A study of algorithms used for evaluating Algebraic Expressions by Computers.

What is an Expression?

An expression is a combination of symbols that can be numbers (constants), variables, operations, symbols of grouping and other punctuation written in a specific format/way. An arithmetic expression is a mathematical expression that consists of operands (constants, variables, or literals) and operators (+, -, *, /, %, etc.) that are used to perform arithmetic calculations. Arithmetic expressions can be represented in various ways such as infix notation (where operators are placed between operands), prefix notation (where operators are placed before the operands), or postfix notation (where operators are placed after the operands). For example, the following is an arithmetic expression in infix notation:

2 + 3 * 4 - 5 / 6

In this expression, the operands are the numbers 2, 3, 4, 5, and 6, and the operators are +, *, -, and /. The expression can be evaluated by following the order of operations: first, multiply 3 and 4, then divide 5 by 6, add 2 to the result, and finally subtract the result from the previous step. Arithmetic expressions are commonly used in DSA when implementing algorithms that involve mathematical calculations. For example, arithmetic expressions can be used in sorting algorithms to compare and swap elements, in tree algorithms to traverse and modify nodes, and in graph algorithms to compute distances and paths.

Prefix	Infix	Postfix
+23	2+3	23+
-xy	x-y	xy-
+a*bc	a+b*c	abc*+

About Operator Precedence & Associativity:

In computer programming, operators in an expression are evaluated in a specific order of precedence and associativity. The order of precedence determines which operator is evaluated first, and the associativity determines the order in which operators of the same precedence are evaluated.

The order of precedence of operators is typically as follows:

1. Parentheses ( )
2. Exponentiation (**)
3. Unary operators (+, -)
4. Multiplication, division, and remainder (*, /, %)
5. Addition and subtraction (+, -)

6. Comparison (==, !=, >, >=, <, <=)
7. Logical AND (and)
8. Logical OR (or)

For example, in the expression **2 + 3 * 4**, the multiplication operator has higher precedence than the addition operator, so the expression is evaluated as **2 + (3 * 4)**, which results in **14**.

If there are operators of the same precedence, the associativity determines the order in which they are evaluated. The associativity of an operator can be either left-to-right or right-to-left. Operators that are left-associative are evaluated from left to right, while operators that are right-associative are evaluated from right to left.

For example, the subtraction operator in the expression **10 - 5 - 3** is left-associative, so it is evaluated from left to right as **(10 - 5) - 3**, which results in **2**. On the other hand, the exponentiation operator in the expression **2 ** 3 ** 2** is right-associative, so it is evaluated from right to left as **2 ** (3 ** 2)**, which results in **512**.

It is important to understand the order of precedence and associativity of operators in expressions to correctly evaluate them and avoid errors in programming.

About Infix, Prefix and Postfix Notations

Infix notation is a way of writing mathematical expressions in which the operators are placed between the operands. For example, the expression "3 + 4" is written in infix notation, where "+" is the operator and "3" and "4" are the operands. In infix notation, the order of operations is determined by operator precedence and parentheses. For example, multiplication and division are typically performed before addition and subtraction, unless parentheses are used to indicate a different order of operations.

In prefix notation, also known as Polish notation, the operator comes before the operands. This format eliminates the need for parentheses and has a fixed order of operations, making it easier to parse and evaluate expressions. However, it can be hard to read for complex expressions.

Reverse Polish notation (**RPN**), also known as **reverse** Łukasiewicz **notation**, is a way of writing mathematical expressions in which the operators come after the operands. For example, the infix expression "3 + 4" would be written in postfix notation as "3 4 +". In postfix notation, the order of operations is determined solely by the order of the operands and operators. The first operand encountered is used with the first operator encountered, and so on.

For example, the postfix expression "3 4 + 5 *" is evaluated as follows:

- Push 3 onto the stack.
- Push 4 onto the stack.
- Pop 4 and 3 from the stack, add them, and push the result (7) onto the stack.

- Push 5 onto the stack.
- Pop 5 and 7 from the stack, multiply them, and push the result (35) onto the stack.
- The result is 35.

Postfix notation is commonly used in computer programming, particularly in stack-based programming languages such as Forth and PostScript. It is also used in some calculators, where it can simplify the process of evaluating complex expressions.

