Software Project – Assignment 2

Due to 29/11/2015 (23:55)

Introduction

In this assignment you will exercise your coding skills using C. You will implement a basic calculator which supports the 5 operations described in assignment 1 ('+', '-', '*', '/', '$'). During the assignment you will be asked to implement Stack data structure and test it to make sure it works properly. Next you will implement a command line program that implements the calculator using the stack you built.

Note: This assignment is not as easy as assignment 1, it requires more work hours than assignment 1 so please make sure you plan your schedule properly.

The Calculator

The calculator is a command line program which receives from the user a command. The command is either an arithmetic expression or a termination command. If the command is an expression the program will evaluate the expression and print the result. If the command is a termination command the program will exit.

Command Types

Arithmetical expression

$$[num_1] \blacksquare [num_2] \blacksquare \ldots \blacksquare [num_n]$$

Each $[num_i]$ is a non-negative integer (Unsigned) and $\blacksquare \in \{+, -, *, \ /\ , \$\ \}$. Please note that operands and operations are separated by at least one space character.

Examples for valid expressions:

```
1 + 3 $ 10 / 3 * 2
1 + 2 - 100 + 109 / 201 * 99
100       + 1323 $ 1324
```

Examples for invalid expressions:

```
+ 100 - 100
1+2
-100
1234 * b / c
(1 $ 2)
```

Recall from assignment 1 the operation is defined as follow:

$$a \$ b = \begin{Bmatrix} \text{not valid} & a > b \\ a + (a+1) + (a+2) + \cdots + (a + (b-a) - 1) + b & a \le b \end{Bmatrix}$$

That is, the operation is defined such that a$b is the sum of all integers in the set [a,b]. Note that the operation is not defined when a>b. For Example:

1$3 = 1 + 2 + 3 = 6
1$1 = 1
2$1 = not valid

The arithmetical expression will be evaluated by the following Rules:

Rule 1: Perform all $ operations, work from left to right.
*Rule 2: Perform all * and / operations, work from left to right.*
Rule 3: Perform all + and – operations, work from left to right.

Note that this defines precedency rules between operations.
$ is precedent to * and /.
* and / are precedent to + and -.

Examples for evaluating an arithmetic expressions:

1 + 2 = 3
1 + 2 * 3 = 6
1 - 2 + 2 $ 3 * 2 = 9
10 / 2 $ 3 / 2 + 2 * 3 $ 4 * 2 = 29

Termination Command

<>

Your calculator should run until either this command was received or the program was forced to terminate.

Output

On each <command> the program will print to the user a message depending on the command.

- If <Command> is a termination command, the program will print:
  "Exiting…"
  After printing the message the program will exit.
- If <Command> is invalid arithmetical expression the program will print
  "Invalid Expression!"
- If <Command> is valid arithmetical expression **but the result is not valid** the program will print
  "Invalid Result!"
  Please note, invalid result will be caused in case of division by zero or an invalid $ operation.
- If <Command> is valid arithmetical expression **and the result is valid** the program will print

"res = Value"
Where Value is the result value. The result will be stored in a double variable.

Example:

```
>> 5 + 2
>> res = 7.0000
>> 5 * 2 / 5
>> res = 2.0000
>> 5 $ 3
>> Invalid Result!
>> 10 + - 333
>> Invalid Expression!
>> <>
>> Exiting...
>>
```

Note: Each message will be followed by a new line!

Implementation

Simple Algorithm: Evaluating a valid expression

Here is a simple algorithm on how to evaluate an expression using two stacks one stack for the numbers and another stack for the operations. This algorithm assumes that the expression is valid and the result is valid too.

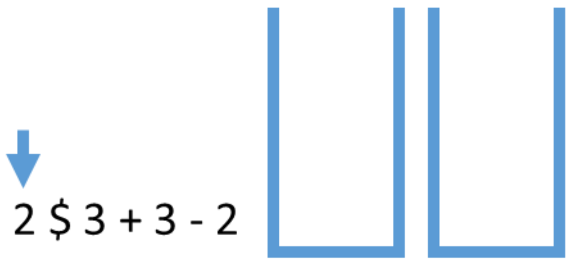
```
1. Split the expression into numbers and operations.
2. For each item in the expression (from left to right)
    a. If The item is a number push it into the numbers stack
    b. If The Item is an operation then
        i. while the condition (the operators stack is empty or the Item is
           precedent to the top operation in the operation stack) is NOT TRUE
           do:
            1. Pop an operation from the operation stack. Let ■ be this
               operation.
            2. Pop two numbers from the stack. Let X be the first number
               and Y the second number.
            3. Perform Y ■ X and push the result to the numbers stack.
        ii. Push the item into the operators stack.
3. While the operators stack is NOT EMPTY do
    a. Pop an operation from the operation stack. Let ■ be this operation.
    b. Pop two numbers from the stack. Let X be the first number and Y the second
       number.
    c. Perform Y ■ X and push the result to the numbers stack.
4. Return the number on the top of the numbers stack.
```

Note: This algorithm assumes the expression is valid and the result is valid as well. You will need to make sure your code runs properly when receiving invalid expressions or performing invalid operations.

