lab4: Interpreter

This is a two-week lab with 200 points. A 5% bonus(10 points) is granted to those who complete this lab in one week. You can contact Huijie Tang(tanghuijie@sjtu.edu.cn) if you find anything wrong in this lab.

1. background and object

An **interpreter** is a computer program that directly executes instructions written in a programming or scripting language. For example, MatLab is executed through an interpreter.

Compared with a compiler, which turns a program into an executable binary file(for example, .exe on windows), an interpreter does not generate binary files directly.

generally speaking, *Compiling principle* is an advanced CS course for undergraduate students. In this lab, we only require you to implement a very simple MatLab interpreter, which can calculate the value of expressions and store variables.

Your objective is to read commands from a txt file, and you need to parse these commands and output the results line by line.

2. Syntax

Input: a txt file named "commands.txt"

Output: stdout (i.e., output to the screen with `printf`).

1. basic operators

"`+`", "`-`", "`*`", "`/`", "`^`","`()`" with the meaning identical to matlab.

Notice that "`^`" in C is not the meaning of power. You may use `pow()` in `math.h`

The output can all be a "`double`" type

example:

input(in commands.txt):

```
2+ 8
6*(2 +9* 5/1.6)
2^4.6/(3*6)
-3.2+8
```

stdout:

```
10.000000
180.750000
```

```
1.347304
4.80000
```

Notice that you need to consider possible spaces between the operator and numbers.

You can use **Reverse Polish expression** and **stack** to parse the expression, which will be covered in part 4 in this file.

2. variables and assignments operator("=")

You need to implement a variable system that can store and use variables in the expressions.

example:

input(in commands.txt):

```
a=2.1
b=a^(6+2.8)
cx=a+b^0.9
cx=cx+1
```

stdout:

```
2.100000
684.746457
358.532104
359.532104
```

3. semicolon requirement

Similar to matlab, if an expression has ";" at the end, there will be no output to the screen.

input:

```
a=2.1;
b=a^(6+2.8);
cx=a+b^0.9;
cx=cx+1
```

stdout:

```
359.532104
```

3. restrictions

To make your life easier, we make some restrictions.

1. We promise that there won't be any syntax errors in the testcases, so you can suppose all the commands in "commands.txt" are valid.
2. In expressions, you needn't consider multiple "`^`" cases. For example, `4^5^6` actually means `4^(5^6)` in mathematics, but you needn't consider this case. We won't have testcase like this.
3. Each line of command will consist of less than 100 characters, which means you can use `char[120]` to store one line of command.
4. There will be at most 20 variables, and a variable's length is at most 10.

4. Reverse Polish expression

To deal with complex computational commands with multiple operators including parentheses, here we can use a useful method: convert the expression to reverse polish expression first, then calculate the reverse polish expression. Before asking what's a reverse polish expression, you need to know something about "Stack".

4.1 stack

"Stack" is a data structure that stores some data. Unlike array which allows you to get and modify the value at any place, stack only supports two basic operations:

1. Push: Add a new element to the end of the stack
2. Pop: remove the element at the end of the stack

You can simply use an array(with enough space) and an index number(int) to implement a stack. The index number indicates the top+1 of the stack.

The way to initialize an empty stack (which stores elements with "`double`" type) is:

```
double stack[120]={0.0}; // initialize every element as 0.0
int top=0;// top is the number of elements in the array
```

The way to push an element(for example, 4.5) to the stack is:

```
stack[top++]=4.5;
```

explain: 4.5 is the variable that we want to push into the stack. since `stack[top]` isn't occupied by any data, we can simply assign 4.5 to `stack[top]`. Then we need to make `top=top+1` because now there's one more element in the stack. We can put these two sentences together, and that's `stack[top++]=4.5;` (note: the index of array in C begins from 0)

The way to pop an element from the stack is:

```
element=stack[--top];//suppose variable "element" has already been declared
```

explain: by making `top=top-1`, we have already popped out the last element. If we want to know what we have popped out, we need to have `element=stack[top-1];` before making "top" smaller. We can put these two sentences together, and that's `element=stack[--top];`

4.2 Reverse Polish expression

Reverse Polish expression, also known as postfix expression, can help us deal with an expression.

"1+4" is a so-called "Infix expression", which means the operator "+" is in the middle of two corresponding operands. This kind of expression is not friendly to computers because of the operator's priority issues. For example, in the infix expression "1+(1+4)*(5+7)", as a human being, you know how to calculate it in the correct order, but you have no idea how to write a program to make the computer know the right order.

"postfix expression", however, makes the computer's life easier. It means to put two operands at the beginning and put the operator at the end of the two operands. In this way, you'll find it easier to calculate its value.

For example, there's an expression "1+4", the reverse polish expression is "1, 4, +".

For expression "(1+4)*(5+7)", the reverse polish expression is "1, 4, +, 5, 7, +, *".

4.3 turn Infix expression to Reverse Polish expression

How to turn normal infix expression into reverse polish expression? That's the hardest part. However, we can use a stack and a set of rules to complete this task.

Follow the rules for an infix expression, and what you print is the reverse polish expression. You need a stack(initially it's empty) to help you.

