Victoria University of Wellington
School of Engineering and Computer Science

SWEN431: Advanced Programming Languages
Assignment 1: Ruby Stack Deluxe

Due: 3 April 2025, before midnight

1 Outline

You are to implement a "Stack Deluxe" interpreter. A Stack Deluxe interpreter supports operators such as "+" & "*" and operands such as "1" & "431". See below, and in particular section "Code Completion Stages", for further operators and operands.

The input for the interpreter is a sequence of elements that need to be pushed on a stack and are evaluated accordingly. Operands such as integers are simply added to the stack, i.e., do not imply any evaluation. So feeding two numbers "1" and "2" to a Deluxe stack results in a stack containing the elements "1 2" with "1" being at the bottom of the stack and "2" being at the top of the stack. A binary operator like "+", however, when encountered in the input sequence, will consume the top two elements on the stack to calculate a result and then push that result on to the stack. For instance, the input sequence "1 2 +" will lead to a stack containing the element "3".

Input sequences are provided as text files and, in their simplest form, contain one element per line. For instance:

ARITHMETIC I

INPUT
```
1
2
+
```
OUTPUT
```
3
```

Binary operators are evaluated as if they were placed between the top two elements of a stack, so the input sequence "3 2 -" results in a stack containing only element "1". A slightly advanced input format allows input sequence elements to be placed on the same line in a file:

ARITHMETIC II

INPUT
```
1 2 + 3 *
```
OUTPUT
```
9
```

Yet another example is

ARITHMETIC III

INPUT `1 2 3 9 - + 2 * + 3 *`

OUTPUT `-21`

Some commands rearrange elements on the stack, e.g., SWAP swaps the position of the two top elements:

SWAP EXAMPLE

INPUT `1 2 3 SWAP`

OUTPUT `1 3 2`

Note that the SWAP command is immediately consumed and that the result of processing an input sequence can be a stack containing any number of elements.

DROP removes the top element while DUP (short for *duplicate*) pushes the top element on the stack.

DROP/DUP EXAMPLE

INPUT `1 2 3 DROP DUP`

OUTPUT `1 2 2`

The ROT command rotates/cycles the top three elements as follows:

ROT EXAMPLE

INPUT `0 1 2 3 ROT`

OUTPUT `0 2 3 1`

ROLL performs the same rotation, but for a variable number of elements. The top element specifies the number of elements to rotate.

ROLL EXAMPLE

INPUT `0 1 2 3 4 4 ROLL`

OUTPUT `0 2 3 4 1`

ROLLD rotates a variable number of stack elements in the opposite direction.

ROLLD EXAMPLE

INPUT `0 1 2 3 4 4 ROLLD`

OUTPUT `0 4 1 2 3`

IFELSE requires a `false` or `true` element at the top of the stack and either leaves the second or the third element from the top of the stack removing the irrelevant element. For instance:

IFELSE EXAMPLE

INPUT `0 -9 9 1 1 == IFELSE`

OUTPUT `0 -9`

Further input sequences with their expected result stacks are provided in the accompanying material.

2 Code Completion Stages

The code mark contributes 40% of the overall mark for this assignment, while the report contributes the remaining 60%.

Your interpreter implementation must be contained in a single `ws.rb` file, i.e., must not make use any other files defined by you. All your definitions must be in that one file. When invoked as

```
ws.rb input-xyz.txt
```

where `xyz` refer to three digits, the program must create a file `output-xyz.txt` in the same (current) directory and populate it with the stack contents – one stack element per line – resulting from processing the input feed in `input-xyz.txt`.

Your code may import ("require") standard libraries as long as running `ruby ws.rb <input-file>` works on our ECS machines. In other words, you must not depend on libraries which are not available on our ECS machines.

Hint: *You may use regular substring extraction to obtain the three* `xyz` *digits of the input file name, or use Ruby's regular expression matching feature; there will always be exactly three digits at the same position.*

Hint: *Remember that an argument provided to a Ruby program invocation can be accessed via* `ARGV[0]`.

Hint: *While you are developing, you will probably want to print/render the result stack to the standard output to save you from looking into the correct file every time you do a test run. You need to ensure, though, that the submitted implementation writes the result stack into the correct output file, as specified above.* **Your submitted version must not print anything to the standard output, not even status messages, or similar.**

Your implementation needs to always properly close the written file, should always terminate, and must not abort with any tracebacks. If necessary, use a generic "begin rescue end" construct around your main processing invocation.

It is of paramount importance to comply to all specifications made in this assignment description since even minor deviations may lead to the loss of a significant amount of marks or even all marks. We will be unable to investigate the causes of any deviations, let alone address them. For instance, if your implementation always produces the correct output to `output.txt` rather than `output-xyz.txt` where `xyz` refer to the three digits of the respective input file, we will not be able to award marks for the code contribution of this assignment, regardless of whether the contents of the `output.txt` files are correct or not.

2.1 Core (55%)

A code grade of up to a "C" can be achieved by supporting

- integer operands, and basic integer operators ("+", "*", "-", "/").
- multiple operands and operators on a single line.
- more operators: exponentiation ("**") and modulo ("%").
- stack commands "DROP", "DUP", and "SWAP".
- a mixed input format (multiple lines, each line with potentially multiple elements).