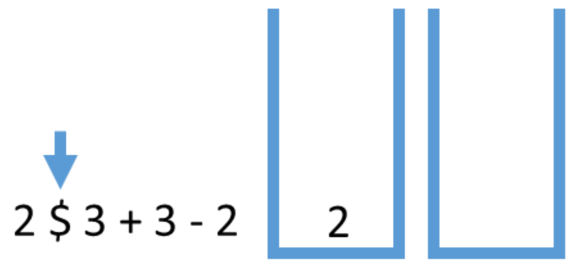
Example:

Let's see how the algorithm works when evaluating the expression:
2 $ 3 + 3 -2
Please note that the expression is valid and the result is valid too. We expect a result 6 since we first evaluate 2$3=5 and then the expression is equivalent to 5 + 3 – 2 = 6.

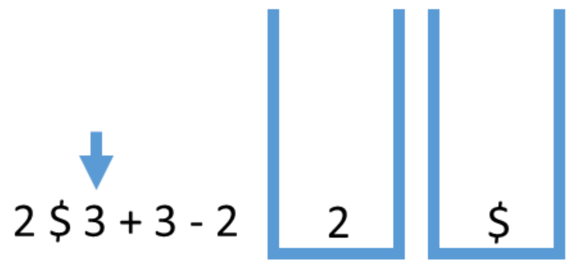
We start by splitting the expression and go through all items.



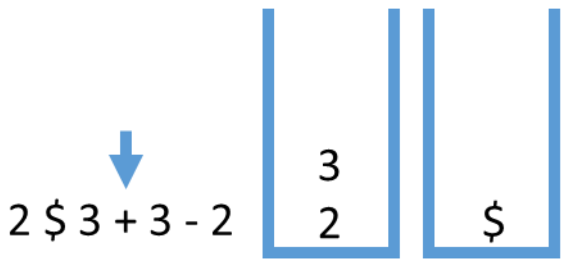
The first item is a number (the number 2) so it is pushed to the numbers stack. And we move to the next item in the expression.



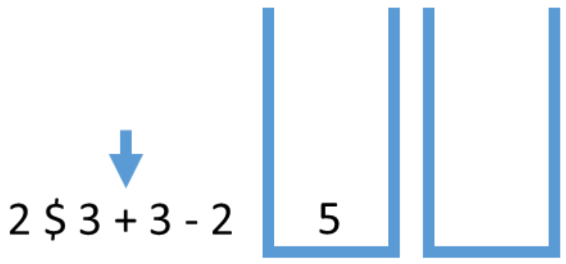
After the number 2 is pushed to the stack. We encounter an operator item, since the operators stack is empty we push the operator to the stack as well.



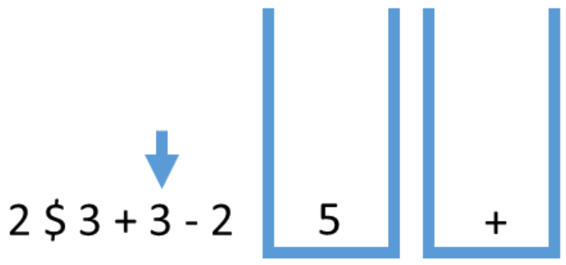
Next item is the number 3, so we push it to the numbers stack as well and we move to the next Item.



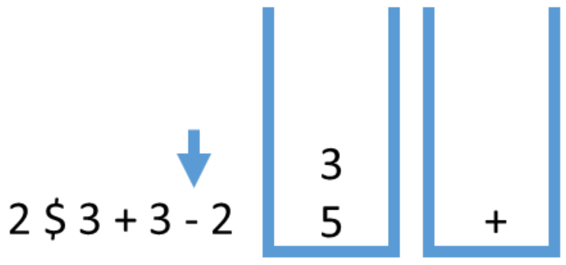
Next, we see that the next item is the plus operation. Since the $ operator is precedent to the + operator, we pop the $ operator from the operators stack and we Pop two numbers from the numbers stack. In the algorithm notation we have **X = 3, Y = 2 and ■ = $**. The result is stored in the top of the numbers stack as can be seen in the picture bellow: (2$3 = 5)