Infix and postfix notations have their own advantages and disadvantages, and which one is better depends on the context and the specific use case.

Infix notation is more familiar and intuitive to most people, as it closely resembles the way we write and speak mathematical expressions. It also allows for the use of parentheses to specify the order of operations, which can make complex expressions easier to read and understand.

On the other hand, postfix notation can be easier to parse and evaluate algorithmically, particularly in computer programming. This is because postfix notation does not require the use of parentheses or operator precedence rules, and the order of operations can be determined simply by iterating over the operands and operators in order.

Conversion from infix to postfix notation is useful in computer programming and other applications for several reasons:

- Evaluation: Postfix notation is often easier to evaluate algorithmically than infix notation, because it does not require the use of parentheses or operator precedence rules. This can simplify the process of evaluating complex expressions, particularly in stack-based programming languages or in situations where memory is limited.
- Parsing: Postfix notation is also easier to parse than infix notation, because the order of operations can be determined simply by iterating over the operands and operators in order. This can simplify the process of parsing mathematical expressions in computer programs.
- Storage: Postfix notation can be more compact than infix notation, because it eliminates the need for parentheses and operator precedence rules. This can reduce the amount of memory needed to store mathematical expressions in computer programs.
- Transmission: Postfix notation can also be more efficient for transmitting mathematical expressions over a network or between systems, because it requires less bandwidth and can be more easily parsed and evaluated by remote systems.

Historical Importance: Hewlett-Packard (HP) engineers were among the first to design and produce calculators that used Reverse Polish Notation (RPN) as their primary input method. The first HP calculator to use RPN was the HP-9100A, which was introduced in 1968. RPN was chosen as the input method for HP calculators because it provided a more efficient and natural way of entering and evaluating mathematical expressions. With RPN, the user would enter the operands first, followed by the operator. For example, to add 2 and 3, the user would enter "2

[enter] 3 +". HP calculators that used RPN became very popular among engineers, scientists, and other professionals who needed to perform complex calculations quickly and efficiently. Today, RPN calculators are still used by many professionals and enthusiasts who appreciate their efficiency and ease of use.

Current Relevancy: Stack-oriented programming languages such as Forth, Factor, and PostScript are some of the current implementations that use Reverse Polish Notation (RPN) as their primary input and evaluation method. The UNIX system calculator program "dc" (desktop calculator) also uses postfix notation as its primary input method. Postfix notation is often preferred in calculators and other mathematical tools because it eliminates the need for parentheses and operator precedence rules, which can make complex expressions easier to enter and evaluate. It also allows for easy manipulation of the stack of operands, which can simplify the process of performing iterative or recursive calculations.

While postfix notation is often preferred in computer programming and calculators, infix notation remains the dominant notation in mathematics and is commonly used in everyday life. It is worth noting that infix and postfix notations are just two of many possible ways to represent and evaluate mathematical expressions. Other notations, such as prefix notation and Polish notation, are also used in various contexts.

Motivation: Converting infix notation to postfix notation is significant in the field of computing and programming for several reasons, including parsing, compiler development, evaluation algorithms, and error handling. Postfix notation can also help in solving significant challenges in computing, such as memory management, recursive functions, expressions with mixed operators, and interoperability between different programming languages and systems.

1. Fixed order of operations: In postfix notation, the order of operations is fixed, which eliminates the need for parentheses to indicate the order of evaluation. This makes the expression easier to evaluate, especially for complex expressions.
2. Easy to parse: Postfix notation can be easily parsed using a stack data structure. Each operand is pushed onto the stack, and when an operator is encountered, the top two operands are popped, the operator is applied to them, and the result is pushed back onto the stack.
3. Efficient evaluation: Postfix notation can be evaluated efficiently using a stack data structure, which makes it a preferred format for evaluating expressions in many computer programs, especially in compilers and interpreters.
4. Reduced complexity: Infix notation can be ambiguous and difficult to parse for complex expressions, while postfix notation is unambiguous and easier to parse, making it a preferred format for many computer programs.
5. Reduced memory usage: Infix notation requires the use of parentheses to indicate the order of evaluation, which can increase the memory usage of the expression. Postfix

notation eliminates the need for parentheses, reducing the memory usage of the expression.

