Calculator
Haskell project 1

130016030, 130009542, 130013767
University of St Andrews
February 17, 2015

Introduction

The aim of this project is to develop a calculator program, which allows the user to enter arithmetic operations interactively using the keyboard, and immediately display the value of such expressions on the screen. The application should also provide support for variable assignment, error handling and command history. To combine the notion of modeling interactive programs as pure functions with the required side effects, we will use Haskell approach of using a new type along with primitives.

Running instructions

First of all, you need to initialize sandbox by typing in a command prompt:

cabal sandbox init

Then, configure this project:

cabal configure

After that, you need do build it:

cabal build

This will generate an executable file that can be accessed in:

./dist/build/Haskell1/Haskell1

The GUI is not included in the configuration file. However, it can be run by GHCI

ghci GUI.hs

This project uses GUI library called Gtk2Hs.
It should be in the sandbox, BUT if you have any problems with it, you can download it from:
http://projects.haskell.org/gtk2hs/

Design and implementation

1 GUI.hs

The graphical user interface contains main function as well as event handlers. It deals only with basic arithmetic operations. It is using a table layout, so each button knows in what cell it should be. The result of typing and evaluation is displayed in a label. There are 3 event handlers. First one updates the label when user presses number or arithmetic operator. Second clears the label and last displays result of evaluation.

This self-driven extension that was done instead of using haskeline. It could be nice for us to attempt it, but we hope it still meets one of the requirements to "Look at the Haskell libraries and consider how to improve the input".

One difficulty that was is using the latest Gtk2Hs library. For some reasons, all strings has to be casted by using ":: String". This minor thing took a lot of time to fix it. According to GitHub, library developers already know this bug and hopefully they will fix it in next version.

2 Helper.hs

Helper module contains functions that are already not in use. For example, updateListVars or dropVar. Since we replaced lists by binary search tree we do not use them anymore and instead of deleting them, we decided to put it into another file. This module also contains definition of BST and supporting functions for reading file.

3 Expr.hs

This module uses Parsing to recognize commands, expressions and functions. Those functions were extended so it can parse commands and functions. On top of that, it contains evaluation function.

First of all, it tries to parse input and return a result as a command or expression. If it did not match, then it would print an error. After that, if it is an expression, it tries to evaluate it by using pattern matching. Evaluation function uses val function that applies a given operator to the left and right expression. One it is evaluated, it would return a result in Either type.

4 Main.hs

Main contains only one function called main that runs a loop. This is default main configuration in cabal. It was already defined, but we turn standard output buffering off so that the prompt flushes before getting input.

5 Parsing.hs

Parsing module was already written by Graham Hutton, Cambridge University Press, 2007. Some changes were added, namely: implementation of type anything, parsing input for a floating point number.

6 REPL.hs

(Read Evaluate Print Loop): repl, the main loop for the program.

We use a binary search tree for vars and functions as it reduces the time complexity to access the values in said data structures. Linear $O(n)$, where as binary search tree would be $O(logn)$. function repl:

The repl function is defined as taking in a variable st which maintains the state of the program; variables, list of commands to execute, function definitions, etc. In short, repl handles the retrieval of input, processing it and printing an appropriate output to the user. Any variables or functions created during an iteration of the REPL loop get added to the state so that they will be available for the next iteration.

First, the repl function, displays a prompt the user which consists of the number of calculations that they have performed, and an arrow to denote the start of input. Next, a line is read in by calling the function readLine. readLine can either be from the standard input (console) or from a list of commands in the state if said list is not empty. The list of commands to get executed is used by operations like loading a file or calling a function so that their commands get dumped into a list of commands which will get executed before in input is read from the console. We remove a command from said list on every iteration of the repl loop; if applicable.

