CPSC 3740

Course Project Report

Haskell Postfix Calculator

Dustin Ward

001211098

April 2nd, 2020

Program Execution

The program exists in the included “calculator.hs” file. I did all of my writing in an online interpreter running “GHCi version 8.6.5”, so I did not test on a local machine install. I don’t think this will affect anything, but in case it does here is the interpreter.

To use the calculator, all you have to do is load the “calculator.hs” file, and run “calculator”. You will then be prompted to enter a postfix expression. This expression must be a valid postfix expression where each digit/operator is separated by a space. No other formatting is required.

After pressing enter, the program will evaluate the expression and print the result. To use the calculator again, you must run the program again. There is no main input loop implemented.

Design Overview

The main function “calculator” is what drives the program. The user will call this function, and the calculator process will begin. It asks the user for input, and then passes the input as a string to “calculate”. Once the result has been received, it prints it to the terminal and ends the process.

“calculate” is the function that handles the actual calculation. The parsing is taken care of by the `words` function inside of the `foldl` call. This will split the string into its individual tokens. This is why input must be separated by spaces. You could most likely create some other parsing function that splits the string to avoid having spaces, but I figured that this implementation would be sufficient. Once the `words` function has created a list of digits and operators, the `foldl` function will recursively apply `f` to each element. `f` is a function that defines how the operators function. There is one version of `f` for each operator. This way the correct version can be run depending on the operator symbol that was chosen. The remaining element in the list will be the result to be returned.

Sample Input / Output

	=	0.0
4 5 7 2 + - *	=	-16.0
3 4 + 2 * 7 /	=	2.0
5 7 + 6 2 - *	=	48.0
4 2 3 5 1 - + * +	=	18.0
4 2 + 3 5 1 - * +	=	18.0
3.14159 2 / sin ceil	=	1.0
3.14159 4 / cos 2 * 2 ^ floor	=	2.0
3.14159 tan	=	-2.653589...e-6
12 8 gcd	=	4.0
3 4 lcm	=	12.0
2 8 log 2 %	=	1.0
16 sqrt 8 - abs	=	4.0

Conclusion

The first thing that I have come to realize about haskell is that there are many ways to do the same task. While I was researching how to do specific things I would come across many implementations that would more or less achieve the same result. Whether this is a good thing or a bad thing, i'm not sure. You could argue that it adds a lot of expressivity to the language and makes it so that an experienced user can do some very specific and powerful operations with few lines of code, or you could argue that it makes code very hard to understand and read because it is unlikely that two programmers would think the same way about implementation.

The parsing of the string is an example of such. I saw many examples talk about how they handled the list of tokens. The example provided in the project description used a register table to store the operator symbols, and then looked each one up at run time. I chose to not use this approach because the operator precedence is handled by the expression being postfix, therefore a simpler (to me…) implementation would do the same work.

One of the difficulties in the process was keeping track of the types of variables. As I added more complex operations, the compiler kept requesting that a different type of variable be used. First, adding division required that I upgrade to `Double` instead of

an `Integer`, but that one is fairly obvious, and my other functions can use double. Adding functions like `mod` and `gcd` needed the `Integral` type, so those operations were extracted into their own function that uses the `fromIntegral` function to convert them to `Double`. Another issue that I ran into was how the `foldl` function handled the list. I assumed that `f (x:y:xs)` meant `x y` and then some operator. (Meaning x was the left hand side, and y was right). But it was actually `y x`, where y comes before x in the expression. This meant that some order specific operations were running in reverse. (‘-’, ‘/’, etc.) Once I realised why my calculator wasn’t giving me the right answers, the solution was simple. I just had to swap x and y.