INDIAN INSTITUTE OF TECHNOLOGY DELHI

COL216

Report

Assignment – 1

Postfix Evaluator in MIPS

Abstract

This program evaluates a postfix expression of digits using a stack based algorithm. This has been written in MIPS assembly language to better understand the hardware abstractions.

Harsh Agrawal 2019CS10431

Saptarshi Dasgupta 2019CS50447

Contents

1 Approach

We implemented a stack based algorithm to evaluate the input postfix expression. The algorithm is as follows

1.1 Algorithm

The algorithm iterates over the input string character by character

1. When the current character is an operand $(0-9)$ the character is pushed to the program stack.
2. When the current character is an operator $(+|-|*)$ the top two entries in the stack are popped and the corresponding operation is performed. The result of the operation is pushed back to the stack.
3. When the current character is either a newline $(\backslash n)$ or the null character $(\backslash x00)$ the topmost entry is popped from the stack and printed.

1.2 Efficiency

The above algorithm iterates over the input string only once and in a single pass is able to evaluate the expression. Hence the asymptotic time complexity of the algorithm is $O(n)$ where n is the length of the input string.

2 Testing

We employed both automated and manual modes of testing. The automated mode helped establish the correctness whereas the manual mode helped handle the corner cases.

2.1 Automated

We wrote a `python3` script that generates random postfix expressions, executes the asm file in a MIPS emulator (`spim`), communicates with the executed process through unix pipes, and compares the output with the expected value. To install the dependencies, enter the following commands once inside root project folder.

```
$ sudo apt install spim
$ python3 -m pip install -r requirements.txt
$ python3 tester.py -n 20 -k 30
```

The last command will run 20 test cases with 30 operands in each test case and print a test log with all the test cases generated and the result of each test case.

2.2 Manual

We have provided the functionality to run custom test cases so that corner cases may be tested for. We can either write the test cases into a file or provide the values on `STDIN`.

```
$ python3 tester.py -i <testcase_file_path>
```

This will read the test cases from input file. If test cases are to be provided on `STDIN`, use the below command.

```
$ python3 tester.py --manual
```

2.3 Results

The test case logs are provided in the `./tests` directory. This includes both the automatically generated test cases as well as the manual test cases. Based on the logs we conclude the following:

2.3.1 Automated

The execution logs for these test cases are stored in the `./tests/automated` directory.

1. When the number of operands were less, most of the test cases passed.
2. All the test cases that failed were those which included integer overflows. The result of the postfix expression exeeded the integer bounds of a 32 bit register.

2.3.2 Manual

The execution logs for these test cases are stored in the `./tests/manual` directory.

1. `extra_operands`: The postfix expression had more operands making it an invalid postfix expression. In this test case our code raised a exception.

2. `extra_operators`: The postfix expression had more operators than operands to operate upon. In this case our program raised an exception.

3. `invalid_operand`: The operand was not a digit and our program raised an exception since it cannnot operate on a non-numeric input.

4. `invalid_operator`: The operators are restricted to $(+|-|*)$. Any other character is flagged as an illegal character and an exception is raised.

5. `invalid_postfix`: The postfix expression is invalid, in such case our program generated an appropriate exception.

6. `empty_input`: The input string was empty, in this case our program generated the output 0.

7. `overflow`: These test cases explicitly introduce arithmetic overflow in computations. Our program generates an arithmetic overflow exception for such cases.

```
❯ python3 tester.py -n5
[*] Executing test case - 256-23+276-1-++++
[+] Test Passed
[*] Expected value: 8, Computed value: 8
    ================================================================================================
[*] Executing test case - 12*8+84*3*2-08**+
[+] Test Passed
[*] Expected value: 10, Computed value: 10
    ================================================================================================
[*] Executing test case - 21*549*056*++6**-
[+] Test Passed
[*] Expected value: -1978, Computed value: -1978
    ================================================================================================
[*] Executing test case - 251124*-166+*+**+
[+] Test Passed
[*] Expected value: 27, Computed value: 27
    ================================================================================================
[*] Executing test case - 53+88*5-+9*8-66++
[+] Test Passed
[*] Expected value: 607, Computed value: 607
    ================================================================================================
[*] Passed 5/5 test cases!
```

Figure 1: Automated Testing example

2.3.3 Efficiency

We tested the efficiency of our program execution. We plotted the above graph with execution time on the y axis and the length of the input string on the axis. We found that the graph came out to be a straight line, indicating that the time complexity of the procedure is $O(n)$

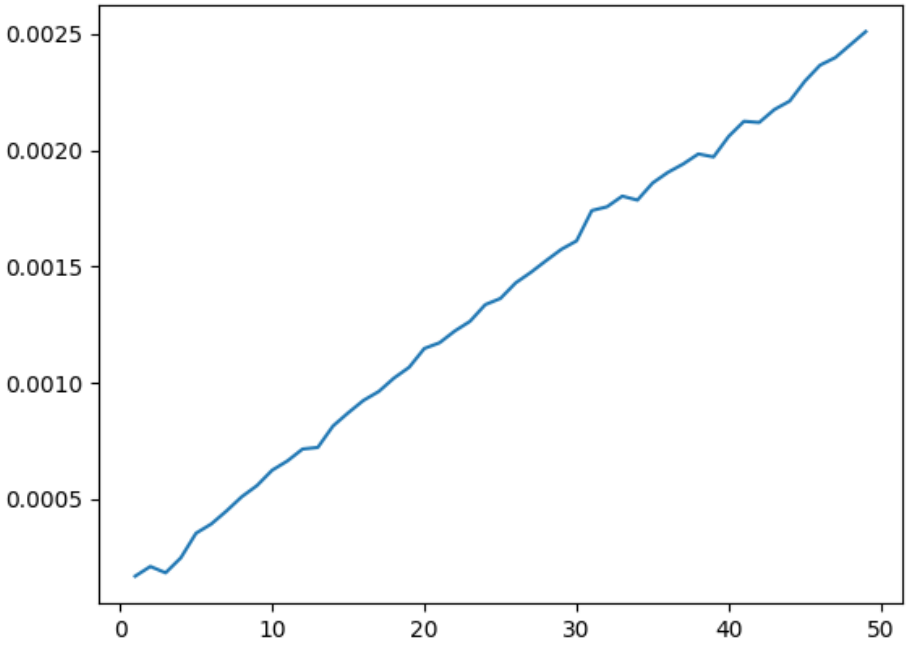

Figure 2: Graph of Execution time vs. Number of Operands

3 Assumptions

- The size of input string is capped at `8 KB`, which is equivalent to 8192 characters. This design decision was made to ensure that the input string does not cause a stack overflow.
- We have assumed that the digits are one of $(0-9)$ and the operands are

$(+|-|*)$, any other operator or operand is flagged as illegal character and appropriate exceptions are raised.

- We make the assumption that the result of the postfix expression will not lead to an integer overflow. This means our program is capable of evaluating only those expressions that are within the bounds $[-2^{31}, 2^{31} - 1]$
- Lastly we assume that the logic for evaluating the postfix expression in tester script is correct since it is fairly easy to code it in a high level programming language like python than a low level programming language like assembly.