> a. Print out operands(i.e, numbers) as they arrive.
>
> b. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
>
> c. If the incoming symbol is a left parenthesis, push it on the stack.
>
> d. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
>
> e. If the incoming symbol has higher precedence than the top of the stack, push it on the stack. (higher precedence means: "^" > "*" = "/" > "+" = "-")
>
> f. If the incoming symbol has equal precedence with the top of the stack, pop and print the top of the stack and then push the incoming operator.

> g. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of the stack. (do rule e-g to them again).
>
> h. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain)

Now what you print is a reverse polish expression.

Please notice that "**print**" here means store something for further use because our journey doesn't end here. We need the reverse polish expression to calculate the value.

Let's see an example for "(1+4)*(5+7)" step by step:

step	remain	stack	print	comment
0	(1+4)*(5+7)			
1	1+4)*(5+7)	(		
2	+4)*(5+7)	(	1	
3	4)*(5+7)	(, +	1	Rule b
4	)*(5+7)	(, +	1, 4	
5	*(5+7)		1, 4, +	Rule d
6	(5+7)	*	1, 4, +	Rule b
7	5+7)	*, (	1, 4, +	
8	+7)	*, (	1, 4, +, 5	
9	7)	*, (, +	1, 4, +, 5	
10	)	*, (, +	1, 4, +, 5, 7	
11			1, 4, +, 5, 7, +, *	Rule d

" 1, 4, +, 5, 7, +, * " is the reverse polish expression.

4.4 calculate the Reverse Polish expression

The final step is to calculate the expression and get the result. This is not hard.

We can still use a stack and follow these 2 rules:

> a. If you get an operand, push it into a stack.
>
> b. If you get an operator, pop two operands from the stack, calculates the result concerning the operator, and push the result back into the stack.

At the end of the reverse polish expression, only one element remains in the stack. This is exactly the result.

Take " 1, 4, +, 5, 7, +, * " as an example:

step	remain	stack	calculation
0	1, 4, +, 5, 7, +, *		
1	4, +, 5, 7, +, *	1	
2	**+**, 5, 7, +, *	**1, 4**	1 + 4
3	5, 7, +, *	5	
4	7, +, *	5, 5	
5	**+**, *	5, **5, 7**	5 + 7
6	*****	**5, 12**	5 * 12
7		60	

5. General tips

1. how to store the reverse polish expression

You may think it's hard to store a reverse polish expression like "1.3, 4.5, +, 3.1, 6, -, /". You won't store it in a C-style string because you will find it hard to calculate its result. So, one feasible way we recommend is to store it in a double array `double[]`.

So, how could we store operators like '+'? One way is to store its ASCII, for '+', its ASCII is 43. However, it will confuse it with the number 43. But you can use an auxiliary array to mark out whether the element represents a number or an operator's ASCII.

This is an example of how it works:

```
double RVP[120]={1.3,4.5,'+',3.1,6,'-','/'};
int helper[120]={1,1,2,1,1,2,2};//0 means not occupied, 1 means double, 2
means char
```

When printing the reverse polish expression into array RVP in 4.3, you also need to update the corresponding place at the helper array, to 1(if it's an operand) or 2(if it's an operator)

Then when you need to operate on one of the elements in the array RVP, for example, RVP[5], you also need to check helper[5] to determine if it's an operator or an operand.

2. how to store the variables

Since there will be at most 20 variables, you can use `char vars[20][15]={{'0'}};` to store all the variable names. use `int varnums=0;` to store the total numbers of the variables, and `double varvalue[20];` to store the value of the variables.

6. Rubrics

1. Be able to calculate an expression(2.1): 125 pts
2. Be able to assign a value or expression to a variable, and calculate an expression with variables(2.2): 50 pts
3. Be able to suppress the printing(2.3) and complete this lab successfully: 25 pts
4. (bonus) Finish the lab in one week: 10 pts.

7. Reminders

I understand that this lab is very hard. However, your hard work will pay off after the lab. You will have a better understanding of C-style String & Array if you keep thinking, searching on the internet, and debugging. Of course, you need to start early.

You're recommended to write some string-related functions to help you, for example, insert a string into another string, because you may perform these operations a lot. In this way, your codes can be shorter.

Chances are you will make some mistakes in this lab. So, you need to know how to debug. You shouldn't write all your codes and run all of them afterward. In this case, you'll find your codes not working and you don't know what's the mistake. Instead, you should debug module-by-module. For example, when you finish the part of transforming reverse polish expressions, you can output them to the screen and see if they meet your expectation. Then you can go on to work on the next part.

8. Submission

Your solution should be written into a .c file called lab4.c. Please submit your code to JOJ by compressing the codes into a single zip file in the name format: lab4-[Your Last Name]-[Your First Name]-[Your SJTU ID].zip, for example, lab4-Tang-Huijie-521021910711.zip. When you come to the lab, show the file to TA according to TA's instructions. After the lab, you need to submit the same zip file to canvas.

9. Acknowledgment

Shuyu Wu(TA in VG101 FA2021), VG101 FA2021 lab4