CS1010E: Programming Methodology

Assessed Lab 3: Functions & Recursions [10%]

15 Mar 2017

Instructions

Please read all the instructions very carefully!

1. This is an **Open Book** assessment:
   - You are allowed to bring any printed materials and calculator
   - You are NOT allowed to use other electronic devices besides the lab's computer
   - You are NOT allowed to talk with your friends, to talk with invigilators please raise your hand
   - You are NOT allowed to access the internet except to the `plab` server via `SSH terminal`
2. This lab assessment consists of **one (1)** problems with several tasks:
   - The tasks are intended to guide you in solving the problem
   - Each task should have **its own separate file** where the task number is written at the back: `task3.c` is used for task 3
   - To proceed to the next level (*e.g., from task 2 to task 3*), copy your program using the command `cp task2.c task3.c`
   - Fill in your **Name**, **Matric** (*starts with* `A`), and **NUSNET ID** (*starts with either* `A` *or* `E`)
3. Numerical and precision guides:
   - **Two (2)** types of *input* numbers: `real` (*may have decimal point*) and `integer` (*no decimal point*)
   - `integer` may contain leading *zeroes*: always use `scanf("%d")` to ensure *decimal* representation
   - `integer` has a range of -2^{31} to $+2^{31}-1$, `unsigned integer` has a range of 0 to $+2^{32}-1$
   - Always use `double` for `real` number input for high precision, but numbers that differs by less than `0.001` are considered *equal*
4. Starting the tests:
   - Use the program `SSH Secure Shell Client`
   - Login to `plab` server using the given `username` and `password`
5. Testing and debugging guides:
   - You may open **two (2)** or more `SSH Terminal`: 1 for *coding* and 1 for *compilation + testing*
   - Assumption stated in the task is considered to always hold and no checking is necessary
   - Assumption NOT stated in the task will be tested in hidden input: *always think of worst case*
   - Test case outputs are organized by task number and test case number:
     - Task number `T` on test case number `C` have output file `testT_C.out`
     - *For example*: task number 2 with test case number 3 have output file `test2_3.out`
   - Test case inputs are the same for all tasks: *e.g.,* `test2.in`
6. Marking:
   - Grading is done *automatically* using CodeCrunch: only the largest correct task is considered
   - For instance: Task 1 is *empty (i.e., not done at all)*, Task 2 is *correct*, Task 3 is *incorrect*
     $\mapsto$ mark for Task 2 is taken
   - The mark for each task is given on the right side, it is a *cumulative* mark
7. Time management suggestion: [Total Time: **1 hour 30 minutes**]:
   - *Coding*: approx. **1 hour** (±**30 minutes** for debugging)
   - *Ending*: approx. last **5 minutes** ensures that you save the filename correctly

Polish Notation [100 %]

Problem Description

"Polish notation (PN), also known as normal Polish notation (NPN), Lukasiewicz notation, Warsaw notation, Polish prefix notation or simply prefix notation, is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity. The Polish logician Jan Lukasiewicz invented this notation in 1924 in order to simplify sentential logic." – Wikipedia

We usually write a Mathematical expression in *infix* notation where the operator is placed in the middle. In the Polish notation, the operator is placed on top. As a limitation, we will only consider operator involving **two (2)** operands. The advantage of Polish notation is that Mathematical expressions are *unambiguous*.

For instance, the expression `4 + 5 * 7 - 8` is ambiguous without the bracketing. If we consider the standard operator precedence, the same expression written in Polish notation as `- + 4 * 5 7 8`. Now, if we add bracketing, we can see how the Polish notation maps more clearly as `(4 + (5 * 7)) - 8` is rewritten as `- (+ 4 (* 5 7)) 8`.

Consider a *sequence* of `characters` such that:

- All numbers are **one (1)** digit
- No brackets in the sequence
- Only addition (`+`) and multiplication (`*`) operators
- The sequence is written in the Polish notation without any `[space]` in the middle
- The sequence is terminated by an equal (`=`) sign

The problem is to evaluate the sequence to produce a single number corresponding to the result of evaluating the expression. This can be done via *recursion* as follows:

- Reading an operator:
  - You are guaranteed that there are **two (2)** Mathematical *sub-expressions* after the operator
  - You can evaluate these **two (2)** sub-expressions via recursion
  - If the sub-expressions are evaluated as **two (2)** numbers n_1 and n_2 with operator $\oplus$, the result of the entire expression is $n_1 \oplus n_2$
- Reading a number:
  - You are guaranteed that this is a **one (1)** digit number
  - The result of the entire expression is the numeric character represented as number

Concepts Tested:

1. Input/Output: `scanf` and `printf`
2. Modulo & Boolean Arithmetic: `%`, `||`, `&&`, `==`, etc
3. Selection Statement: `if` and/or `if-else`
4. Repetition Statement: `while` and/or `for` as well as *nested repetition*
5. Function: including *simple recursion*

Final Objective

Given a sequence of `character` representing a Mathematical expression, evaluate the Mathematical expression.

Assumptions

The following assumptions are considered to be true, they limit the inputs to the following restrictions:

- ▷ The sequence consists only of `0-9`, `+`, and `*` characters, with numbers are treated as single digit number
- ▷ The sequence is terminated by a single `=`
- ▷ The sequence is a valid Mathematical expression written in Polish notation

Tasks

The problem is split into 3 tasks with 4 number of testcases given. In the sample run, please note the following:

- ↩ is the *invisible* `[newline]` character.
- User input in blue and program output in purple color.
- Comments are in green color and are not part of the input and/or output.
- If the test(s) give(s) **NO** message(s), it means your program is correct.

Task 1/3: ***[Input/Output]*** **[10%]**

Write a program to read the sequence of `character` and print it back without the `=` sign. Note the `[newline]` on the output.

