COL216 Assignment-2

Dhruv Kumar Gupta 2019CS10346
Swaraj Gawande 2019CS10406

March 2021

1 Design Overview

1.1 Input and Output

The Program when run, asks for a postfix expression.

The program waits for the string and stops accepting either when a line feed('\n') is entered or length of string becomes 127.

127 is a design choice as last character is always **null** which takes 1 byte, and hence 128 is the size allocated for input string, which is reasonable as the input will be entered manually and 127 characters is more than enough.

After the input of the string, the program returns the calculated value of the given postfix expression.

Example screen :

```
Enter the postfix expression string:
325*+

The calculated value of postfix expression is : 13
```

1.2 Error Handling

There are two kinds of error which could occur :

- Input character does not belong to 0-9 or * or + or -
- The expression, even though contains valid characters but does not constitute a valid postfix expression, like : 9+

These errors are properly handled before popping or evaluating.

1.3 Ambiguity

The problem statement of the assignment does not specify whether the operators '+' and '-' can be used as unary operators or not, but considering that '+' and '-' may be overloaded, leads to ambiguous postfix expressions which can mean 2 different infix expressions which evaluate to different values.

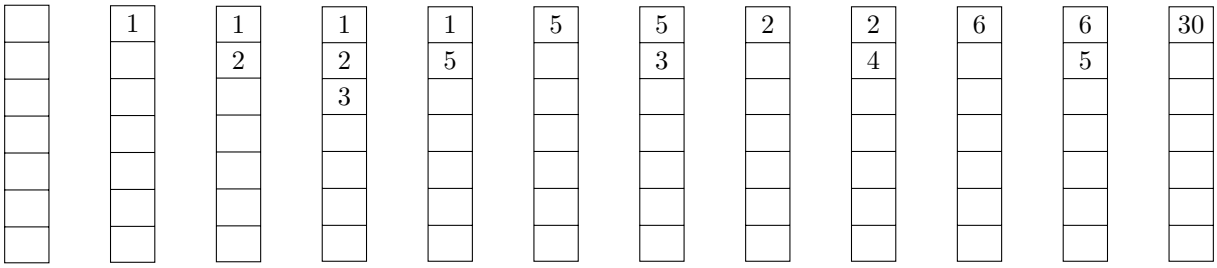
2 Explanation of approach and code

2.1 Algorithm description

The algorithm described pushes the digits, that is, 0-9 in the stack and when an operator is encountered, the last 2 values inside the stack are popped, evaluated and then pushed, thus decreasing size by 1.

To illustrate the algorithm, Consider the following postfix expression and the stack stages after reading each character :

$$\mathbf{123} + *\mathbf{3}\text{–}\mathbf{4} + \mathbf{5}*$$



The stacks have been shown inverted due to MIPS internal structure where the head of stack moves down in memory address, thus indicating that the bottom of the stack is at higher memory address than the top of the stack.

The above algorithm assumes that the operators '+' and '-' stand for binary addition and subtraction operator only and not the unary sign operators. If the operators are considered overloaded with both operations, then a single postfix expression can stand for 2 different infix expressions which leads to ambiguity[1]. This ambiguity is illustrated in the following console.

```
D:\IITD\COL216>in_to_post_converter.o
enter expression
3-(-2)
32--
D:\IITD\COL216>in_to_post_converter.o
enter expression
-(3-2)
32--
```

Two different infix expressions evaluating to different values, $3-(-2)$ and $-(3-2)$ have the same postfix expression $32--$, and hence, the symbols '+' and '-' have not been considered as overloaded. Thus, to use -2 or any negated digit as a value in postfix, one must use $02-$ as it's postfix expression, to ensure non ambiguous postfix expressions.

[1]This is according to conversion from infix to postfix by the given cpp code

2.2 Code explanation

The program first asks for the complete postfix expression as input and then store it in memory at location **input** which is an 128 byte space. The register **s0** contains the address of the character of the string on which it will be working on in the loop(**s0** is initially set to **input**), and **t1** contains the size of stack which is useful for some checks done later.

The code has a loop which considers each character in the string one at a time and does the respective task according to what the character is. A valid character can be a number **(0-9)**, addition operator +, subtraction operator **-** or multiplication operator *****, with a termination for a string which can be line feed LF or null if the string has length 128. Any other character would be invalid for a postfix expression and if detected, an error message is printed and program terminates.

Now, if the character is a number then the program stores its actual value as an integer[2] in register **s2** and then pushes it into the stack. If the character is an operator then the program checks if it can pop two numbers from the stack that is if the size of the stack is greater than or equal to two. If this is not the case, then the postfix expression provided must be invalid and an appropriate error message is printed.

On the other hand, if the stack size is greater than or equal to two, then it pops two numbers and store them in register **s3** and **s4**, performs the binary operation corresponding to the operator we have in register **s2** and stores the result in **s3** which is then pushed into the stack.

After traversing through the whole input string, the number of elements in the stack should be 1 for a valid postfix expression. 0 or 2 or more numbers indicate that the expression must not be a valid postfix expression. The console displaying several kinds of errors :

```
Enter the postfix expression string:
34* 67+*

Error : Invalid character input at position : 4
Enter the postfix expression string:
p23++

Error : Invalid character input at position : 1
Enter the postfix expression string:
3+

Error : Invalid postfix expression

Enter the postfix expression string:
34++

Error : Invalid postfix expression

Enter the postfix expression string:
345+

Error : Invalid postfix expression
```

[2]Since '0' as character has ascii value 48, we subtract 48 from the the character, thus converting digit character to value of digit as integer

3 Testing

The testing was done component wise like Assignment 1.

The loop in the program and character reads were first tested to ensure it runs until the end of the string. We created a simple program which would print '$' after each character, thus ensuring that each character is traversed one by one.

Next, the recognition of each character based on their ascii value was ensured, by creating branches and printing different character alongside it based on the string character. Since only, **0-9**, +, **-**, ***** are allowed, the program specifically checks equality with them and uses **n** and **null** as end characters. Several other characters like whitespace, tab, alphabets were entered to ensure the program displays the error.

All the possible flows of the program were possible were tested including the ones which lead to an error. Additions, subtractions and multiplication were first tested individually and then their combinations in expressions. The condition that these operations give an error when the stack size size is less than two was also tested in each case.

We also tested the program for empty input strings and made some changes to print appropriate message when provided with empty string as input.

Input strings of length 128 were also used for testing to insure that the program never takes a string with length more than 128.

Some test cases were also used to test the condition that the final stack size needs to be one for the input string to be a valid postfix expression.

3.1 Some Testcases which were used

- 123+
- 123 + +
- 234 ∗ 1 ∗ +
- (empty)
- 12−
- 032 − −
- 111 . . . (64 times)+++ . . . (63 times)
- 1 2∗ (There's a space)
- a2+

```
Enter the postfix expression string:
123+

Error : Invalid postfix expression

Enter the postfix expression string:
123++

The calculated value of postfix expression is : 6
Enter the postfix expression string:
234*1*+

The calculated value of postfix expression is : 14
Enter the postfix expression string:


Error : Empty string input

Enter the postfix expression string:
12-

The calculated value of postfix expression is : -1
Enter the postfix expression string:
032--

The calculated value of postfix expression is : -1
Enter the postfix expression string:
1 2*

Error : Invalid character input at position : 2
Enter the postfix expression string:
a2+

Error : Invalid character input at position : 1
```