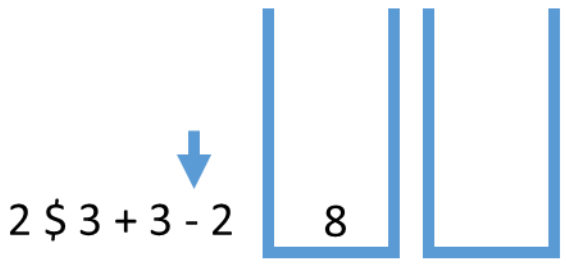
Now since the operators stack is empty, then we can push the '+' operator into the operation stack and move to the next item.



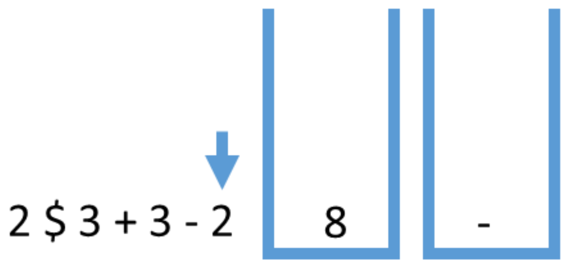
We push the number 3 to the numbers stack. Now the operation is '-' and it has the same precedency as '+', then we must perform the operations from left to right. That is we need to evaluate '+' first.



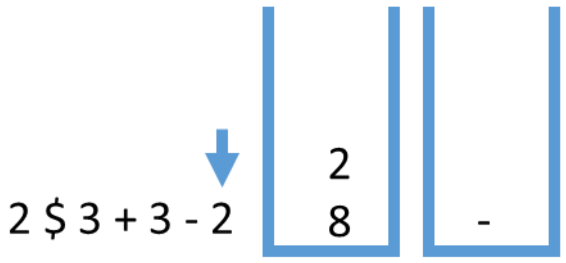
The + operation is performed the same as before and the result is stored again in the stack.



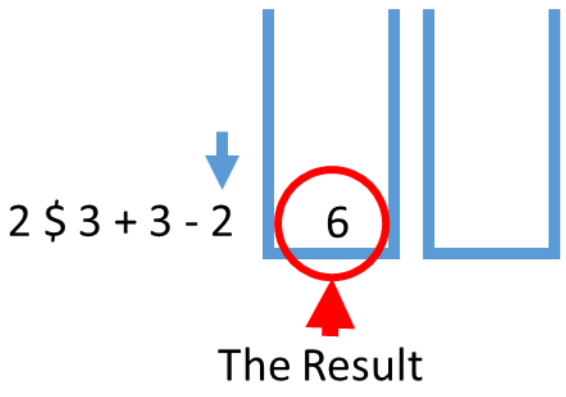
Now we can push the '-' operation since the operators stack is empty.



We push the number 2 to the numbers stack.



Now that we have gone through all the expression. We will continue evaluating the result the same as before until the operation stack is empty, which implies that we have our result, which is 6.



Now that we know how to evaluate a valid expression, we need write a C program which implements the calculator. Before doing so, we need to divide our problem into sub-problems. This principle is very important when designing a program, and it is called **Modular programming.** In general Modular programming is a design technique in which we separates the functionality of a program into independent and interchangeable modules. First you will need to implement the stack, which is used in the algorithm. Afterwards, you need to implement a simple parser which takes an expression and evaluates it by following the algorithms steps described above. Finally you will implement the main function which is responsible of control flow of the program.

SP_Stack

You are given a header file SP_Stack.h, this header file includes declaration of functions and structs that you will need to implement. You will need to implement the interface in a file called SP_Stack.c . Make sure your code runs as expected from the interface, read carefully the requirements of each function in the header file.

Stack struct

You have a freedom of choice on how to implement your stack, to hide your implementation of the stack we declared in the header file a new type called SP_STACK.

```
typedef struct sp_stack_struct SP_STACK;
```

To define what SP_STACK is, you need to define it in the source file SP_Stack.c. To do so you will need to define exactly what sp_stack_struct is. You can define the struct in SP_Stack.c as follow:

```
struct sp_stack_struct {
    //Your code goes here
};
```

You may assume that the stack size (Number of elements in the stack) at any time is no more than 1024.

Stack Element

We saw earlier our algorithm uses two stacks, one for numbers and the other is for operators. C doesn't support templates and there are several ways we can bypass this problem. In this assignment we will declare a new element type. This element type will be used to represent both a number and a sign.

