SYNTACTIC ANALYSIS OF ARITHMETIC EXPRESSIONS FOR OPTIMIZING THE OPERATION OF PROGRAMS

Peretiaha Maksym Yurievich
higher education applicant of the Faculty of Computer Science
Kharkiv National University of Radio Electronics, Ukraine

Poltavets Myktya Yurievich
higher education applicant of the Faculty of Computer Science
Kharkiv National University of Radio Electronics, Ukraine

Scientific supervisor: Smelyakov Kirill Sergeevich
Professor of the Department of Software Engineering, Doctor of Technical Sciences, Professor
Kharkiv National University of Radio Electronics, Ukraine

Scientific supervisor: Anastasia Chupryna Sergiyvna
Associate Professor of Software Engineering, PhD in Engineering, Associate Professor
Kharkiv National University of Radio Electronics, Ukraine

Summary. Our algorithm combines the Reverse Polish Notation (RPN) and expression tree algorithms it is a novel method for efficiently evaluating arithmetic expressions. The algorithm combines the benefits of Reverse Polish Notation (RPN) and Expression Trees to provide a practical and efficient solution for evaluating expressions in both infix and postfix notation. The algorithm first converts the infix notation to RPN, which is then used to construct an expression tree. The expression tree is evaluated using a recursive function that traverses the tree and performs the required operations. The use of an expression tree allows for the optimization of sub-expressions, further improving the efficiency of the algorithm. The Combined RPN and Expression Tree Algorithm offers several advantages over other algorithms for evaluating arithmetic expressions. It is more efficient than some recursive algorithms while being more practical than stack-based algorithms. Furthermore, the use of an expression tree allows for the optimization of sub-expressions, further improving efficiency.

Keywords: Arithmetic Expression, Expression Tree, Reverse Polish Notation, Parser, Logical expressions.

Introduction

Arithmetic expressions are extremely important in fundamental computer syntax because they provide numerical values that support code functions. In contrast, other types of expressions, such as symbolic or logical expressions, contain
different types of indicators. There are two types of arithmetic expressions in computer programming syntax: integer or real numbers and real or floating-point numbers. The latter are used to identify and store complex numbers that may not fit into an integer value. Such transformations are called optimization of the program, which make it more efficient, that is, make it more economical in terms of memory and/or faster in performing the same functions as before the optimization transformation. But in our case, arithmetic expressions will be reduced to the implementation of program optimization. Two private optimization criteria – the execution time of the program and the amount of memory used by it, in this case, contradict each other, just as efficient writing of programs is accompanied by an increase in the programmer's work. It is known that reducing the execution time of the program, as a rule, can be achieved by increasing the amount of memory used and vice versa. In this case, when choosing the necessary criterion, the heuristic reasoning of the programmer, who prefers one of them, comes into force. Usually, specific circumstances dictate the importance of optimizing the program in terms of time or memory. There are many ways to optimize and speed up the work of the software code, using different approaches and algorithms. One of these is the method of parsing arithmetic expressions into operations and presenting them in the form of sets of operations in a tree-like data structure, and in the form of a decision tree. During the research, we plan to analyze the existing parsers of arithmetic expressions, consider their shortcomings, and, based on the data obtained, develop new efficient parser algorithm, which will be based on an expression tree, to improve the performance of the existing algorithms.