6. While postfix notation is unambiguous, infix notation can be ambiguous, especially for expressions involving operators with the same precedence. For example, the expression 3 + 4 * 5 can be interpreted as (3 + 4) * 5 or as 3 + (4 * 5), depending on the order of evaluation. To eliminate this ambiguity, parentheses are used to explicitly indicate the order of evaluation in infix notation.

   In postfix notation, there is no need for parentheses, and the order of evaluation is determined by the position of the operators in the expression. This eliminates the ambiguity associated with infix notation, making postfix notation a preferred format for many computer programs that manipulate expressions.

Hence, the conversion algorithms for infix to postfix notation need to be studied and understood, and learning about these algorithms can help in understanding the fundamentals of programming and computer science.

ALGORITHMS TO CONVERT INFIX EXPRESSIONS INTO POSTFIX EXPRESSIONS

I. Manually Converting Infix to Postfix [Parenthesizing Infix]

The first thing you need to do is to fully parenthesize the expression. So, the expression (3+6) *(2-4) +7 becomes (((3 + 6) * (2 - 4)) + 7)

Now, move each of the operators immediately to the right of their respective right parentheses. (((3 + 6) * (2 - 4)) + 7) becomes 3 6 + 2 4 - * 7 +

As you see this can be computationally expensive since “parenthesizing” the expression is not feasible for longer expressions and hence becomes inefficient.

Algorithm

1. Define a function called "isOperator" that takes a character as input and returns true if it is one of the operators (+, -, *, /), false otherwise.

2. Define a function called "getPrecedence" that takes an operator character as input and returns an integer value indicating its precedence level.

3. Define a function called "infixToPostfix" that takes an infix expression as input and returns its equivalent postfix notation.

4. Inside the "infixToPostfix" function, create an empty stack called "s" and an empty string called "postfix".

5. Iterate through each character in the input infix expression:

   a. If the character is an operand (a letter or digit), add it to the "postfix" string.

   b. If the character is an operator, repeatedly pop operators from the stack and add them to "postfix" if they have equal or higher precedence than the current operator, then push the current operator onto the stack.

   c. If the character is a left parenthesis, push it onto the stack.

   d. If the character is a right parenthesis, repeatedly pop operators from the stack and add them to "postfix" until a matching left parenthesis is found. Discard both the left and right parentheses.

6. After iterating through all characters, repeatedly pop any remaining operators from the stack and add them to "postfix".

7. Return the final "postfix" string.

8. In the main function:

   a. Define an input infix expression as a string called "infix".

b. Create a temporary string that fully parenthesizes the infix expression by adding left and right parentheses around it. Store this in a variable called "temp".

c. Create an empty string called "parenthesized" and an empty stack called "s".

d. Iterate through each character in the "temp" string:

i. If the character is an operand, add it to the "parenthesized" string.

ii. If the character is an operator, repeatedly pop operators from the stack and add them to "parenthesized" if they have equal or higher precedence than the current operator, then push the current operator onto the stack. Also add a space after the operator.

iii. If the character is a left parenthesis, push it onto the stack and add it to the "parenthesized" string.

iv. If the character is a right parenthesis, repeatedly pop operators from the stack and add them to "parenthesized" until a matching left parenthesis is found. Discard both the left and right parentheses and add only the right parenthesis to the "parenthesized" string.

e. Create an empty string called "replaced".

f. Iterate through each character in the "parenthesized" string:

i. If the character is a left parenthesis, ignore it.

ii. If the character is an operator, add it to the "replaced" string followed by the next character (which should be a right parenthesis).

iii. If the character is an operand or a right parenthesis, add it to the "replaced" string.

g. Call the "infixToPostfix" function with the "replaced" string as input to obtain the final postfix notation.

h. Print out the input infix expression, the fully parenthesized expression, and the final postfix expression.

9. End of the program.

Algorithm Analysis

The time complexity of the infixToPostfix function is O(n), where n is the length of the input infix expression. This is because each character in the infix expression is visited once, and each operation performed in the function takes constant time.

The space complexity of the infixToPostfix function is also O(n), because in the worst case, all the characters in the infix expression could be operators that are added to the stack, and then added to the postfix expression after being popped from the stack.

For the main function, the time complexity of the algorithm is O(n), where n is the length of the input infix expression. This is because there are three loops that iterate over the input expression, each taking O(n) time.