Our stack will hold elements of type SP_STACK_ELEMENT. Each element has a type, the type of an element is represented by SP_STACK_ELEMENT_TYPE. Plus each element has a value, which is a double variable, if the element type is NUMBER then value holds this number, otherwise value is not valid. Please review the header file to see the definition of SP_STACK_ELEMENT.

Messaging Mechanism

Unlike java the exception mechanism is not supported in C, which sometimes can be used to inform the user if an error occurred. We will bypass this problem by defining a new type called SP_STACK_MSG. Each function receives a pointer of type SP_STACK_MSG* and the status of the operation will be stored in the memory location stored in this pointer. If this pointer is set to NULL then the status of the operation will not be set.

Basic function to manipulate a stack

You need to implement the basic functions of a stack (push, pop, top, size, create and destroy). You can find a detailed description of each function in the header file (SP_Stack.h). Make sure you covered all the requirements in the header file.

Testing SP_Stack

Before using SP_Stack, you need to make sure you have implemented the stack properly and as expected from the interface. In programming principles, this is called Unit testing. You are provided with a basic Unit test for your stack implementation (SP_Stack_UnitTest.c), please review it and make sure you understand how to use SP_Stack. This unit test doesn't cover all cases and you need to make sure to extend this so you are 100% that your implementation works as expected. You need to take the following considerations when testing your implementation:

1- **Memory Leaks** – Use valgrind (More on valgrind can be found on moodle) to check that all the memory resources which was dynamically allocated is freed. This is very important because if you have memory leaks your final grade will be affected.
2- **Functionality** – Make sure all the functionalities of the stack is well behaved and that you get the expected result.
3- **Edge Cases –** Make sure your code works properly in edge cases (such as what happens when the stack empty or when receiving a NULL pointer.

Remarks

- You can assume that at any time the stack size (number of elements in the stack) will not exceed 1024. That is **size(Stack) ≤ 1024**.
- You need to extend SP_Stack_UnitTest.c taking under consideration all the points pointed out before. Your test will be checked manually as well as automatically.
- Please do not change SP_Stack.h. Make sure all your code will be in SP_Stack.c.
- Time complexity: your stack should satisfy the time complexity of a regular stack. That is :
    - spStackCreate(): $O(1)$
    - spStackTop(): $O(1)$
    - spStackPop(): $O(1)$
    - spStackPush(): $O(1)$
    - spStackIsEmpty():$O(1)$
    - spStackDestroy(): $O(n)$

SP_Aux and main

In this section, after you implemented the stack, you will implement the algorithm presented earlier. Make sure your main function is implemented in main.c and all auxiliary functions you define will be declared in SP_Aux.h and implemented in SP_Aux.c.

First you need to receive from the user commands, to do so you can use the function fgets(). This is a simple main function which reads a line from stdin (The standard input channel) and stores the line in a string called 'line'. You may assume that **MAX_LINE_LENGTH = 200**.

```
int main(){
    char* line = malloc(MAX_LINE_LENGTH + 1);
    while(fgets(line,MAX_LINE_LENGTH,stdin)!=NULL){
        // Your Code goes here!
    }
    free(line);
    return 0;
}
```

After that you will need to parse the commands and interpret them as described earlier (Check if command is a valid expression, has a valid result or a termination command). We advise you to use the following function to implement the rest of the calculator:

1- atio() – This function converts a string into an integer.
2- strtok() – This function tokenizes your string given a delimiters.
   You can use " \t\r\n" as your delimiter.
3- strcmp() – This function compares two strings.
4- strcp() – This function can be used to copy strings.

Again you are expected to use the modularity principle in this part as well, you are expected to implement all the auxiliary functions you will use in SP_Aux.c and the main function should be in main.c

Makefile (Self Reading)

In assignment1 you saw how to use makefiles. In this section we will explain how the makefile for this project was built. You are not asked to change the makefile in this assignment file, however you will be asked to make your own makefile in next assignments. This section will be a good practice for you to understand how makefiles are built.

Let us review our project:

main.c : The source code which implements the main function. We use SP_Aux.h so we could use the functions declared in that file (The implementation however is in SP_Aux.c).

SP_Aux.h: This file contains all declaration of the auxiliary function that will be used in the main function. We use functions and declaration which are in SP_Stack.h.

