T1A3 Terminal Application Slide Deck

By Phillip Markovic

What is the Evaluator?

- It is a terminal application that extends the functionality of a modern calculator.
- Enables the user to submit multi variable mathematical equations
- Draws input data from the terminal
- Draws input data from files

What is the Evaluator (Continued)?

- It does bulk calculations on any number of expressions in an instant
- Displays results to console
- Outputs results to a file
- Allows users to visualise single variable functions on cartesian plane
- Accomplished by generating image files which can be viewed in any browser

Expression Class

- The core part of the system .
- It drives the terminal application
- One instance for each expression (conceptually representative of an expression)
- All application features which are visible to the user are built on this class
- Contains all the structures and information required to evaluate expressions for a given set of values

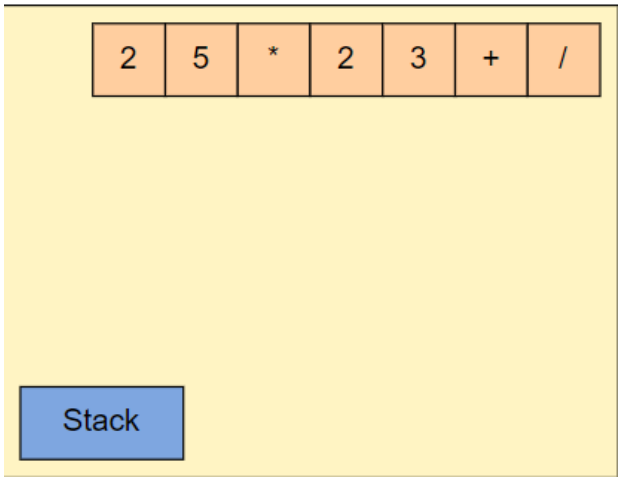
Expression Class Internals (How does it work?)

- Internally the expression class is a reverse polish notation calculator
- Reverse polish notation refers to the way in which the operators follow their operands in a mathematical expression e.g. 3 4 + instead of 3 + 4
- This is distinct from polish notation in which the operator precedes the operands e.g. + 3 4
- Reverse Polish notation is more commonly known as Postfix Notation
- This is distinct from Infix Notation e.g. 3 + 4 which we are all familiar with

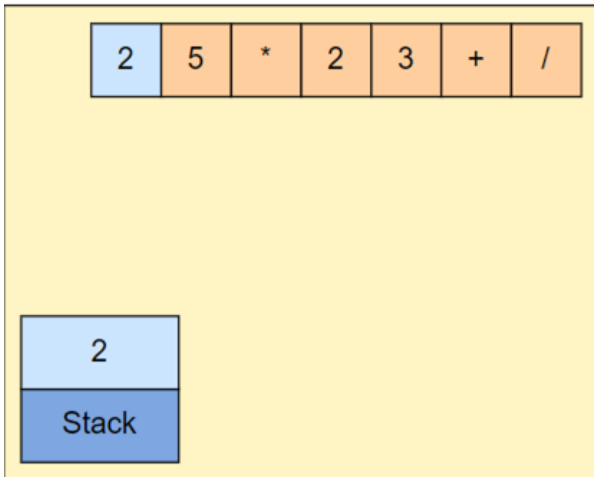
Expression Class Internals (Continued)

- An algebraic expression must be converted from infix notation to postfix notation in order to evaluate it
- Expression class only deals with numbers and operators when evaluating an expression
- Consider the following expression:
- In infix notation is looks like this (2 * 5) / (2 + 3)
- Once converted into postfix notation is looks like this 2 5 * 2 3 + /
- Note the absence of parentheses
- The Expression object will then evaluate this ‘postfix’ expression in the following manner

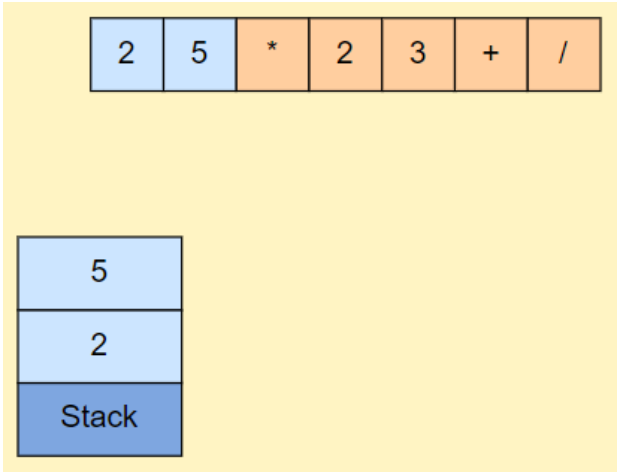
(1)