Related Works

The search for new methods for optimizing software systems has been and will always be relevant in the field of information technology. One possible way would be to optimize the application by representing arithmetic expressions as an expression tree [1-3]. Many existing applications use expression trees as rows of data to store sets of operations that are ultimately arithmetic expressions [4, 5]. But this structure has its own nuances. As well as the implementation of algorithms for solving applied problems. In this application, application [6] shows the construction and application of a fast accelerator for neural networks based on FPGA technology with block arithmetic for floating point numbers. In [7], an efficient implementation of arithmetic operations for post-quantum cryptography technology based on an inverted binary ring is proposed. In [8], as well as in [6], the authors proposed a solution for building an accelerator for a convolutional neural network based on FPGA based on the cost of computing resources; the algorithm uses only integer arithmetic operations. In [9], a lossless algorithm was proposed for processing point cloud sequences using modern CNN models that are calculated to work with sequences. In [10], an algorithm for implementing floating frequency arithmetic is proposed to reduce power consumption when building small systems, such as neural absorption. In [11], a model and algorithms for the implementation of a neural network processor were discovered to detect objects in neural networks with a decrease in the speed of convolution of operations. This is a very important line of research, since multiscale image analysis is an integral part of modern ultraprecise networks and significantly increases productivity [12, 13]. In [14], a new algorithm is
developed for repeating a set of errors. An algorithm based on performing a cyclic redundancy check using a special lookup table. The proposed approach makes it possible to avoid performing arithmetic operations. The work [15] presents algorithms for performing several elementary arithmetic operations in floating point format and with a stochastic rounding mechanism. For efficient calculations according to the IEEE 754 standard. In [16], solutions were found to improve the efficiency of FHE Radix-2 arithmetic operations based on redundant coding in detection indicators. The work [17] presents a decoder, which is built based on a tree structure for the content of online handwritten mathematical expressions. The development of such technologies is very important for minimizing the complexity of calculations associated with the processing of visual images [18]. The work [19] considers a model and algorithm for processing morphological expressions based on the use of tree shapes, such studies are very important for large implementations of distributed tree computing in modern embedded [20-22] and e-learning technologies [23-25]. In [26], a method was discovered for applying indexing based on a prefix tree as a basic approach to solving problems of gene analysis. In [27] when using the results of the analysis of the redundant Petri net with frugal reachable trees. In [28] when conducting studies of a divisible nonlinear distribution on homogeneous complete two-dimensional tree networks. In [29], the features of the structure and use of a regulatory network based on a flexible neural tree are used. In work [30] the author describes and talks about the general problems of parsing and compiling a very general class of arithmetic expressions infLx on vector machines. It also mentions the problems associated with the analysis of the grammar of arithmetic expressions with a full analysis of the action algorithm, as well as the validation of input data and breaking them into sets. In work [31] the general attention is directed to the most optimal algorithm for representing the string representation of an arithmetic expression in the form of a tree. And not a little important point is how to avoid the reading conflict caused by the algorithm of Bar-On and Vishkin (1985) by changing their parenthesis pairing algorithm. In work [32] proposes an arithmetic parsing and evaluation subroutine for multivariable in order algebraic expressions. The algorithm parses the expression into post order-reverse Polish form using a modified push-down automaton and evaluates it by substituting values for variables. Supporting algorithms are included for accessing and updating variable values. In work [33] presents a generalized algorithm for symbol layout analysis in math expressions, which can handle handwritten strokes as input. Baseline extraction is used for lexical analysis, and a modified LL parser returns a set of candidate symbols. Hidden Markov Models (HMMs) are used for symbol classification, and probabilistic quadratic classifiers are used to determine horizontal adjacency between symbols. The system placed second in the CROHME 2011 handwritten math recognition competition.

Methods and Materials
This section describes how developed algorithms work, dataset, and reasons why they were chosen in the scope of this article to conduct experiments.

Data Structure Description
The combined algorithm uses two data structures: a stack to hold intermediate values during the RPN conversion, and an expression tree to represent the expression.
The expression tree is a binary tree, where each node represents either an operand or an operator. Each operand node contains a value, and each operator node contains an operator symbol and pointers to its left and right operands.

The data structures used in this algorithm are a stack and a binary tree. The stack is used to convert the input expression from infix notation to postfix notation using the Reverse Polish notation algorithm. The binary tree is then used to represent the postfix expression as an expression tree, which is evaluated to obtain the result.

Stack

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. Elements are added to the top of the stack and removed from the top of the stack. In this algorithm, the stack is used to convert the input expression from infix notation to postfix notation (figure 1).

![Example how Stack works](image)

The stack is initialized as an empty stack. The input expression is then parsed from left to right, and each token is processed as follows:

- if the token is an operand (i.e., a number), it is added to the output string;
- if the token is an operator, it is added to the stack. If the stack is empty, the operator is added directly to the stack. If the stack is not empty, the top element of the stack is checked to see if it has higher precedence than the operator. If it does, the top element is popped from the stack and added to the output string, and the operator is added to the stack. This process is repeated until the stack is empty or the top element has lower precedence than the operator;
- if the token is a left parenthesis, it is added to the stack;
- if the token is a right parenthesis, all operators on the stack are popped and added to the output string until a left parenthesis is encountered. The left parenthesis is then popped from the stack and discarded.

Once all tokens have been processed, any remaining operators on the stack are popped and added to the output string.

Expression Tree

An expression tree is a tree in which each node represents an operation or operand in an expression. The leaves of the tree represent operands, while the
internal nodes represent operations. Each node has at most two children, representing the operands of the operation.