Once we have a string of input, we need to parse it. This is achieved by calling a function parse which is defined in the given parsing library Parsing.hs (by Graham Hutton, Functional parsing library from chapter 8 of Programming in Haskell). parse takes in a type to parse the input for. First we try to parse for commands by passing in the function pCommand defined in Expr.hs. Whats returned depends on the input. If the input is not a command then the input is tried for expressions and derivatives there of. We then process the returned parsed command or expression. If we get nothing returned, then we simply display a message that the input could not be parsed.

Once the input as been parsed, it calls - from Expr.hs - the appropriate process function to handle it in REPL.hs. The general form of the process function is to apply the actual operation on the data (like calling eval to evaluate the data in Expr.hs), or printing to the screen, etc. It may or may not update the state, in which afterwards, it calls repl to continue the loop with the current state.

7 Lit.hs

We define our own data type Lit which can be either a float, integer or string. In this file, we also define our own functions to handle the result of applying an operator to different operands. In doing so, we allow behaviour such as numbers being concatenated to strings and integer conversions to floats from our overloaded "operators". This allows the language to be easily extended to handle new types and operations.

Testing

```
*Main> main
0 > 4+5
9
1 > x=9
OK
1 > x=x+1
OK
1 > x
10
2 > !1
OK
2 > x
11
3 > 4+8
12
4 > it+4
16
5 > :q
Bye
*Main>
```

Similarly to the example provided in the specification, the basic requirements work exactly as expected. This example demonstrates the quit command, basic arithmetic operations, variable assignment, command history, the implicit variable and also the parser improvements.

```
*Main> main
0 > -5.6 + 12
6.4
1 > 756-234.985
521.015
2 > 14.6 * 2
29.2
3 > 65/2
32.5
4 >
```

After implementing floating point numbers, all arithmetic functions work as expected with both floating point numbers and integers.

```
*Main> main
0 > !1
Index too big! Please provide an appropriate index into the history.
0 > 12+5
17
1 > !0
17
2 > !-1
Index less than zero! Please provide a positive index into the history.
2 >
```

As opposed to throwing an exception and exiting the application, informative error messages are provided when the user tries to access an invalid history index and the program continues running as normal.

```
*Main> main
0 > it
Use of undeclared variable
0 > 23 + (-5)
18
1 > it/6
3
2 > it + it
6
3 > x = 1
OK
3 > it
6
4 >
```

The implicit variable 'it' also does not break the program if used incorrectly. We also demonstrate normal, expected behavior when used in arithmetic operations.

```
*Main> main
0 > x = x + 10
Use of undeclared variable
0 > x = 12
OK
0 > x = x +1
OK
0 > x
13
1 > x + x ^ |-2| mod x
182
2 > x
13
3 >
```

For variables which haven't been defined, it is set as 'Nothing' as opposed to zero meaning we can return an error message if a user tries to use it.

```
*Main> main
0 > (5^3) mod (3^2)
8
1 > |-12| mod |5|
2
2 > (3 mod 16) ^ (12 mod 54)
531441
3 > |-12.54| ^ |-4|
24728.064
4 > | (4.5^(-16)) |
3.5367126e-11
5 > | -1 - 5 mod 2 |
2
6 >
```

Here we demonstrate our implementation of the extra functions, mod, power and abs, which all work well with each other.

```
*Main> main
0 > |15.3| mod 2
Can't mod floats
0 >
```

The only case in which one of the extra functions does not work is mod with any floating point numbers, which is perfectly reasonable behaviour to expect as it is not possible to find the modulus when floating point numbers are involved.

```
*Main> main
0 > abc
Use of undeclared variable
0 > print "abc"
abc
0 > print 54
54
0 > print "+-/*"
+-/*
0 > print "a1b2c3"
a1b2c3
0 >
```

As you can see, the print command works for all types of characters as input, without the use of anything other than the command 'print'.

```
*Main> main
0 > loop 3 -4/12
0 > -4/12
-1
1 > -4/12
-1
2 > -4/12
-1
```