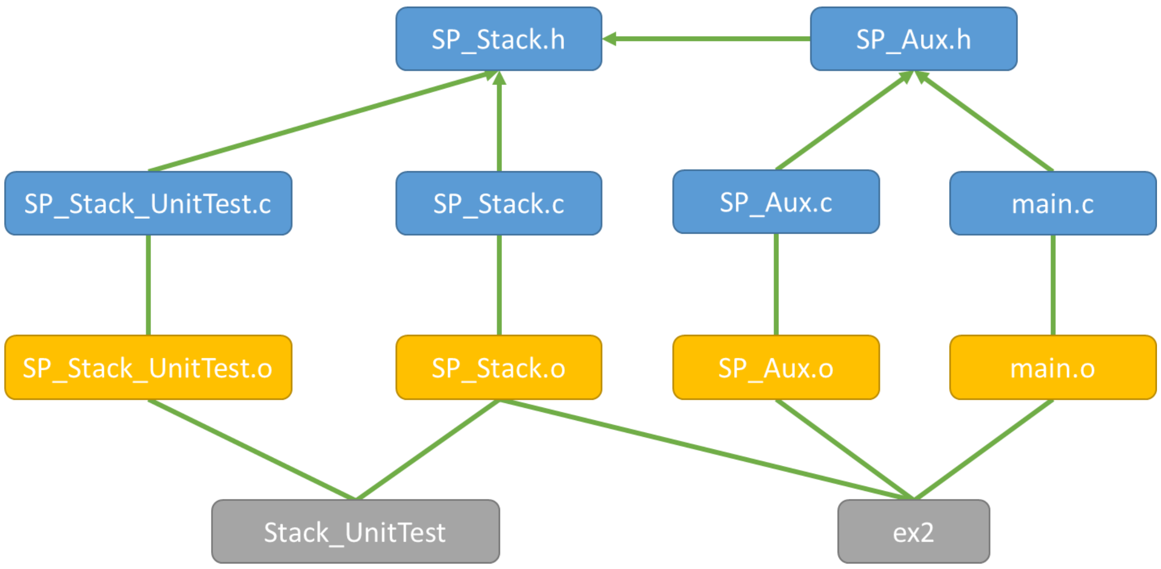
SP_Aux.c: This source code implements all functions in SP_Aux.h.

SP_Stack.h: This file includes all declaration of functions and types which used to represent a stack.

SP_Stack.c: This source code implements the stack and all functions which manipulates the stack.

SP_Stack_UnitTest.c: This source code test your stack implementation.

A simple dependency graph is for our project is shown below:



An arrow between two files implies a dependency, if one file changed then we need to recompile the other file.

First we need to make rules which build all object files (*.o) of our project. For example, the object file main.o depends on main.c, SP_Aux.h and SP_Stack.h, if any of these files changes then we need to recompile the source file main.c. The rule would be as follow:

```
main.o: main.c SP_Aux.h SP_Stack.h
    gcc -std=c99 -Wall -Werror -pedantic-errors -c main.c SP_Aux.c
```

After making all the rules for the object files. We need to be able to link these object files and create an executable program. In our project we have two programs, ex2 which is the calculator and Stack_UnitTest which is an executable unit test which checks the functionality of our stack.

For example, in order to make the program ex2 we see that it depends on main.o,SP_Aux.o and SP_Stack.o (SP_Stack.o will contain the implementation of all functions). Our rule for ex2 will be as follow:

```
ex2: main.o SP_Aux.o SP_Stack.o
    gcc -std=c99 -Wall -Werror -pedantic-errors  main.o SP_Aux.o SP_Stack.o -o ex2
```

Remember to use make you need to use the following command:

```
>> make [-f filename] [target]
```

Options:

- [-f filename]: this is optional flag, if specified, make will use 'filename' as a makefile. If not specified then make will use either 'makefile' or 'Makfile' as a filename.
- target: This is optional flag, if specified, make will do the rule 'target'. If not specified make will do the first rule in the makefile file.

Submission

Please submit a zip file named **id1_id2_assignment2.zip** where id1 and id2 are the ids of the partners. The zipped file must contain the following files:

- main.c , SP_Aux.c, SP_Aux.h – Your implementation of the parser and the algorithm.
- SP_Stack.c , SP_Stack.h – Your implementation for the stack.
- SP_Stack_UnitTest.c – A unit test for SP_Stack. Make sure you cover all the points pointed in the section "Testing SP_Stack".
- partners.txt – This file must contain the full name, id and moodle username for both partners. Please follow the pattern in the assignment files. **(do not change the pattern)**
- makefile – the makefile provided in the assignment files.

Notes:

- **Both students** are asked to submit the assignment.
- You can back up your submission on Nova.

Remarks

- For any question regarding the assignment, please don't hesitate to contact Moab Arar by mail: moabarar@mail.tau.ac.il.
- Late submission is not acceptable unless you have the lecturer approval.
- Cheating is not acceptable.

Good Luck