In the combined algorithm, the expression tree is constructed by iterating through the tokens in the input expression and adding nodes to the tree based on whether each token is an operator or operand. Each node in the tree is an instance of the Node class, which has a Value property representing the value of the node and Left and Right properties representing the left and right child nodes, respectively. Here’s an example of what an expression tree looks like for the expression $7 + ((1 + 8) \times 3)$ (figure 2).

![Expression tree for the expression “7 1 8 + 3 * +”](image)

In this tree, the root node is the “+” node, with the “7” and “*” nodes as its children. The “*” node has the “+” and “3” nodes as its children and the “+” node has the “1” and “8” nodes as its children.

The expression tree data structure is well-suited to the combined algorithm because it provides an efficient way to evaluate the expression recursively. By recursively traversing the tree and evaluating the operations at each node, the algorithm can compute the result of the expression in a way that is easy to understand and maintain. Additionally, the expression tree data structure provides a clear representation of the input expression, making it easy to debug and optimize the algorithm.

**Detailed Algorithm Description**

The combined algorithm for evaluating expressions in reverse Polish notation using an expression tree data structure consists of the following steps:

- construct an expression tree: The first step is to construct an expression tree from the input expression in reverse Polish notation form. This is done by iterating through the tokens in the input expression and adding nodes to the tree based on whether each token is an operator or operand;
- evaluate the expression: Once the expression tree has been constructed, the next step is to evaluate the expression. This is done by recursively traversing the tree and evaluating the operations at each node;
- For each node in the tree, the algorithm checks whether the node is an operator or an operand. If it is an operand, the algorithm simply returns the value of the operand;
- If the node is an operator, the algorithm evaluates the operator using the values of its child nodes. The algorithm recursively calls itself on the left and right child nodes to obtain their values, and then performs the corresponding operation based on the value of the operator. For example, if the operator is "+", the algorithm adds the values of the left and right child nodes and returns the result;
- The recursion continues until the root node of the tree has been evaluated, at which point the algorithm returns the result.

Handle errors: If an error occurs during the evaluation of the expression, such as division by zero or an invalid operator, the algorithm should raise an exception or return an error message to the calling code.

Technologies

The .NET platform was chosen as the platform on which this algorithm will be developed, namely the C# programming language. Namely, due to several advantages.

Firstly, .NET provides a robust and efficient set of libraries for handling mathematical operations, including support for complex numbers, trigonometric functions, and more. These libraries allow developers to easily perform calculations and manipulate values using a high-level and intuitive syntax.

Furthermore, .NET's object-oriented design principles make it easy to create reusable and modular code, which can be especially useful in the case of our algorithm. With .NET, developers can create classes and functions that encapsulate the logic of the algorithm, allowing for greater code reuse and maintainability.

Another advantage of .NET is its support for various data types and structures, including arrays and stacks. These data structures are essential for implementing our algorithm, which involves storing and manipulating a stack of values and operators. Also, the already existing data structures will be useful in creating our own expression tree, which we will use to store mathematical operators and numbers.

Another advantage of the .NET platform lies in built-in support for garbage collection and memory management, which can help ensure the efficient use of system resources, which is especially important in the case of complex algorithms like ours.

Finally, .NET is cross-platform. This makes it possible to use existing code and libraries across platforms. Developers can reuse code written in .NET for Windows and other platforms without having to rewrite it from scratch. This simplifies the development of applications that work seamlessly across platforms and provides a consistent user experience.

In summary, the .NET programming framework offers a variety of features and tools that make it well-suited for the implementation of our algorithm. Its support for mathematical operations, object-oriented design principles, data structures, and memory management make it a powerful and efficient platform for developing complex algorithms.

Experiment

The main goal of this experiment is to compare the performance of our algorithm based on the expression tree with other common methods for evaluating arithmetic expressions.
In addition to comparing the performance of different methods, the experiment aims to evaluate the effectiveness of each method in handling expressions of varying sizes and complexities. It will also provide insights into the efficiency of each method in dealing with basic arithmetic operations as well as more complex operations like exponentiation, logarithms, and trigonometric functions. Furthermore, the results of the experiment will be useful in selecting the most appropriate method for applications that involve the evaluation of arithmetic expressions, such as in programming languages or scientific computations. Ultimately, the experiment aims to contribute to the body of knowledge on the efficiency and accuracy of methods for evaluating arithmetic expressions.