When using the loop command for basic arithmetic operations, it works as expected.

```
*Main> main
0 > x=1
OK
0 > loop 5 x = x+1
0 > x = x+1
OK
0 > x = x+1
OK
0 > x = x+1
OK
0 > x = x+1
OK
0 > x = x+1
OK
0 > x
6
1 >
```

If the user tries to use loop in a more complex way, such as to update a variable already declared and defined, the variable is updates n times. In this example, 1 is added to x 5 times, as opposed to 1 being added to the original x=1 each time it is executed.

```
0 > loop 3 print "hello world"
0 > print "hello world"
hello world
0 > print "hello world"
hello world
0 > print "hello world"
hello world
0 >
```

Looping and printing also work as expected when used together.

```
*Main> main
0 > function main(): dcjknlddsc
0 > main()
0 > dcjknlddsc
Use of undeclared variable
0 > function a():
Could not parse "function a():"
```

When an invalid operation is used as the definition of a function, there is no problem as the function is simply being stored at this point. Then when the function is called, the appropriate error is displayed.
However, if the user tries to define an empty function, this is not permitted by the parser.

```
*Main> main
0 > function main(): x + y ^ |-2| mod 2
0 > main()
0 > x + y ^ |-2| mod 2
Use of undeclared variable
0 > x = 4
OK
0 > y = 9
OK
0 > main()
0 > x + y ^ |-2| mod 2
5
```

Similarly to the test above, if the user tries to use variables which haven't been declared, this will give an error when the function is called. This allows the user to easily call the function after declaring the correct variables without having to redefine the function itself.

```
*Main> main
0 > "hello" + "world"
helloworld
1 > loop 2 :l cal
1 > :l cal
1 > "test!" + 1
test!1
2 > :l cal
2 > "test!" + 1
test!1
3 >
```

After introducing a 'literal' type into the program to handle both floating point numbers and integers, we are also able to handle string operations such as concatenation which behaves as expected with strings and integers.

```
*Main> main
0 > 3 +
Error: could not parse: 3 +
0 > 5 mod
Error: could not parse: 5 mod
0 > ^ 2
Error: could not parse: ^ 2
0 > ||
Error: could not parse: ||
0 > |abc|
Use of undeclared variable
0 > "hello" +
Error: could not parse: "hello" +
0 >
```

If any expression is not completed by the user, it is handled by returning an error message.

The implemented GUI works as expected with simple arithmetic operations using only integers as seen below:

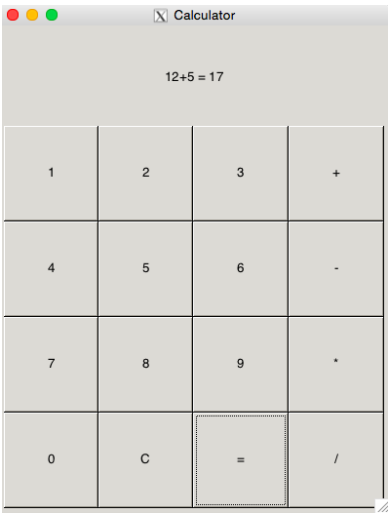

Figure 1: Addition

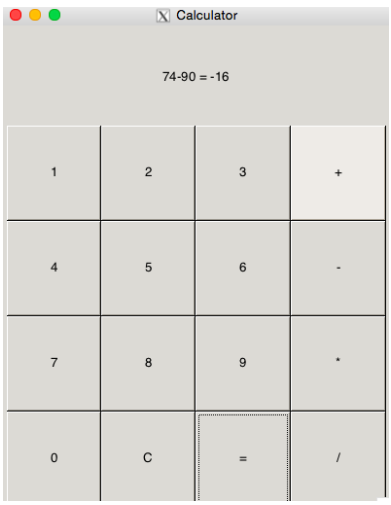

Figure 2: Subtraction

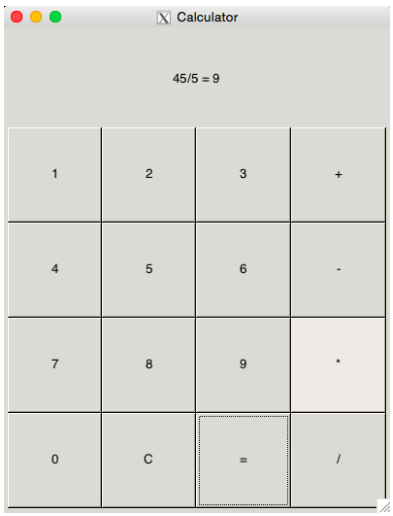

Figure 3: Division

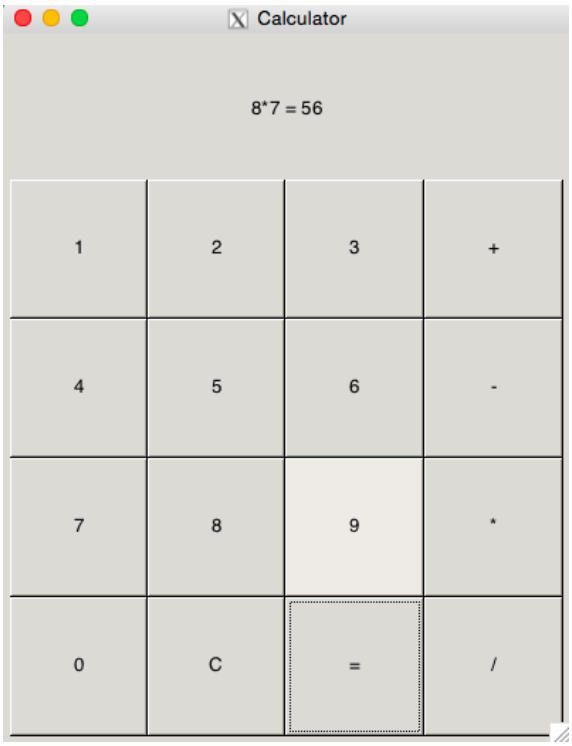

Figure 4: Multiplication

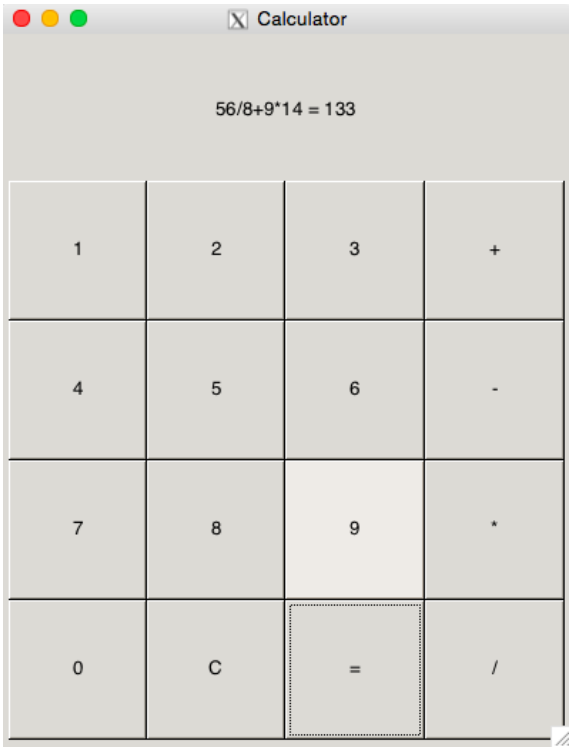

Figure 5: Complex example

Individual reports

130016030

We divided this project equally. My contribution for this project was a managing others, setting tasks and managing our time. Although, I was a team leader, we also spend a lot of time doing pair programming. On top of that, I implemented a binary search tree, functions, print command, loops and some basic requirements. But, nonetheless, the majority of my work was done in pair.