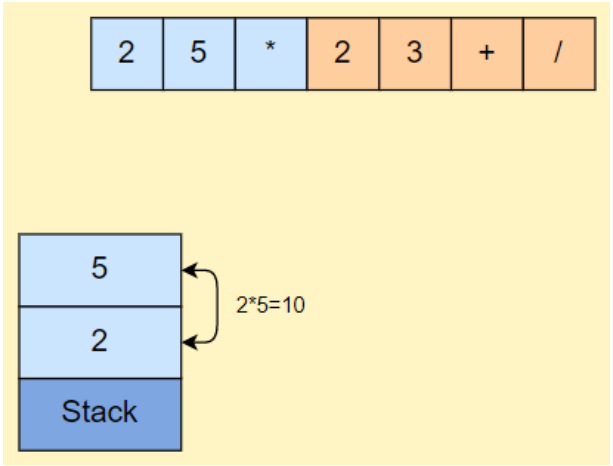
(2)



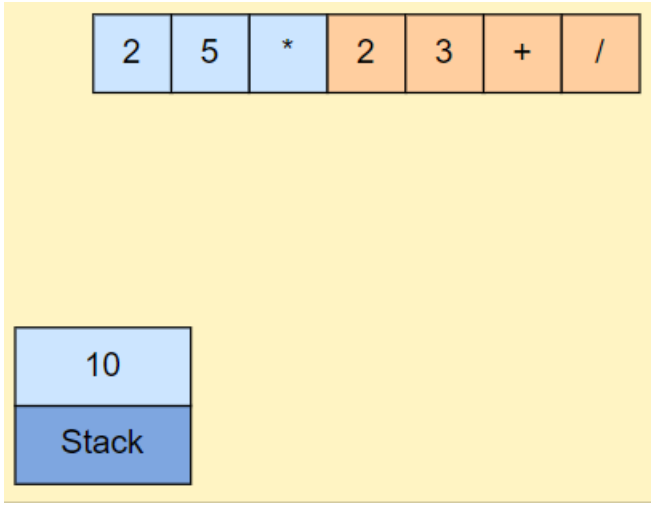
(3)



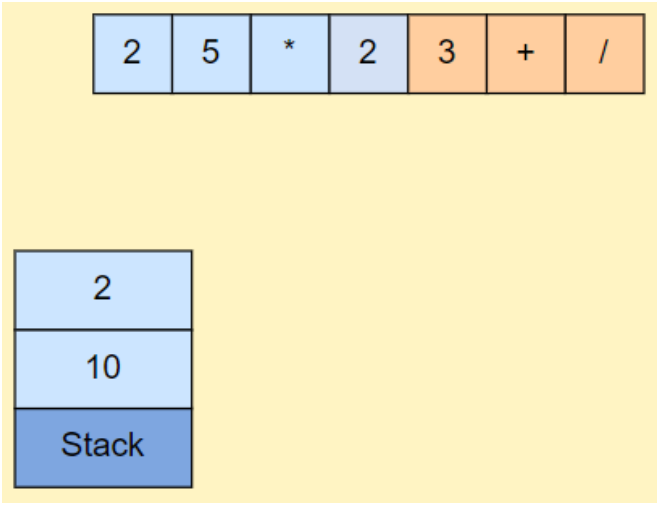
(4)



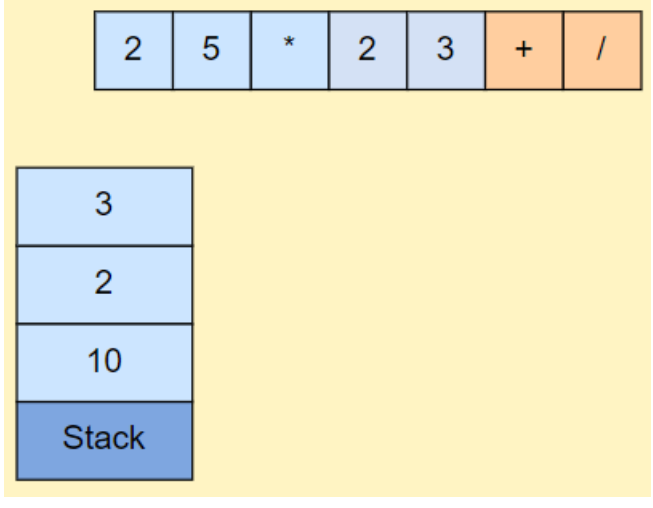
(5)