Sample Run:

Inputs:	Outputs:
`++4*578=`	`++4*578↩`

Save your program in the file named `polish1.c`. No submission is necessary.

Test your program using the following command(s):

```
./a.out < test1.in | diff - test1_1.out
./a.out < test2.in | diff - test1_2.out
./a.out < test3.in | diff - test1_3.out
./a.out < test4.in | diff - test1_4.out
```

To proceed to the next task (*e.g., task 2*), copy your program using the following command:

```
cp polish1.c polish2.c
```

Task 2/3: ***[Rewriting]*** **[75%]**

Write a program to read the sequence of `character` and print it back without the `=` sign in an *infix* notation with added brackets. Note the [`newline`] on the output.

Hint:

- Consider an expression `E` and a number `n`
- `eval("+EE") := (eval("E")+eval("E"))`
- `eval("*EE") := (eval("E")*eval("E"))`
- `eval("n") := n`

Sample Run:

Inputs:	Outputs:
`++4*578=`	`((4+(5*7))+8)↩`

Save your program in the file named `polish2.c`. No submission is necessary.

Test your program using the following command(s):

```
./a.out < test1.in | diff - test2_1.out
./a.out < test2.in | diff - test2_2.out
./a.out < test3.in | diff - test2_3.out
./a.out < test4.in | diff - test2_4.out
```

To proceed to the next task (*e.g., task 3*), copy your program using the following command:

```
cp polish2.c polish3.c
```

Task 3/3: ***[Evaluation]*** **[100%]**

Write a program to read the sequence of `character` and print it back **with** the `=` sign in an *infix* notation with added brackets followed by the result of evaluating the expression. Note the [`newline`] on the output.

Hint:

- Consider an expression `E` and a number `n`
- Consider the algorithm from previous task:
  - Let the return value of `eval` is an `integer`
  - Then you can evaluate the final expression

Sample Run:

Inputs:	Outputs:
`++4*578=`	`((4+(5*7))+8)=47↩`

Save your program in the file named `polish3.c`. No submission is necessary.

Test your program using the following command:

```
./a.out < test1.in | diff - test3_1.out
./a.out < test2.in | diff - test3_2.out
./a.out < test3.in | diff - test3_3.out
./a.out < test4.in | diff - test3_4.out
```

Useful VIM and SSH Terminal Commands

- **VIM Mode Switch:**
  - **i** i nsert (*from* Command)
  - **esc** esc ape to Command
- **Basic VIM Commands:** [mode=Command]
  - **:w** w rite file
  - **:q** q uit file
  - **:q!** q uit file (*forced*: *without saving*)
  - **:wq** w rite and q uit
- **Advanced VIM Commands:** [mode=Command]
  - **/text** find text
  - **n** find n ext text
  - **shift + n** find previous text
  - **gg=G** auto-indentation all lines
- **VIM Text Edit Commands:** [mode=Command]
  - **dd** d elete line at cursor (*cut*)
  - **yy** y ank line at cursor (*copy*)
  - **p** p aste after current cursor
  - **u** u ndo one change
  - **x** cut one character at cursor
  - **:red** red o undone changes
  - *N* **dd** d elete N lines down (N is number)
  - *N* **yy** y ank N lines down (N is number)
- **VIM Auto-Completion:** [mode=Insert]
  - **ctrl + n** complete word
  - **ctrl + x** complete line
- **Basic SSH Terminal Commands:**
  - **cd dir** open folder dir
  - **cd ..** open *parent* folder
  - **rm file** remove file file
  - **rm -r dir** remove folder dir
  - **vim file** open file in VIM
  - **ls** list files in folder
  - **ls -all** list ALL files in folder
  - **cat file** open *small* text file
  - **less -e file** open *large* text file
  - **cp f1 f2** copy f1 to f2
  - **mv f1 f2** move f1 to f2

    (*in effect, rename if in same folder*)
- **Execute Your Program in SSH Terminal :**
  - **gcc -Wall file** compile file
  - **gcc -Wall -lm file**

    compile file with math library (i.e. #define <math.h>) included
  - **./a.out** run program
  - **gcc -Wall file -o f1**

    compile file and rename executable into f1 (run using ./f1)
- **Advanced Program Execution Commands in SSH Terminal :**
  - **./a.out < f_in**

    run program with input redirection from file located at f_in

    (e.g. **./a.out < test1.in**)
  - **./a.out < f_in > f_out**

    run program with input redirection from file located at f_in and redirect the output to write into (*non-existing*) file called f_out

    (e.g. **./a.out < test1.in > output1**)
  - **diff f1 f2**

    compares the two files (f1 compared with f2) line by line (*note: no news is good news*)

    (e.g. **diff output1 test1_1.out**)
  - **./a.out < f_in | diff - f_out**

    run program with input from f_in immediately compare output with f_out

    (e.g. **./a.out < test1.in | diff - test3_1.out**)
- **SSH Terminal Emergency Commands:**
  - *Infinite loop* press **ctrl + c**
  - *End input* press **ctrl + d**

  (*better way is to use input redirection*)
- **VIM DO NOT DO LIST**
  - **ctrl + z** move to background

    (if done, type **fg** into **SSH Terminal**)
  - **ctrl + s** suspend

    (if done, press ctrl+q)
  - *Close without using* **:q**
    - on reopen, .swp file created
    - open file, choose Recover & exit VIM
    - open file again & choose Delete
- **GCC DO NOT DO LIST**
  - **gcc file -o file**

    compile file and rename into file (now, file is no longer a C program file)
    - ***pray hard...***
    - look for .file.history by typing **ls -all**
    - copy to windows using **SSH File Transfer**
    - ***hope*** latest code is at *end of file*