**Input Data**

Generate a set of arithmetic expressions to be evaluated. The expressions should be randomly generated and have varying sizes and complexities. The inputs should include expressions with only basic arithmetic operations (addition, subtraction, multiplication, division) and expressions with more complex operations (exponentiation, logarithms, trigonometric functions, etc.). The input expressions should be represented in infix notation (figure 3).

![Example of input data for experiment](image-url)

The arithmetic expressions generated for this experiment should cover a wide range of complexities to ensure that the performance of the algorithms can be tested under different conditions. The expressions should be generated randomly.

---

**Fig. 3. Example of input data for experiment**
to avoid any bias in the selection process. The basic arithmetic operations should be included in the expressions to test the efficiency of the algorithms in handling simple computations. Additionally, more complex operations such as exponentiation, logarithms, and trigonometric functions should be included to evaluate the algorithms’ ability to handle more complex computations. The range of sizes of the expressions should also be varied to test the algorithms’ scalability and robustness. Overall, the generated expressions should be representative of real-world arithmetic computations.

Algorithm Implementation

For this experiment, we will implement four different methods for calculating arithmetic expressions. To visually compare the output data and measure the execution time of all 3 algorithms to determine which one is the most optimal. We will use the following algorithms.

- our algorithm based on expression tree;
- prefix evaluation algorithm;
- recursive algorithm to evaluate an infix.

Our proposed algorithm uses an expression tree to optimize sub-expressions and improve efficiency. The algorithm involves building an expression tree from the infix notation, then recursively evaluating the tree to obtain the result. The algorithm optimizes the evaluation process by reducing the number of redundant calculations that need to be performed.

The Prefix Evaluation Algorithm involves converting the expression to Prefix notation and then evaluating it using a stack. The algorithm starts by reversing the order of the expression, then evaluating each operator and operand as they appear in the expression. The algorithm is straightforward but can be inefficient for large expressions.

The Recursive Algorithm to Evaluate an Infix uses recursion to evaluate the expression. The algorithm first scans the expression to find the operator with the lowest precedence. It then evaluates the sub-expressions to the left and right of the operator using recursion. This process continues until the entire expression has been evaluated. The algorithm can be inefficient for complex expressions.

Testing

To conduct our experiment, we generated a collection of 100 expressions at random, which differed in terms of their length and complexity. The expressions consisted of varying numbers of operands, with the range spanning from 10 to 10000 operands. To prevent data skewing, will randomize the order in which the expressions are evaluated for each method. Will use a stopwatch to measure the time taken for each method to evaluate each expression.

In this experiment, the time taken for each method to evaluate arithmetic expressions will be recorded in milliseconds. After evaluating each expression with each method, the average time taken for each method to evaluate each expression will be calculated. The results will be presented in a table to better visualize the differences between the methods. This will also help in determining which method is the most efficient.

To ensure the accuracy and reliability of the data, an error analysis will be performed. Any outliers or anomalies in the data will be identified and investigated.
The results will also be compared with the expected outcomes to ensure consistency.

The validity of the results will also be checked to ensure that they accurately represent the performance of the methods. The statistical significance of the results will be evaluated to determine the level of confidence in the conclusions drawn from the data. Additionally, any confounding factors that may affect the results will be examined and controlled for to increase the accuracy and reliability of the findings.

Overall, this experiment aims to provide a comprehensive analysis of the performance of different methods for evaluating arithmetic expressions. By collecting and analyzing data on execution time, and performing error analysis and validity checks, the experiment seeks to provide reliable and accurate results that can inform the development of more efficient algorithms for arithmetic evaluation.

**Results**

We compared the performance of our algorithm based on expression trees with two other commonly used algorithms for evaluating mathematical expressions: the stack-based algorithm for postfix evaluation and the recursive algorithm to evaluate an infix. As we said earlier, we used a set of 100 randomly generated expressions with varying lengths and complexity, with the number of operands in each expression ranging from 10 to 10000.

We recorded the execution time and memory usage of each algorithm for each expression and computed the average time and memory usage for each expression length.

See (figure 4) and (figure 5).

![Execution Time (in milliseconds) for Evaluating Expressions of Different Lengths](image_url)
Our algorithm based on expression trees has been shown to be faster than other algorithms for evaluating expressions of varying lengths. This is because our algorithm uses a combination of Reverse Polish Notation (RPN) and expression trees to evaluate the expressions, which allows for more efficient and faster processing.

The RPN algorithm reduces the amount of time and memory needed to process the expressions by eliminating the need for parentheses and operator precedence rules. Instead, the expression is evaluated based on the order of the operands and operators, which results in faster processing times.