2
5
*
2
3
+
/
10
Stack

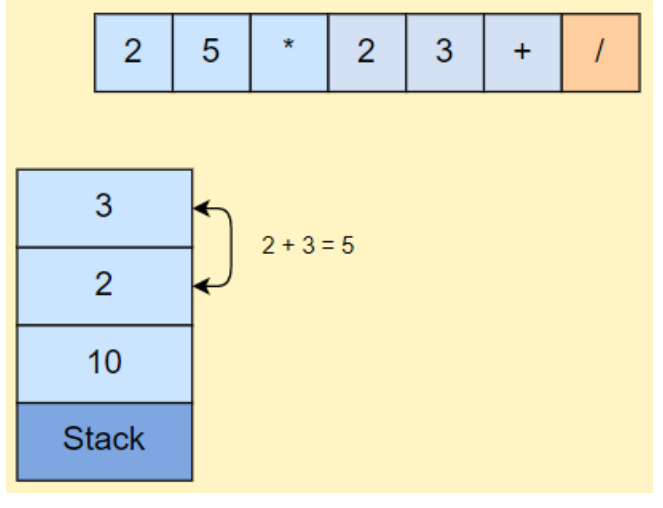
(6)


2
5
*
2
3
+
/
2
10
Stack

(7)

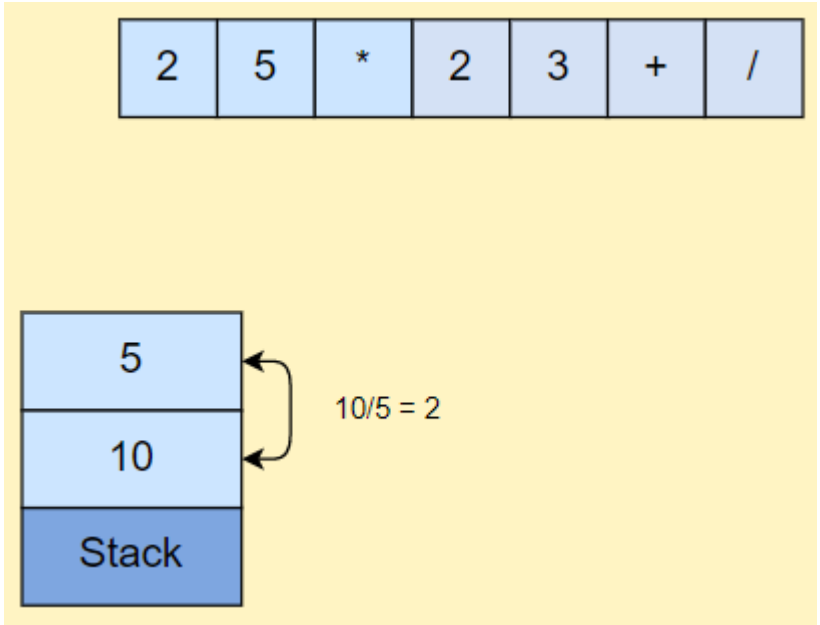

2
5
*
2
3
+
/
3
2
10
Stack

(8)


2
5
*
2
3
+
/
3
2
10
Stack
2 + 3 = 5

- Evaluation takes place by using a stack
- Operands are pushed onto the stack until an operator is encountered.
- The top 2 numbers are popped off the stack and the operator is applied to them.
- The result is pushed back onto the stack.
- This process is repeated until only one value (result) is left in the stack.

(9)



How is process implemented by the Expression class?

