

2023 Spring

CmpE 230 Project I - Report

Mustafa Berk Turgut
Nurullah Uçan

Table Of Contents

Overview	2
Implementation Details	2
Interpreter Screen	3
How to Use the Program	4
Input Output Examples	4
Problems Encountered	5

Overview

This is a code for a simple calculator that can evaluate arithmetic expressions and also perform bitwise operations. It also supports variable assignment and reusing the assigned variables.

Implementation Details

The program takes input from the user in the form of an expression and tokenizes it. The input ending with the '\n' character is tokenized and tokens are stored in the tokens array. Then begin() function starts the parsing process. The code also includes an 'evaluator()' function that evaluates the expression and prints the result to the output.

The evaluator function is responsible for calculating the parsed input stored in str in postfix form. It starts by adding the tokens to stack1 and then reverses the stack and stores it in stack2. The function then pops an element from stack2 and makes the operation using the two topmost operands in stack1. The result is then pushed back to stack1. This process is repeated until there are no more elements left in stack2.

The other functions in the program implement different parts of the parser, such as 'begin', 'assignment', 'hyperexpr', 'expr', 'term' and 'factor'. These functions implement the parsing logic to parse the expression, and when a parse tree is possible, all functions return 1 and generate "str", which stores the tokens in postfix notation.

We originally constructed a non-ambiguous grammar with the following production rules:

Production Rule	Corresponding Operator
$\langle \text{start} \rangle \rightarrow \langle \text{assign} \rangle \mid \langle \text{hyperexpr} \rangle$	
$\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle \text{"="} \langle \text{hyperexpr} \rangle$	=
$\langle \text{hyperexpr} \rangle \rightarrow \langle \text{hyperexpr} \rangle \text{" "} \langle \text{superexpr} \rangle \mid \langle \text{superexpr} \rangle$	
$\langle \text{superexpr} \rangle \rightarrow \langle \text{superexpr} \rangle \text{"\&"} \langle \text{expr} \rangle \mid \langle \text{expr} \rangle$	&
$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \text{"+"} \langle \text{term} \rangle \mid \langle \text{expr} \rangle \text{"-"} \langle \text{term} \rangle \mid \langle \text{term} \rangle$	+ / -
$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle \text{"*"} \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$	*

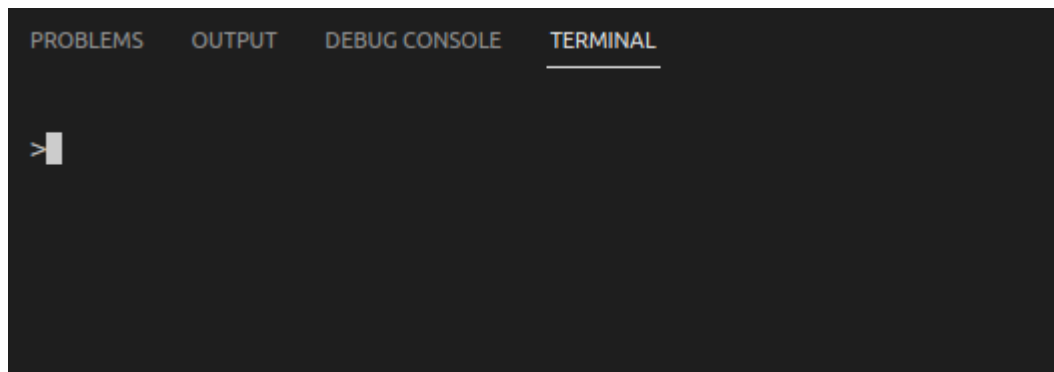
<code><factor> → "(" <hyperexpr> ")" <var> <num> <function></code>	<code>()</code>
<code><function> → "rs" "(" <hyperexpr> "," <hyperexpr> ")" "xor" "(" <hyp..> ")" ... "not" "(" <hyperexpr> ")"</code>	All functions

Before implementing the grammar, we transformed the production rules to non-left recursive versions. We used helper functions `is_number` and `is_variable` to parse numbers and variables.

Lastly, in order to assign and retrieve variables, we implemented a dictionary like structure with two arrays. The function `'add_item_to_dict'` is used to add a variable and its value to this dictionary. When the `'='` operator is encountered, the program stores the variable name and its corresponding value in a dictionary. When a variable is encountered in the expression, the program looks it up in the dictionary and retrieves its value.