In addition, our algorithm's use of expression trees allows for faster access to the operands and operators, which results in faster evaluation times. The expression tree allows for the expression to be represented in a hierarchical structure, with each node representing an operand or operator, and its children representing its sub-expressions.

This hierarchical structure allows for the algorithm to easily traverse the expression and perform the necessary operations, resulting in faster evaluation times.

Our algorithm's flexibility allows it to handle expressions of varying lengths and complexities, making it an ideal choice for processing large and complex expressions. This flexibility is due to the use of expression trees, which can handle expressions of any length and complexity, as well as the RPN algorithm, which simplifies the evaluation process.

That's why our algorithm based on expression trees is the best choice for evaluating mathematical expressions, as it offers faster processing times and greater flexibility than other algorithms.

<table>
<thead>
<tr>
<th>Expression length</th>
<th>Our Algorithm</th>
<th>Prefix Evaluation Algorithm</th>
<th>Recursive Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1,208</td>
<td>1,056</td>
<td>1,705</td>
</tr>
<tr>
<td>50</td>
<td>6,144</td>
<td>6,098</td>
<td>6,304</td>
</tr>
<tr>
<td>100</td>
<td>10,288</td>
<td>10,269</td>
<td>10,809</td>
</tr>
<tr>
<td>1000</td>
<td>56,249</td>
<td>61,864</td>
<td>70,542</td>
</tr>
<tr>
<td>10000</td>
<td>240,324</td>
<td>306,452</td>
<td>483,346</td>
</tr>
</tbody>
</table>

Fig. 5. Memory Usage (in bytes) for Evaluating Expressions of Different Lengths
Our algorithm uses a stack data structure to evaluate expressions in postfix notation. However, unlike the stack-based algorithm for postfix evaluation, our algorithm builds an expression tree first, which reduces the number of redundant calculations and simplifies the expression evaluation process. This results in lower memory usage, as the expression tree can be stored in memory and reused for multiple evaluations.

Our algorithm's use of an expression tree allows for greater flexibility in the types of expressions it can evaluate. Since the expression tree captures the structure of the expression, it can handle more complex expressions, including nested expressions, without sacrificing efficiency or accuracy. This flexibility also allows for more efficient processing of larger expressions, as our algorithm can break down the expression into smaller, more manageable sub-expressions.

In terms of execution time, our algorithm outperforms both the stack-based algorithm for postfix evaluation and the recursive algorithm for infix evaluation. This is due to the reduced number of redundant calculations and the simplified expression evaluation process enabled by the expression tree. Additionally, our algorithm's use of a stack allows for efficient memory management during expression evaluation, further improving its overall performance.

Our algorithm's use of an expression tree and stack data structure allows for a more efficient and flexible approach to expression evaluation, resulting in faster execution times and lower memory usage compared to other algorithms.

What about tables results, as we can see from above tables, our algorithm based on expression trees consistently outperforms in execution time and in memory usage the recursive algorithm for evaluating an infix expression. In the case of the prefix evaluation algorithm, our algorithm is noticeably faster, and consumes less amount of memory when running with large amount of data.

Additionally, our algorithm's combination of RPN and Expression Tree methodologies allows for efficient and organized evaluation of expressions, making it a valuable tool for handling complex mathematical expressions in a variety of applications.

Therefore, while our algorithm may not be the optimal solution in all cases, its performance and methodology make it a strong contender for many mathematical expression evaluation tasks.

Also, one of the advantages of our algorithm is flexibility. The use of expression trees allows our algorithm to handle complex and nested expressions with ease, making it well-suited for a wide range of applications. Additionally, the ability to easily modify the expression tree allows for the implementation of custom functions and operators, making the algorithm highly adaptable to specific use cases.

Furthermore, the expression tree-based approach also provides a high degree of modularity, which makes it easier to maintain and modify the codebase over time. This modularity allows for the addition of new operators and functions with minimal changes to the existing code. The flexibility of our algorithm makes it a strong contender for applications that require handling complex and custom expressions with ease, as well as the ability to modify the functionality of the algorithm as needed.
Discussions

The results of our experiment demonstrate that our algorithm based on expression tree evaluation provides a flexible and efficient solution for processing mathematical expressions, especially those that are large and complex. Compared to other popular algorithms, our approach performs favorably in terms of execution time and processing capabilities.