4 major methods which are used to carry out this process

- create_infix_list
- check_syntax
- create _postfix_list
- evaluate

Create_infix_list method

- Purpose: To convert an expression into a list of tokens consisting of operators and operands.
- Takes a string representing the expression as a parameter
- Strips out whitespace
- Checks that the expression is properly formed (call to check_syntax)
- Identify and convert unary minuses ‘-x’ to this form mu(x). This enables unary minus operators to be handled like other unary functions e.g. sin, cos, tan, etc.
- Infix notation is preserved in the list

Check_syntax method

- Purpose: Check that a list of elements is a properly formed infix expression
- This is accomplished by looking at each element, identifying the type of element that it is (operator, parentheses, number or variable) and comparing it to the previous element.
- Keeps track of parentheses by pushing “(‘ onto a stack when encountered, then popping them off when “)” found. If any “(“ left in the stack at the end then raise a SyntaxError exception.
- This method has no return value. It takes a list of elements as a parameter and generates exceptions if something in the list is out of place
- Called by create_infix_list

Create_postfix_method

- Create_infix_list provides the input parameter for this method
- Steps through the infix list
- If the element is an operand add it to the postfix list
- Uses an operator stack to hold operators when encountered in the infix list
- Checks precedence of each operator to determine when they should be popped from the operator stack and added to the postfix list
- All parentheses are discarded

Evaluate method

- Takes a postfix list as a parameter
- Reads the postfix list left to right and pushes operands into an operand stack until an operator is encountered
- The top 1 or 2 operators will be popped depending on if the operator is unary or binary
- The calculation is carried out and the result pushed back onto the stack
- This is repeated until there is only one number left in the stack (the answer). This is then returned by the evaluate method
- Note: Even though the postfix list can contain variables, there must be no variables present when this method is called.

Visible Features

- File input and output
- Interactive Mode
- Support for Unlimited Variables
- Image file generation

File Input and Output

- The expression evaluator reads a data set from an input JSON file specified on the command line.
- Allows the app to obtain the multiple expressions their corresponding variables and value assignments
- For each line the app puts together a calculation dictionary which has the following structure:

File input and Output (Continued)

```
calculation_dict[
        {
            'equation': 'sin(x^2)+sin(x*y)-cos(y^2)',
            'result': -2.49548102969227,
            'solved': True,
            'values_obtained': True,
            'substitutions': {'x': '7', 'y': '3.5'}
        }
    ]
```

File input and Output (Continued)

- This structure is passed to the Expression method evaluate_calc_dict, which calculates the result and inserts it back into calculation dictionary which it then passes back.
- The results are then displayed to the screen as in the following run:

```
phillip@MSI:~/projects/PhillipMiguelMarkovic_T1A3/src$ ./evaluate.sh -i input.json
EXPR: x^2+2*x+1, VARS: x=5, RESULT: 36.0
EXPR: -sin(x)-cos(x)+sin(3*x), VARS: x=10, RESULT: 0.3950610158729603
EXPR: 2*x^3+5*x^2-3*x+10, VARS: x=12, RESULT: 4150.0
EXPR: 2*x^3+5*x^2-3*x+10, VARS: x=15, RESULT: 7840.0
EXPR: 2*x^3+5*x^2-3*x+10, VARS: x=20.5, RESULT: 19280.0
EXPR: sin(x^2)+sin(x)-cos(x), VARS: x=7, RESULT: -1.0506683083839874
EXPR: sin(x^2)+sin(x*y)-cos(y^2), VARS: x=7, y=3.5, RESULT: -2.49548102969227
phillip@MSI:~/projects/PhillipMiguelMarkovic_T1A3/src$
```

File Input and Output (Continued)

If an output file is supplied on the command line the results are written to the applications current working directory in JSON format.

phillip@MSI:~/projects/PhillipMiguelMarkovic_T1A3/src$ ./evaluate.sh -i input.json -o output.json

output.json written to the current working directory

phillip@MSI:~/projects/PhillipMiguelMarkovic_T1A3/src$

Note: The app will not permit an output file to be supplied without a corresponding input file

Interactive Mode

- If the evaluate terminal app is invoked with no command line parameters it defaults to interactive mode (a simple shell)
- The user is prompted for an expression
- Is there are any variables in the expression the user is prompted for corresponding values
- A calculation dictionary is constructed with this input data and passed on to Expression through evaluate_calc_dict.
- The answer is returned in the calculation dictionary and displayed to screen
- The user is then prompted for the equation
- The user types ‘quit’ or ‘exit’ to finish

Interactive Mode (Continued)

