***ABSTRACTION OF DATA STRUCTURES: A CRAZY CALCULATOR***

Jethro G. Albano

Keanu G. Go

1. **Abstract**

An abstract data type is a [mathematical model](https://en.wikipedia.org/wiki/Mathematical_model) for [data types](https://en.wikipedia.org/wiki/Data_type) where a data type is defined by its behaviour ([semantics](https://en.wikipedia.org/wiki/Semantics_(computer_science))) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behaviour of these operations. This contrasts with [data structures](https://en.wikipedia.org/wiki/Data_structure), which are concrete representations of data, and are the point of view of an implementer, not a user.

1. **Introduction**

This module aims to highlight how restricted access linear list data structures, namely: stacks, and queues can be used to solve the computational problem of evaluating algebraic expressions. The entirety of this paper discusses about proper implementation and usage of data types as abstract structures. The various goals of the task are to: read and understand fully about the underlying data structures that will be applied to the program made. In addition, by creating our own version of implementing these underlying data structures, the ability to convert expression programmatically is enhanced.

We have made a program called “A Crazy Calculator”, which uses the combination of underlying data structures to convert an algebraic expression and evaluates it. It is called crazy in such a way that it implements each data structure derived from another data structure. The tasks that we have done are: to create a GUI program which returns the result of some input algebraic expression in which the user inputs an algebraic infix expression. The infix expression is then converted to a postfix expression and uses the converted expression to evaluate the expected result.

1. **Backgrounder**

Basic concepts on abstract implementation of data structures is needed to clearly understand the essence of abstraction in terms of object-oriented programming. The various concepts on stacks and queue manipulation will be important in the process which necessitates the use of stack as queues to assist in the calculation of the problem.

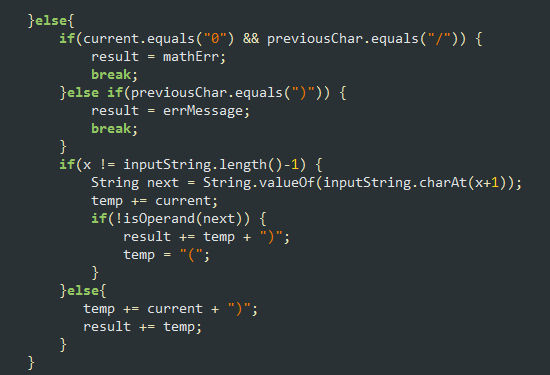
**Abstract Data Structures**

1. Abstraction
2. Linked Lists as Arrays
3. Static sized linked lists
4. Stacks
5. Implementation as Queues
6. Evaluation of an Infix to Postfix Expression
7. **Report Proper**

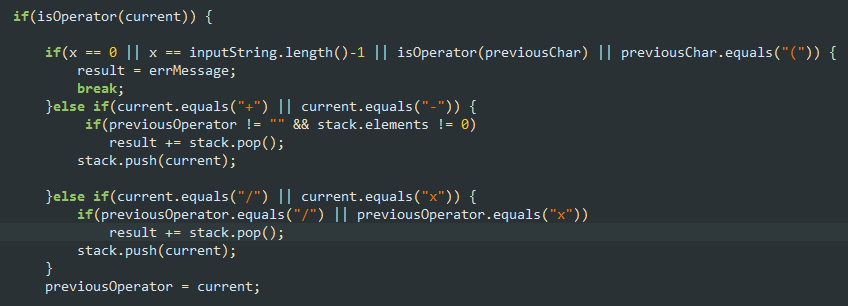
**Infix-Postfix Conversion Algorithm –** the conversion algorithm loops through the input string having comparison and stacking (push and pop) as the main processes throughout the program.

The algorithm is a loop process through the entire string. Starting from the first character, every time it moves to each character in the string, it is checked whether the current character index is an operator, operand, a left or a right parenthesis. After several comparisons and conditions, the current character will then be stored in a previousChar variable, to be compared with the next character of the string. The process is repeated until the last character of the string.

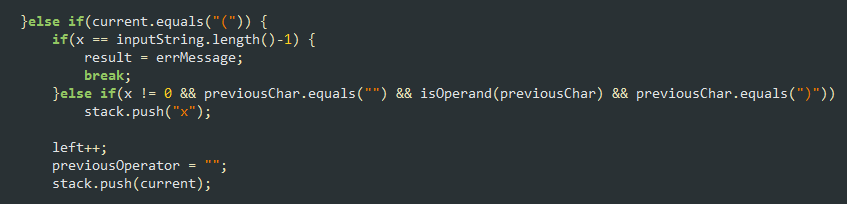
**If OPERAND (number/digit)**

1. If the current character is a zero, it checks if the previousChar of the string is the divide operator. If so, it breaks the loop and returns an error message. Else it checks if the previousChar is “)” which also breaks the loop and returns an error message.
2. It checks if the next index of the current character is also an operand. **(1)** If it is an operand, the current character is stored in a temp string. As the index moves, the concatenation to temp will continue until the next index of the current operand is no longer an operand, then temp is added to the output string. This condition is necessary for the case of having two or more digit numbers present in the input expression (e.g. *256, 93*). **(2)** Otherwise, it is immediately added to the output string.
3. The current character will then be the previousChar for the next index comparison.