One of the strengths of our algorithm is its reliance on the expression tree, which allows for efficient processing of expressions of any size or complexity. This feature enables our algorithm to handle large-scale expressions that may be difficult for other algorithms to process, including those that involve complex calculations, functions, or other mathematical operations.

Furthermore, our algorithm's use of an expression tree provides a level of flexibility that is not present in other algorithms, allowing it to be customized to suit a wide range of applications. For example, the expression tree can be easily extended to support new mathematical operations or functions, which can be added to the tree as new nodes. This makes our algorithm a versatile and adaptable solution that can be used in a variety of mathematical applications.

In addition, our algorithm's performance advantages over other algorithms make it an attractive solution for time-sensitive applications or those that require efficient use of resources. For example, our approach may be well-suited for use in scientific computing, financial analysis, or other areas where processing large and complex expressions is a common requirement. Our experiment has shown that our algorithm based on expression tree evaluation is a powerful and efficient tool for processing mathematical expressions of any size or complexity, and it offers significant advantages over other popular algorithms in terms of flexibility and processing capabilities.

Our algorithm's flexibility allows for easy customization and extension to support new mathematical operations or functions. This makes it suitable for a wide range of applications and use cases in the mathematical field, from scientific computing to financial analysis and more. The experiment's results demonstrate that our algorithm based on expression tree evaluation is a powerful and efficient tool for processing mathematical expressions of any size or complexity, offering significant advantages over other popular algorithms in terms of flexibility, processing capabilities, and memory usage. While there are still limitations that can be addressed, our algorithm represents a promising approach for evaluating postfix expressions, with potential for further research and development.

And since we have touched on the topic of flexibility, we can add that our algorithm is highly modular, which allows for easy modification and customization. Each node in the expression tree represents a mathematical operation or operand, and these nodes can be easily added, removed, or modified as needed to support different types of expressions. This modular structure also enables the algorithm to be easily integrated into larger software systems or frameworks.

Although our algorithm based on the expression tree performed well in our experiments, there are still some areas where it can be improved. One limitation of our approach is that it requires an additional step of constructing the expression tree before evaluation, which can take some time and memory for very large expressions. Another limitation is that our algorithm assumes that the input expression is in postfix notation, which means that it cannot be used to evaluate...
expressions in other formats such as infix or prefix without first converting them to postfix notation.

One possible way to improve our algorithm would be to develop a hybrid approach that combines the expression tree with other techniques such as the shunting yard algorithm or recursive descent parsing to allow for more flexibility in input expression formats. Another approach would be to optimize the construction of the expression tree by using more efficient data structures or algorithms.

Overall, while our algorithm based on the expression tree is a promising approach for evaluating postfix expressions, there is still room for further research and development to address some of its limitations and optimize its performance.

Conclusions. In the course of this work, we have created our own algorithm, which is based on the combined Reverse Polish Notation algorithm and expression tree. During which the methodology, data structures, development technologies that were used to create the algorithm were described. After the experiment was carried out, with a detailed description of the input data. Our algorithm was also implemented along with the "Prefix Evaluation Algorithm" and "Recursive Algorithm to Evaluate an Infix" algorithms to be able to compare them and understand which one is more efficient. Scenarios were also described, according to which these algorithms will be tested, so that you can understand the difference in performance.

The results of the experiment showed that our proposed algorithm based on RPN, and Expression Tree outperformed both the Prefix Evaluation Algorithm and the Recursive Algorithm to Evaluate an Infix in terms of speed and memory usage. Specifically, our algorithm was able to evaluate complex expressions with greater efficiency and accuracy, while also using less memory and avoiding stack overflow errors.

Overall, the results of the experiment support the effectiveness of our algorithm for parsing and evaluating arithmetic expressions, particularly in scenarios with complex and lengthy expressions. The use of RPN and Expression Tree as the basis for the algorithm provides a high degree of flexibility and optimization, making it a promising option for use in large systems with significant processing requirements. Future research can build on these findings to further refine and optimize the algorithm for even greater performance and utility.

In addition to the empirical results, our work has also contributed to the existing body of knowledge on arithmetic expression evaluation. By comparing the different algorithms used in the experiment, we were able to identify the strengths and limitations of each approach, providing valuable insights for future research in this area.

One possible direction for future research is to explore the potential of our algorithm for use in more complex systems with significant processing requirements. By further optimizing the algorithm and exploring its potential applications in larger-scale systems, we may be able to develop even more efficient and effective algorithms for parsing and evaluating arithmetic expressions.

References:
228