Here is an example run:

```
phillip@MSI:~/projects/PhillipMiguelMarkovic_T1A3/src$ ./evaluate.sh
Please enter an expression> x^3+3*x^2-y^2+sin(y)+z
Please enter the value for x> 12
Please enter the value for y> 9
Please enter the value for z> 5
2084.4121184852415
Please enter an expression> sin(x)+3**x+7
Error at column 8 in sin(x)+3**x+7
                                    ^
Please enter an expression> quit
phillip@MSI:~/projects/PhillipMiguelMarkovic_T1A3/src$
```

Note what happens if you submit a badly formatted expression

Support for Unlimited Variables

- Supports any number of variables with user defined names in expressions e.g. 3*x+4*(a-dog)-10*sin(cat)
- To do this the app sets up the infix and postfix lists
- Calls extract_variable_names to traverse through a postfix list and identifying which elements are variables
- Those elements are put into a dictionary with their values set to ‘NONE’

  [{'x' : 'NONE', 'a' : 'NONE', 'dog' : 'NONE', 'cat' : 'NONE'}]

- This list is inserted into the calculation dictionary. The values are later obtained from the user

Image File Generation

- The evaluate terminal application can be invoked with the ‘-png’ option followed by an expression (must be limited to 1 variable)
- This will result in the expression plotted to a cartesian plane and then saved as a png file in the applications current working directory
- Accomplished by calling plot_and_save function and passing in an Expression object as well as the upper and lower bounds of a range of values
- This function calls evaluate_list_of_values in Expression. Returns an evaluated list of values
- Both lists are passed in to plot() function in matplotlib module
- A unique file name is generated by the application for each image
- The function savefig() is the called on matplotlib with the generated file name

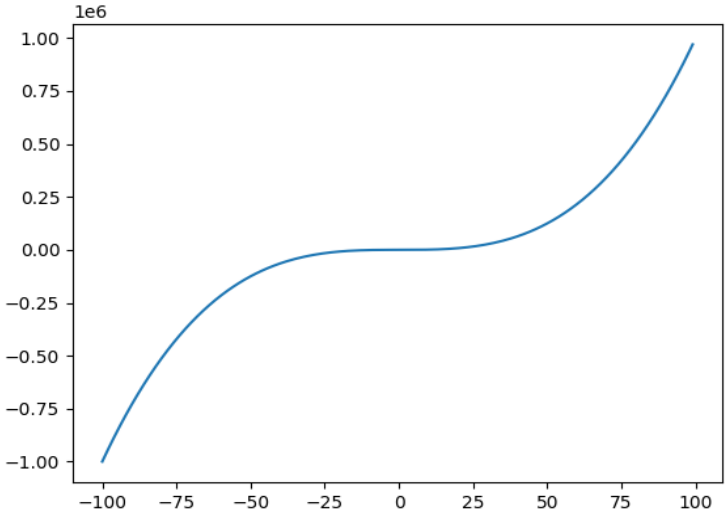
Image File Generation (Continued)

Here is an example of the command line for this feature (image is displayed)

phillip@MSI:~/projects/PhillipMiguelMarkovic_T1A3/src$ ./evaluate.sh -png “x^3”

The figure has been saved to Figure_2023-5-13_15-11-47:83683.png

phillip@MSI:~/projects/PhillipMiguelMarkovic_T1A3/src$



Structure of Application

Application behaviour (features) are based on command line parameters

It checks these and carries out the feature in the following order

- If input file and no output file display results to screen
- If input file and output file write results to output file
- If -png option and expression supplied plot and save image
- If no command line parameters supplied enter interactive mode

Surprisingly, image generation is my favourite part of the app as I enjoy visualising my creations. It was the simplest to implement.

Development & Build Process

- Research the principles that guide the operation of a Reverse Polish Notation calculator
- Identifying methods to create in Expression class for specific purposes
- Identifying and implementing order in which to execute the main features of the app
- Using GitHub every day as a best practice
- Refactoring code to conform to DRY principles
- Ensuring code conforms to coding/commenting standards
- Informative comments
- Unit tests

Challenges

- Translating an algorithm demonstrated as a picture into code
- Handling of unary minus vs binary minus
- Code reuse and abiding by DRY principles
- Strictly adhering to a project implementation plan
- Correctly interpreting requirements
- Deciding on an app to develop
- Clarity in code

Thank you for watching :)