Another major part that I want to raise was to write a graphical user interface. As it was already mentioned above, I had a problem with the library. Although, the direct casting to string is not the most elegant way of writing code, but it is still only one solution that works with Gtk2Hs.

I did not have any difficulties with writing a code or managing other. Sometimes it comes to be misunderstanding of the language. There is a big difference between functional and imperative programming.

Nevertheless, time management was excellent. By doing all work step by step helped us to finished this project on time. As a result, we had a chance to do hard and really hard extension. Only one thing that we have not done is to write a simplify command. I believe this could improve our project.

130009542

I implemented the parser improvements (whitespace and allowing multiple digit numbers) after reading parts of Hutton's "Programming in Haskell" to understand how the parsing library and parser work. This was simply done by replacing the parsing rules being used.

I then attempted the first few extensions. Supporting negative integers was straightforward as it was a similar problem to the parser improvements I made.

Following the example of student 130013767 from when they implemented the basic arithmetic functions, I added the extra functions; mod, power and abs. This involved amending the Expr constructor, creating evaluation functions for the expressions and including them in the parser. I had some trouble with this, also because of a lot of whitespace issues but student 130016030 showed me how to fix this and they also helped me with where each function fit into the parser.

I also implemented support for floating point numbers by writing a new parsing rule in the library called 'FloatInt' which parses an int, followed by a decimal point, followed by a natural number. Two do statements were used to allow either an integer or floating point.

My last task was testing; I checked for any cases where the calculator didn't work so that the team could fix these bugs.

130013767

I started off implementing the basic arithmetic operations by extending the evaluation function in Expr.hs. I did so by creating a function val which takes in an operator (to apply to the operands), a list of variables (which might be used), and two expressions (the operands).

After grasping an understanding of the parser and how it is evaluated, I modified the quit command (previously defined in REPL.hs) so that it is parsed and checked through the parser, which semantically makes more sense. Next I moved on to providing support for loading files, while at the same time providing better error checking as we added more complex operations. The use of Either helped in displaying runtime errors to the user.

At this point, it became more efficient to use pair programming to tackle the harder extensions. We started work on simplify in Simplify.hs but never finished it due to not having enough time. I also defined another file, Lit.hs, to handle the new type Lit (stands for literal), which is used to make floating and integer handling easier but at the same allowing it to be extended for things like String. As a result, strings can be concatenated to the numbers (that have been converted into strings).

For operations like loading a file, and functions, they needed to be executed before getting input from the user again so I created a system where commands stored in the state are executed if they exist and get removed afterwards.

Finally, we gave print ability to print strings that are evaluated before they are printed, allowing for the dynamic printing of variable values.

Conclusion

To conclude, through the development of this calculator application and the implementation of features with different difficulties, we are a lot more confident with the fundamental features and structure of Haskell applications. We have learnt how to program with recursive equations, algebraic data types, manipulating lists and working with functional parsers. Our program supports variable assignment and update, all arithmetic operations, clean error handling, as well as, all of the extension features except for the simplify command.

References

1. Hutton, G. *Programming in Haskell*. Cambridge, UK: Cambridge University Press, 2007
2. Thompson, S. Haskell *The Craft of Functional Programming* Essex, UK: Pearson Education Limited, 2011
3. Marlow, S. *Parallel and Concurrent Programming in Haskell.* Sebastopol, CA: O'Reilly Media, Inc, 2013.
4. Lipovaca, M. *Learn You a Haskell for Great Good!* No Starch Press, 2011. Web. 10 Feb 2015.
5. O'Sullivan, B., Stewart, D. and Goerzen, J. *Real World Haskell* Sebastopol, CA: O'Reilly Media, Inc, 2009. Web. 10 Feb 2015.
6. Shevchenko, D. Haskell humanly. The MIT License, 2015. Web. 10 Feb 2015. `http://ohaskell.dshevchenko.biz/en/index.html`