HOUSE TRIP®

Stack Machine

Abdullah Ali

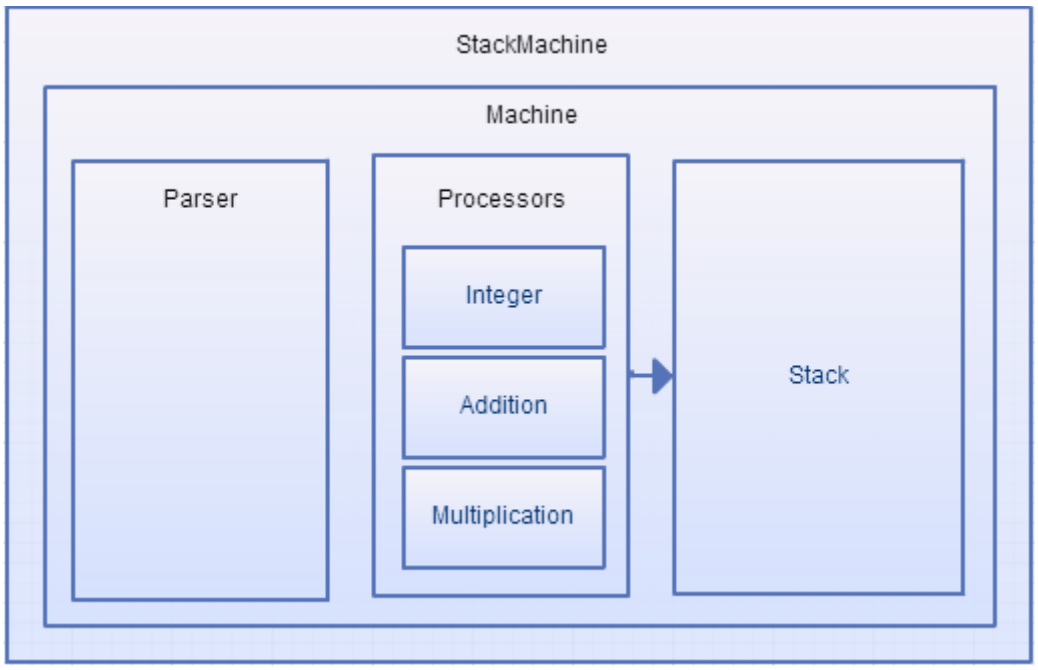
1 INTRODUCTION

The aim of this project was to create a Stack Machine, which, performs addition and multiplication on integers placed on the stack. I have used this document to express how I went about solving this task, my inspirations and thinking behind my decisions.

2 APPROACH

2.1 SYSTEM INTERACTION

To aid the visualisation of the code, the following diagram shows the class interactions/structure.



2.2 CLASSES/MODULES

2.2.1 PARSER

The Parser module is perhaps the heart of the stack machine application. Upon user input, this module's functions are called upon to ensure that the input is valid. I have decided to use this approach, since in our idiom; we need the input to conform to a specified syntax, that is:

a) The input must not be empty
b) The input must start with an integer
c) The input only contains valid characters (0-9, +, *)
d) The input contains more integers than operators (this ensures that the operators have enough integer characters to operate on)

Although this class is the heart of the application, it only has a few functions, since does not have any other responsibilities.

The 'valid_input?' method is the main method which carries out the parsing of the input. It is fairy efficient since it runs in O(n) for both memory and speed.

This method works by firstly ensuing that the input is not empty is matched first before carrying out any further processing. Following that, the input is progressively stepped through to ensure it conforms to all of our requirements. You can note that the series of checks have been placed in order, from cheapest, to most expensive – allowing to maximum efficiency.

The 'starts_with_integer?' function uses inclusion checks on the first character of the input instead of a regex. A typical regex approach would have been to stipulate that the first character conforms to [0-9] or \d. However, I have decided to use an array inclusion check since I felt that it is more readable and clear with its intent.

2.2.2 PROCESSORS

Initially, I used a dynamic function, which interacted from the Machine directly to the Stack. However, after doing some online research (similar solutions); I felt that the best way to create separate processor classes. By doing this, we can easily and quickly add more processor in the future (should we wish to).

2.2.2.1 Integer Processor

Delegator class – passes messages from Machine to stack

2.2.2.2 Addition Processor

Delegator class – passes messages from Machine to stack

2.2.2.3 Multiplication Processor

Delegator class – passes messages from Machine to stack

2.2.3 STACK

The Stack class is very simply. It is an Array class, which pushes integers on the stack, adds them and multiplies them.

2.2.4 STACKMACHINE::MACHINE

This class can be thought of as the "Controller" of this application. Upon the user triggering the stack_machine_emulator function, the initial step is to validate the input. I have thought about letting the input trigger all the functions first, then validating it second (i.e.: when rails 3 used to allow invalid parameters pass the controller and only get validated once it reaches the model), however, I though the approaching it in such a way, would mean that the Stack class and the Processor classes would need to have extra code to protect against all types of errors (since we would not have assumptions). However, on second though, I settled for a similar approach as in rails 4 (strong parameters) – where we ensure that all the input is as we want it, and only then do we trickle the input down to the rest of the application.

The processor for this class works by matching characters to a predefined map, which is used to trigger the correct processor class.

3 PERFORMANCE

3.1 TIME

Expected worse-case time complexity is O(n)

3.2 SPACE

Expected worse-case space complexity is O(n)

4 TESTS

All the codes has been tested (TDD where possible). Please refer to the /spec folder to see these tests.

Note: Although the logic of private methods has not been tested directly, it has been tested via its intended flows.

5 ALTERNATIVE SOLUTIONS

An alternative solution which I considered was to have one function, which carries out the entire task. In essence, this function would have a case statement (on the operators), which will then pop items from the stack (ensuring there are enough items), and then carries out the operation on those items. However, I felt that this approach would be much less maintainable, since it would contain a lot of if-else statements, which would obfuscate the intention of the code.

6 WHAT I WOULD DO DIFFERENTLY

If I had complete control over the requirements, I would perhaps do the following differently:

a) Do not return -1 for all types of errors. In the past, I used an erLang approach to error reporting, where by the error is returned in a 2 item array, in the following format: [ERROR_CODE, ERROR_MESSAGE]

   By doing so, we will be able to return the error code (to be used in the application logic) and an error message for the user, as follows:
   [-1, “Input cannot be empty”]

b) Test private methods. This may be controversial amongst Ruby developers, since it is believed that all private methods should not be tested explicitly, but instead more of a “flow” test/ behaviour test. Whilst I agree is some cases, I feel that sometimes it is very important to test private methods, since they may have very critical logic which must be correct.

7 FURTHER WORK

Although the application achieves what it is set out to do, there is room for improvement with regards to the efficiently of the methods. Storing the input as well as the stack (both on memory at some points) can be expensive, if the input was very long. However, this approach is convenient and is simple enough for the requirements of this task.

I hope that my chosen ideas and my thinking behind my designs are sensible in most cases – however, I am always open to learning new ways of making my code work better, faster and more reliable. Hence, I welcome your feedback.