**If OPERATOR (+, -, \*, /)**

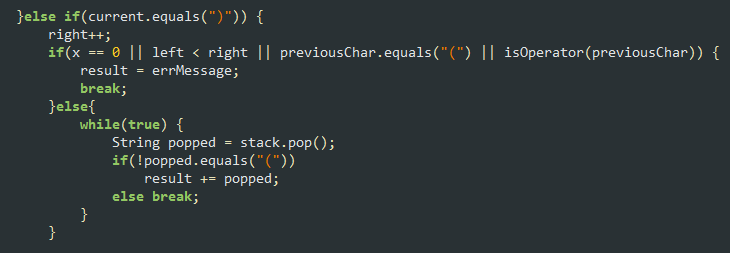
1. Only operators are allowed to be pushed into the stack. If an operator is found in the string, it would be pushed into the stack and compared with the topmost operator in the stack.
2. Immediately breaks the loop and returns an error message if the first or last index of the string is an operator, if the previousChar of the current operator is also an operator (e.g. *9+/25*) and if the previousChar of the current operator is an opening parenthesis (“(”).
3. The precedence of the operations are as follows : (a) / and \* (b) + and –
4. (+ or -) = pushed if the stack is empty. Otherwise, pops and adds the popped character to the output string. Then the current operator is pushed to the stack.
5. (/ or \*) = checks if the previousOperator is also a / or \*. If so, it will pop the previousOperator and add it to the output string, then push the current operator. If not, the current operator would just be pushed to the stack.
6. ****After every comparison and procedures the current operator will then be the previousOperator for the comparison of the next operator.

**If LEFT PARENTHESIS (“ ( ”)**

1. Returns an error message if found at the last index of the string.
2. Pushes the multiply operator (“x”) if the previousChar is an operand or a right parenthesis.
3. Resets the previousOperator to blank. It is because the operations in the parenthesis would only be exclusive within.
4. Pushed into the stack.

**If RIGHT PARENTHESIS (“ ) ”)**

1. Returns an error message and breaks the program if the parenthesis is found at the first index of the string, if the previousChar is a left parenthesis (which signifies a null) or if the previousChar is an operator.
2. If a right parenthesis is encountered in the string, it pops every operator in the stack and adds it to the output string until the left parenthesis is popped.



1. **Result and Discussion**

The conversion from infix to postfix expression is very important since the parsing of the input expression and evaluation is much easier than the infix expression. Infix expression to interpretation or evaluation will be more difficult to execute since the operator precedence is concerned. Postfix expression can be evaluated with a complexity of O(n) since the process of evaluating is from left to right, stacking the operands and when it is an operator then pops two operands and pushes the corresponding result. The postfix expression’s complexity is more difficult to evaluate since operator precedence is much concerned for the program to execute. The postfix expression is no longer concerned with the operational procedure that the operator precedence is no longer concerned.

The use of data structures helps the conversion of the expression containing the step by step process to convert and evaluate the algebraic expression. By the use of own implementation from linked list of array (fixed size) of queues of stack as home data structure provides further knowledge on understanding the different data structure as a fundamental abstract data structures. The data structures are the tools that help convert and evaluate the expression through implementing it programmatically.

1. **Conclusion**

Abstraction is a process of hiding the implementation details and showing only functionality to the user. It is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

The programming assignment mainly focused on the implementation of the property of abstraction. It showed the correct implementation and usage of abstraction in the application. Because of abstraction, the stack’s pop and push functions were easily used without knowing the insides of the implementation of the functions. It helped encapsulate program behaviour and helped having more self-contained modules. The abstraction made the application extendable and flexible. It made the process of restructuring existing computer code without changing its external behaviour which improves the non-functional attributes of the software. More importantly, it gave a much easier setup on source code maintainability and puts the source code in an easily readable structure. Lastly, abstraction is a good way to improve and create a more extensible internal object model of the application.

1. **References**

<https://ece.uwaterloo.ca/~dwharder/aads/Abstract_data_types/>

<http://www.javatpoint.com/abstract-class-in-java>

<https://www.tutorialspoint.com/java/java_abstraction.htm>

<https://dzone.com/articles/why-abstraction-really>

<https://en.wikipedia.org/wiki/Code_refactoring>