Project 1 - AdvCalc

CmpE 230, Systems Programming, Spring 2023

Instructor: Can Özturan
TA: Gökçe Uludoğan
SAs: Bahadır Gezer, Ömer Talip Akalın

Due: 28/03/2023, 09:00 Sharp

1 Introduction

In this project, you will implement an interpreter for an advanced calculator using the C programming language. The advanced calculator (AdvCalc) will accept expressions and assignment statements.

2 Details

Possible expressions in the AdvCalc language are given in the table below. Expressions of this language will be infix expressions that may contain the followings:

a + b	Returns summation of a and b.
a * b	Returns multiplication of a and b.
a - b	Returns the subtraction of b from a.
a & b	Returns bitwise a and b.
a | b	Returns bitwise a or b.
xor(a, b)	Returns bitwise a xor b.
ls(a, i)	Returns the result of a shifted i bits to the left.
rs(a, i)	Returns the result of a shifted i bits to the right.
lr(a, i)	Returns the result of a rotated i times to the left.
rr(a, i)	Returns the result of a rotated i times to the right.
not(a)	Returns bitwise complement of a.

You will implement the interpreter based on the following assumptions:

- Every value and every calculation will be integer-valued.
- There will be no expressions or assignments with a result that exceeds a 64-bit number.
- Similarly, every bit-wise intermediate operation will abide by the 64-bit limit.
- The language does not support the unary minus (-) operator (i.e x = -5 or a = -b is not valid). However, as can be seen above, the subtraction operation is allowed.
- The variable names will consist of lowercase and uppercase Latin characters in the English alphabet `[a-zA-Z]`.
- Expressions or assignments will consist of 256 characters at most.
- Undefined variables have a value of 0.

- '%' characters denote comments. Any characters after '%' will be considered as a comment, not code.
- The execution of the program will be done through the interpreter screen working in the terminal - in other words, it won't be a file-based program. It should work just like the Python interpreter.
- You must run and test your code on an Ubuntu Linux machine before submitting. You can use a Linux virtual machine or WSL in this context.
- Apart from the valid expressions and statements, the AdvCalc interpreter must state an error in case of a faulty operation. However, the program should continue to run. For such cases, an error message should be displayed in a new line in the exact form below:

  ```
  Error!
  ```

- The input may include all sorts of syntax errors.
- All of the following can be given as an input - and all of them are valid: a + b, a+ b, a + b, a+b, a +b, (((a))) + (b)

3 Submission

Your project will be tested automatically. Thus, it's important that you carefully follow the submission instructions. The root folder for the project should be named according to your student id numbers. If you submit individually, name your root folder with your student id. If you submit as a group of two, name your root folder with both student ids separated by an underscore. You will compress this root folder and submit it with the `.zip` file format. Other archive formats will not be accepted. The final submission file should be in the form of either `2020400039.zip` or `2019400046_2020400039.zip`.

You must create a Makefile in your root folder which creates a `advcalc` executable in the root folder, do not include this executable in your submission, it will be generated with the `make` command using the Makefile. The `make` command should not run the executable, it should only compile your program and create the executable.

Late Submission

If the project is submitted late, the following penalties will be applied:

Hours late	**Penalty**
0 < hours late <= 24	25%
24 < hours late <= 48	50%
hours late > 48	100%

4 Grading

Your project will be graded according to the following criteria:

- **Code Comments (8%):** Write code comments for discrete code behavior and method comments. This sub-grading evaluates the quality and quantity of comments included in the code. Comments are essential for understanding the code's purpose, structure, logic, and any complex algorithms or data structures used. The code should be easily readable and maintainable for future developers.

- **Documentation (12%):** A written document describing how you implemented your project. This sub-grading assesses the quality and completeness of the written documentation accompanying the project. Good documentation should describe the purpose, design, and implementation details of the calculator interpreter screen, as well as any challenges encountered and how they were addressed. The documentation should also include examples of input/output and how to use the program. Students should aim to write clear, concise, and well-organized documentation that effectively communicates the project's functionality and design decisions.
- **Implementation and Tests (80%):** The submitted project should be implemented following the guidelines in this description and should pass testing. This sub-grading assesses the quality and correctness of the implemented calculator interpreter screen, including its functionality and accuracy in handling expressions and assignment statements.

5 Warnings

- You can submit this project either individually or as a group of two.
- All source codes are checked automatically for similarity with other submissions and exercises from previous years. Make sure you write and submit your own code. Any sign of cheating will be penalized with an F grade.
- Do not use content from external AI tools directly in your code and documentation. Doing so will be viewed as plagiarism and thoroughly investigated.
- Project documentation should be structured in a proper manner. The documentation must be submitted as a .pdf file, as well as in raw text format.
- Make sure you document your code with necessary inline comments and use meaningful variable names. Do not over-comment, or make your variable names unnecessarily long. This is very important for partial grading.
- Do not start coding right away. Think about the structure of your program and the possible complications resulting from it before coding.
- Questions about the project should be sent through the discussion forum on Piazza.
- There will be a time limit of 30 seconds for your program execution. Program execution consists of the total amount of time for all queries. This is not a strict time limit and the execution times may vary for each run. Thus, objections regarding time limits will be considered if necessary.

6 Examples

Below are some examples that may be useful.

Example 1:

```
% ./advcalc
> x = 1
> x * 3
3
> y = x - 4 * (x + x)
> y
-7
> <Ctrl-D>
%
```

Example 2:

```
% ./advcalc
> 1 + -1
Error!
> xor(((5 + 3) * 7), (ls(10, 2) & (rs(15, 1) | 12)))
48
> h = (rs(11, 2) | ls(4, 1)) * (6 - not(3))
> ((9 * 2) - 15) & (rs(7, 2) xor ls(4, 1))
Error!
> h
100
> h = (rs(h, 1) & (not(h) | ls(6, 2))) + not(2)
> h
15
> <Ctrl-D>
%
```

Example 3:

```
% ./advcalc
> x = 1
> y = x + 3            % 4
> z = x * y * y*y      % 64
> z
64
> qqq = xor(131, 198)
> qqq
69
> xor(((x)), x)
0
> xor(((x)), x) | z + y
68
> rs(xor(((x)), x) | z + y, 1)
34
> ls(rs(xor(((x)), x) | z + y, 1), (((1))))
68
> lr(ls(rs(xor(((x)), x) | z + y, 1), (((1)))), 1)
136
> rr(lr(ls(rs(xor(((x)), x) | z + y, 1), (((1)))), 1), 1)
68
> qqq * not(not(10))
690
> rr(lr(ls(rs(xor(((x)), x) | z + y, 1), (((1)))), 1), 1) - qqq * not(not(10))
-622
> 0 & rr(lr(ls(rs(xor(((x)), x) | z + y, 1), (((1)))), 1), 1) - qqq * not(not(10))
0
> <Ctrl-D>
%
```

Example 4:

```
% ./advcalc
>            5555555555
5555555555
>            5555         555555
Error!
> t = ((11 + 2) * 5) | (rs(121, 5) & rs(30, 2))
> t
67
> qt = t + QTQTQT              % Please note that undefined variables are 0.
> qt
67
> t & t | t + xor(t, ls(qt, 2))
467
> Sad
0
> Mega Sad
Error!
> Giga + Sad
0
> https://youtu.be/dQw4w9WgXcQ
Error!
> <Ctrl-D>
%
```