Hint: *Processing multiple elements on a single input line input will become more difficult later on, once more complex elements are added. However, marks will still be awarded for functionality that only works with multiline input.*

2.2 Completion (15%)

For a better code grade, up to a "B", your solution is required to additionally support

- stack commands "ROT", "ROLL", and "ROLLD".
- comparison operators ("==", "`!=`", ">", "<", ">=", "<=", "<=>").
- Boolean operands ("true"/"false") and operators ("`&`" (and), "`|`" (or), "`^`" (xor)).
- IFELSE operator
- string operands ("`"xyz"`") with string operators ("+", "*", and all comparison operators listed above).
- bitshift operators ("<<", ">>", "`^`" (xor)).

2.3 Challenge (30%)

For a very good grade, your solution needs to additionally support

- unary operators ("`!`" (not), "`~`" (one's complement)).
- float operands and all respective float versions of the integer operators described in the Core and Completion sections.
- vector and matrix operands.
- vector and matrix operators ("plus" (+), "times" (* (between matrices, or a matrix and a vector)), "dot" (* (between vectors)), "cross" (x), "transpose" (TRANSP)).
- lambdas, quoting, and the "EVAL" command.

Vector and Matrix operators should work as they do in Ruby; test files 220–234 provide respective examples.

Quoting means prefixing any element with an apostrophe (`'`), resulting in that element to be placed on the the stack when it occurs in the input stream, rather than executing it.

QUOTE EXAMPLE

```
INPUT    1 2 '+
OUTPUT   1 2 +
```

The "EVAL" command executes the top element on the stack, as if it were received as part of the input stream:

EVAL EXAMPLE

```
INPUT    1 2 '+ EVAL
OUTPUT   3
```

Note that evaluating an operand, such as the number "1", with "EVAL", does not change the operand, i.e. "EVAL" will have no effect in such cases.

One of the elements that can be evaluated is a lambda. The latter has the form `{ n | <body>}` where `n` describes how many parameters are popped from the stack, and `body` contains regular stack operands/commands, etc. that are executed. Within `body` the parameters x_i can be accessed with variables `x0`, `x1`, etc.

LAMBDA EXAMPLE I

INPUT
```
1 2
{2 | x0 x1 +}
```

OUTPUT
```
3
```

LAMBDA EXAMPLE II

INPUT
```
1 2 3
{3 | x2 x1 x0}
```

OUTPUT
```
3 2 1
```

Within a lambda body, the term "SELF" refers to the lambda expression itself. This means that a lambda term can push itself on to the stack. In combination with "EVAL" this can then be used for recursive evaluations. The following example, calculates the factorial of the top element on the stack, provided that the element below it is a "1":

LAMBDA EXAMPLE III

INPUT
```
1 5
{ 2 | x0 x1 * x1 1 - DUP 0 > SELF 'DROP ROT IFELSE EVAL}
```

OUTPUT
```
120
```

If in doubt about the semantics of an operator (e.g., of the signum operator <=>), use the Ruby semantics of the operator. You may implement additional functionality, but this will not result in any additional marks. In other words, implementing interpreter capabilities not listed above, will not result in bonus marks.

For each completion stage, we have provided respective sample input sequences and the corresponding output files (using an `expected-nxy.txt` naming scheme, to avoid name clashes with your output files). Core functionality relates to files `input-0xy.txt`, where `x` and `y` may range from 0–9 respectively. Completion functionality relates to files `input-1xy.txt`, and Challenge functionality relates to files `input-2xy.txt`. The assignment 1 archive contains respective sample input files and `expected-nxy.txt` files that specify the expected output.

To be awarded marks in a particular category, the `output-nxy.txt` file produced by your solution must contain exactly the same content as the supplied `expected-nxy.txt`, after your implementation has processed the input sequence of file `input-nxy.txt`. For marking, we will use minor variations of the public test files and additional private test files. The percentage of the maximum grade achievable per completion stage will depend on which public and private tests will be passed.

3 The Report

Your report must be formatted using 12pt font, standard line spacing, margins no smaller than 2.54cm, *without exceeding* two A4 pages.

Marks will be awarded for the following:

- **Design**. Explain the main design of your solution, e.g., important data structures, distribution of responsibilities, etc. If you considered design alternatives, please mention those here and explain why you have discarded them.
- **Ruby Features**. List which *noteworthy* Ruby features and libraries you used and why. For instance, mention the use of blocks, but not regular if statements. This section and the ones above will highly benefit from attempts on your behalf to use Ruby idiomatically and/or making the best use of its features. If you explicitly did not make use of a Ruby feature because you did not feel it would have been appropriate or it would not have helped, justify such decisions here.
- **Tooling**. Briefly describe what tooling you used and how it supported or hindered the development of your solution.
- **Comparison**. Discuss how Ruby made it easier or more difficult to create the solution than Java would have.

Use the above four sections in this order to structure your report. Note that your discussions need to specifically relate to your implementation. Generic statements, e.g., enumerating Ruby features without referencing your implementation, will not be awarded any marks.

4 Submission

Submit your

- code – as `ws.rb`, and
- report – as `report.pdf`

electronically by 3 April 2025, before midnight.