Take-Home Exam 3

(Deadline: 11th of January 2023, Wednesday, 23:59)

1 Introduction

Strings are frequently dealt with in programming languages and text mining. You will solve two tasks that give a glimpse of programming with strings in that regard.

2 Tasks

2.1 The 99 Interpreter (50 pts)

In this task, you will write a program that acts as an *interpreter* for a simple programming language that we call **99**. Your interpreter accepts and executes the commands each of which is given on a separate line. The following are the list of commands you are supposed to implement:

Command	Effect
`set [var] [value]`	Sets the variable `[var]` to `[value]`. `[value]` is written as an integer inclusively between 0 and 99, without any leading zeroes.
`add [varD] [varS]`	Stores `[varD] + [varS]` in `[varD]`. If the result of the addition is more than 99, it is truncated to 99.
`sub [varD] [varS]`	Stores `[varD] - [varS]` in `[varD]`. If the result of the addition is less than 0, it is truncated to 0.
`show [var]`	Prints the value of `[var]`. The format is "`[var] = value`".
`goback [lines] [varD] [varS]`	Go back `lines` lines if `[varD]` $<$ `[varS]`. The interpreter should automatically continue executing the commands as necessary. The execution will get past this point only if the `goback` command is re-executed while `[varD]` $\geq$ `[varS]`.
`exit`	Exits the interpreter.

Note the following regarding the commands:

- You are supposed to *ignore any whitespace* in the commands (unless they separate fields).
- Any undeclared variable is *automatically declared* with the *initial value of* 0 whenever it is referred.
- Command & variable names are *case-sensitive*.
- Traditionally, an interpreter executes commands one by one, providing immediate feedback. However, to simplify your job in this task, we do not expect the immediate feedback behavior; your interpreter can read the *whole program* before starting any execution.

The following will be guaranteed to your interpreter.

- There will be at most 99 commands.
- All commands (including the last one) are to end with a newline character.
- Any valid command line contains at most 99 characters (excluding the newline character).
- All entered commands has correct syntax.
- The program is guaranteed to finish its execution by executing `exit`.
- All variable names are alphanumeric.
- All mentioned values are between 0 and 99, inclusively.
- All `goback` commands are to a valid previous line.

Example Input	Example Output

`set ONE 1`
`set i 10`
`add sum i`
`sub i ONE`
`goback 2 ZERO i`
`show sum`
`exit`
	`sum = 55`

2.2 Text Parsing (50 pts)

You are given a large piece of *text* some of whose subparts are marked with two *delimiter* words. In particular, a piece of text is *marked* if it is situated between the *start delimiter* and the *end delimiter*. Write a program to find and return the marked parts. The parts should be returned in increasing order of *their length*. For parts with the same length, sort them by increasing order of *their position* in the text. As a final task, you are expected to print the text *without the delimiters*.

The input format will be:

s

e

t

where s is the start delimiter, e is the end delimiter and t is the text.

The following apply:

- The text has at most 100 characters and does not contain any null characters.
- The delimiters have at most 10 characters each and do not contain any whitespace or null characters.
- The text may contain newline characters, which should be ignored.
- The newline characters and the delimiters count toward the 100 limit.
- The delimiters are different.
- The delimiters are guaranteed to be unambiguously identified in the text. No start/end delimiter overlaps with another start/end delimiter.
- The delimiters match; the first delimiter to the right of each start delimiter is an end delimiter and the first delimiter to the left of each end delimiter is a start delimiter.

The output format will be:

m_1

$\vdots$

m_k

u

where k is the number of marked parts, m_i are these parts sorted as described and u is the text without the delimiters.

Example Input	Example Output

`%`
`#`
`Hi %pal!#`
`It is %a nice day#.`
	`pal!`
	`a nice day`
	`Hi pal! It is a nice day.`

`[begin]`
`[end]`
`[begin]CENG 140[end] is [begin]`
`great[end].`
	`great`
	`CENG 140`
	`CENG 140 is great.`

`(`	`at`
`)`	`ul`
`Congr(a`	`at`
`t)(ul)(at)ions!`	`Congratulations!`
`{{`	`king`
`}}`	`gard`
`They are ta{{king}} the {{hobbit}}s`	`hobbit`
`to the Isen{{gard}}!`	`They are taking the hobbits to the Isengard!`

Output explanation:

- On the 1^{st} example, "`pal!`" is shorter than "`a nice day`", so "`pal!`" is written first.
- On the 2^{nd} example, "`great`" is shorter than "`CENG 140`", so "`great`" is written first.
- On the 3^{rd} example, "`at`", "`ul`" and "`at`" have the same length, so they are written in their position order.
- On the 4^{th} example, "`king`", "`gard`" have the shortest length and are written in their position order. They are followed by "`hobbit`" which has larger length.

3 Submission & Rules

- Submit a separate C source file (with given name in ODTÜClass/VPL system) for each of the tasks.
- We will use VPL system for submission (which will be case for lab exams as well.) You can use the system directly as an editor or work locally and upload your source files to the system later.
- **Run** and **debug** modes in VPL system are disabled since we have 4 main functions exist in submission files. You can directly use **evaluate** mode in VPL to run all tasks at once and see your grade. You can evaluate your submission infinitely many times.
- Late submission ***IS NOT*** allowed. **Please, do not ask for any deadline extensions**.
- In your solutions, you should **not use any library functions other than** those in *"stdio.h"*, *"stdlib.h"* and *"string.h"*.
- We will compile your code with the following command:

```
gcc yourCode.c -o yourExecutable -g -std=c17 -Wall -Wextra -Wpedantic
```

- If you are working on a command prompt, you can likely feed input files to your program instead of typing inputs. This way, you can test your inputs faster. For instance, you can type:

```
> ./yourExecutable < yourInput.txt
```

- Your grade will be given by an auto-grader upon submission. The grade will be determined by running your code against several input files including, but not limited to the test cases given to you as an example. It is possible to get a partial grade from each task.
- You should stick to the output format described in the tasks. Your program's output should not contain any unnecessary characters or whitespace. (The only exception to this is that you may optionally print a newline character at the end of your output.) It is your responsibility to check the correctness of the output in terms of whether it contains any invalid invisible characters. This also means you are not supposed to print any prompts for input.
- Your programs should terminate with an exit code of 0.
- Your programs should be reasonably well-written such that they do not run slowly. The time limit we impose in the auto-grade is more than enough with respect to the tasks that you are assigned; however, if you have a bug or a code that is extremely slow, your program may exceed the time limit and you will not receive any points.
- Grade from the auto-grader is ***NOT*** final. We may re-evaluate your submissions in black-box or white-box manner and adjust your grade. Submissions that do not reasonably attempt to solve the tasks may lose points.

- As usual, you are expected to complete your solutions in ***academic integrity***. We have zero-tolerance policy for cheating. People involved in cheating will receive a grade of zero and may be prosecuted according to the university regulations. *Sharing code (in any amount) with each other or using third party code is strictly forbidden.* Be aware that there are “very advanced tools” that can detect this type of cheating.

Good Luck!