COSC2451: Programming Techniques
Assignment 2
Extensible Calculator Shell
Semester B, 2013

Denis Rinfret

Due: end of Week 12, Sunday September 8, 2013 at 23:59.
Assignment to be done individually.
Total of 50 marks.

1 Goal

The goal of this assignment is to make an *extensible* calculator shell (or interactive command line), similar in many ways to a Python interactive shell. Doing a calculator shell is not that hard, but doing one that is easily extensible with good programming practices is more difficult.

2 Parsing Command Line Arguments (3 marks)

Create a file called `calc.c` . This file will contain the `main` function to execute your program. This file should be compiled into an executable file called `calc` . You have to include the header file `getopt.h` to parse the command line arguments given to your program. Using `getopt.h` is how experts usually parse command line arguments.

Your program can be invoked with the following command line arguments:

```
calc [options] rpn|in
```

There are 2 possible options (both optional): `-b` and `-e`. The square brackets mean that everything inside them is optional. `-b` stands for *batch mode*, meaning that no prompt will be printed. `-e` stands for *echo*, meaning that you have to echo (or print) an input line immediately after reading it.

The vertical bar `|` means that you have to choose one of the 2 options given. `rpn` stands for *reverse polish notation*, and `in` stands for *inorder*. This choice between `rpn` and `in` will determine the mode in which your calculator is going to work.

Start by writing your main function to process these command line arguments with `getopt.h` and make a loop that is going to print your prompt (unless the `-b` option was given), read a line of input from the standard input, echo the input back if the `-e` option was given, and do nothing else for now. Your prompt should be your student id with 3 *greater than* characters:

```
s1234567>>>
```

3 Testing

Test what you have up to now not only in interactive mode, but also by using commands such as this one:

```
./calc -b -e rpn < test1.txt
```

Where the file `test1.txt` contains expressions to be computed, one expression per line.

4 RPN (3 marks)

Reuse the `rpn.h` and `rpn.c` files give in class to implement the `rpn` mode of your calculator. Each line of input given to your program should be a valid RPN expression. You have to parse it, evaluate it and print the results. If there is an error in the expression, you have to print a meaningful error message.

Assume that tokens in the expressions to evaluate are separated by spaces, and that numbers are going to be **real** numbers, **not integers**. You have to modify the code to deal with values of type `double` . The only operators you have to handle at this point are `+ - / *` and they have the usual meaning.

5 Inorder (6 marks)

Implement the `in` mode of your calculator. Assume at this point that the input expressions will contain parentheses `( )` everywhere so that there will never be any ambiguity about the precedence of operators. Every expression will be written, for example, in this way:

```
( ( 2 + 5 ) * 4.2 )
```

6 Make it Extensible (6 marks)

In the original RPN code there was an ugly part in the code: a series of `if ... else ... if ...` to execute the proper function corresponding to the operator. This works for small number of operators, but it is hard to upgrade the code to more operators. The code would become very ugly if we add many more operators to support.

Make your code more extensible by using function pointers, similar to what you did in the first assignment. There should be one function for each operator, and a table mapping each operator to a function pointer. Then you can replace the ugly code mentioned above by a look up in the operator table.

An operator can be 1 or 2 characters long. All operators are binary operators.

7 Add Operators (4 marks)

Add the following operators to your program:

`**` power operator. Example: `2 ** 5` = 32

`#` maximum operator. Example: `2 # 5` = 5

`_` minimum operator. Example: `2 _ 5` = 2

`||` distance operator. Example: `2 || 5` = `5 || 2` = 3

8 Variables and Assignments (10 marks)

Add the assignment operator `=` to assign the value of an expression in the right part of the expression into a variable on the left. For example, `x1 = 5.7` will assign the value 5.7 to the variable `x1` . The value resulting from the assignment is the value assigned to the variable. Therefore, expressions like this one `x2 = x1 = 5.7` are valid.

Variables do not have to be declared before being assigned to. The first time a variable is assigned to will create the variable. A variable can be used on the right side of an expression only after it has been assigned to at least once.

The first way to handle variables in your program is to use a fixed-number of variables `x0, x1, x2, ..., x9` similar to fixed registers. Only these 10 variables will be available in your program. The second way is to have variables similar to many programming languages: any number of variables is permitted and variable names should start with 1 letter followed by any number of letters or digits.

The first way to handle variables will be worth maximum 3 marks than the second way maximum 6 marks.

9 Improving the Inorder Expressions (8 marks)

In this part you can make parentheses optional in your inorder expressions. To do that you have to handle the precedence of your operators. The highest precedence should go to `**`, then `*` and `/` , followed by `+` and `-`, then `#` and `_`, and finally `||`.

10 Code Quality and Documentation (10 marks)

Code quality is very important in this assignment. Your code has to be documented properly, you have to use a proper Makefile and your code has to be broken down into modules and functions in a proper way. You don't have to write a report, but you have to write your documentation using the doxygen format and generate the HTML documentation from your code.

11 Submission Format

You should create a folder named `COSC2451_A2_s1234567`, put all your files inside it, and then zip it and upload it to Google Drive. Please clean up your folder by executing `make cleanall` before zipping your folder. Your folder should contain your source code, your `Makefile`, your doxygen configuration file, and a `doc` folder containing the documentation generated by running doxygen.

The default target in your `Makefile` should compile everything and create your `calc` executable. There is **no** need to have a `tests` target in this assignment.

12 Submission Procedure

Upload your zip file to your Google Drive and share it with the lecturer.

13 Notes

- Not following the submission format and/or the submission procedure might incur a penalty.
- Late submissions will receive the usual 10% penalty per late day, up to a maximum of 50%. After 5 late days, assignments will not be marked.