Interpreter Screen

When you run our program, you get the interpreter screen as following:



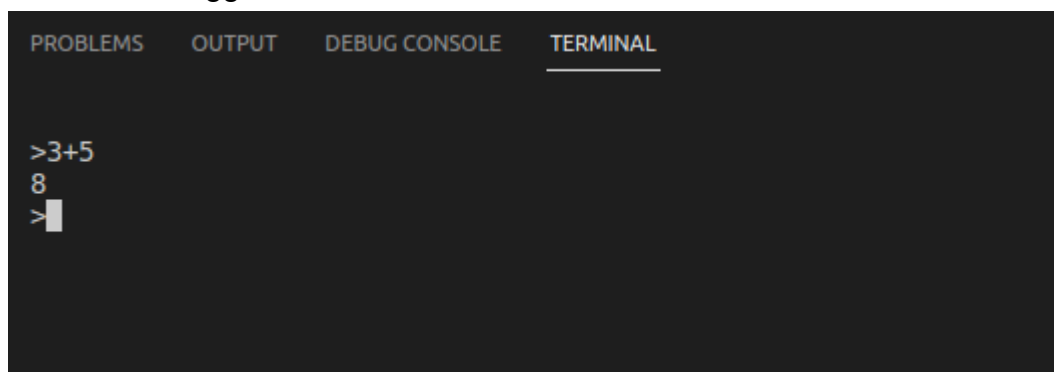
```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

>

```

The program then takes the input, prints the result in the next line if there is an output, and continues printing the `'>'` character in the next line, ready to take input until EOF is triggered.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

>3+5
8
>

```

How to Use the Program

The conventional way of using the program through the interpreter screen is explained above. One other way to run the program on multiple lines of input is through the terminal.

In order to get an executable of the advcalc.c, you should run “make” command through the terminal. It will then generate advcalc, which you can run by ./advcalc command to get the interpreter mentioned previously, or you can directly give the input through a file via terminal, explained below.

```
./advcalc < input.txt
```

This code runs the executable file created by Makefile, and passes the content of input.txt as parameters. The input and the output forms are explained in more detail in the next section.

Input Output Examples

For the following file named input.txt, this is the output our program generates, which complies with the result provided in the project document.

```
CmpE230-Parser > input.txt
1  % ./advcalc
2  x = 1
3  y = x + 3 % 4
4  z = x * y * y*y % 64
5  z
6  qq = xor(131, 198)
7  qq
8  xor((x), x)
9  xor((x), x) | z + y
10 rs(xor((x), x) | z + y, 1)
11 ls(rs(xor((x), x) | z + y, 1), ((1)))
12 lr(ls(rs(xor((x), x) | z + y, 1), ((1))), 1)
13 rr(lr(ls(rs(xor((x), x) | z + y, 1), ((1))), 1), 1)
14 qq * not(not(10))
15 rr(lr(ls(rs(xor((x), x) | z + y, 1), ((1))), 1), 1) - qq * not(not(10))
16 0 & rr(lr(ls(rs(xor((x), x) | z + y, 1), ((1))), 1), 1) - qq * not(not(10))
```

```
>>>>64
>>69
>0
>68
>34
>68
>136
>68
>690
>-622
>0
```

Note: When the input is given via a file, ‘>’ signs add up in the same line, which is not the case in the interpreter mode as a user gives an input it ends with the next line ‘\n’ character.

Problems Encountered

There were several problems we encountered during the development of the project. Some of those problems and how we managed to solve them are as follows:

- Using strcmp() to compare a character and a string

We figured out that a string such as `x = "u"` is different from a char `y = 'u'`, as the former one includes a null character and is stored as `"u\n"`, which is different from `'u'`. Therefore we used the format `strcpy(x,"u")` for a string `x`, or `y == 'u'` for a char `y`.

- Receiving error for not(expr) functions due to a bug in evaluator()

In the while loop initialized under the line `"cur_eval = 0;"`, we first tried to implement a switch-case statement for the evaluation of different functions, but we later stucked with a series of if statements, which caused `"not"` to go both as a function and go to the stack as a variable named `"not"` and undefined value, hence 0. We fixed the issue by inserting a continue statement, so that when `not` is parsed, operation completes and while loop continues with the next iteration.

- Modified cur value unable to parse begin -> assignment | hyperexpr

An input can be either an assignment or a hyperexpression. When our code tries to parse it, through a series of calls subparts are identified and the cursor is moved accordingly. However, when generation of a parse tree through assignment fails, the cursor holds an altered value and must be reset before parsing the hyperexpression. To that end, we added `cur = 0` in between the function calls in `begin()`.