a different data structure and completing the repeat function so that it could take in multiple instructions.