The space complexity of the main function is O(n), because the stack used to store operators can at worst case hold all the operators in the input expression.

The best case scenario is when the input expression is empty, in which case the time and space complexities are both O(1).

The worst-case scenario is when the input expression is fully parenthesized and all the operators have equal precedence, in which case the time and space complexities are both O(n).

II. Single Stack-based algorithm to convert from Infix to Postfix expression [The Shunting-Yard Algorithm]

History

The Shunting Yard algorithm was developed by Edsger W. Dijkstra in the mid-1960s while he was working on the design of the ALGOL 68 programming language. The algorithm was originally intended to be used as part of the ALGOL 68 compiler.

Dijkstra's motivation for developing the Shunting Yard algorithm was to solve the problem of parsing mathematical expressions in a way that could be easily implemented by a computer program. He realized that traditional methods for parsing mathematical expressions, such as the use of recursive descent parsers, were difficult to implement and often led to problems with ambiguity and inefficiency.

Dijkstra's solution was to develop a parsing algorithm based on a stack data structure that could be used to convert an infix expression to a postfix expression, which could then be evaluated using a simple stack-based algorithm. The algorithm was named after the Shunting Yard railway yard, which Dijkstra visited during his childhood, and which inspired the stack-based design of the algorithm.

The Shunting Yard algorithm became widely used in the development of compilers and other software systems that need to parse mathematical expressions. It is known for its simplicity and efficiency and has been implemented in many different programming languages and environments.

Analogy

The Shunting Yard algorithm is named after the process of shunting railway cars in a marshalling yard or classification yard. In a railway yard, railway cars are separated onto one of several tracks based on their destination, in order to make up new trains or to divide existing trains.

Similarly, the Shunting Yard algorithm separates mathematical expressions into individual tokens and places them onto one of two stacks based on their precedence or associativity, in order to convert the expression from infix notation to postfix notation.

Just as railway cars must be shunted several times along their route in order to reach their final destination, an expression must be transformed several times in order to be evaluated or simplified. The Shunting Yard algorithm facilitates this process by breaking down the expression into smaller units and organizing them in a way that makes them easier to work with.

The use of the term "Shunting Yard" as the name for the algorithm serves as an analogy to the process of shunting railway cars in a marshalling yard, where cars are separated and organized in a way that enables them to reach their intended destinations efficiently. Similarly, the Shunting Yard algorithm separates and organizes mathematical expressions in a way that enables them to be evaluated or simplified efficiently.

Introduction

The Shunting Yard algorithm is a parsing algorithm used to convert an infix notation mathematical expression to postfix notation. The algorithm was developed by Edsger Dijkstra in 1961 and was named after the Shunting Yard train yard where train cars are sorted and arranged. The algorithm was originally designed for use in compilers to parse mathematical expressions and evaluate them efficiently.

This is one of the most popular algorithms for converting infix to postfix. It involves using a stack to keep track of operators and operands as they are encountered. As each token is read from the infix expression, it is either added to the output string (if it is an operand) or pushed onto the stack (if it is an operator). When another operator is encountered, it is compared to the operator on top of the stack. If the new operator has higher precedence, it is pushed onto the stack. If the new operator has lower or equal precedence, the operator on top of the stack is popped and added to the output string until the new operator can be pushed.

Algorithm

1. Define a function **precedence** that assigns a numerical value to each operator, based on its precedence level.
2. Define a function **infixToPostfix** that takes an infix expression as input and returns the equivalent postfix expression as output.
3. Initialize an empty string called **result** to store the postfix expression, and a stack called **operators** to store the operators.
4. Iterate through each character of the input infix expression, and for each character, perform one of the following actions:

- If the character is an operand (a letter or a digit), append it to the **result** string.
- If the character is an opening parenthesis, push it onto the **operators** stack.

- If the character is a closing parenthesis, pop operators from the **operators** stack and append them to the **result** string until an opening parenthesis is found. Then pop the opening parenthesis from the **operators** stack.
- If the character is an operator, pop operators from the **operators** stack and append them to the **result** string as long as the operator at the top of the stack has equal or higher precedence than the current operator. Then push the current operator onto the **operators** stack.

5. After iterating through all characters of the input expression, pop any remaining operators from the **operators** stack and append them to the **result** string.
6. Return the **result** string as the equivalent postfix expression.

The code also includes a function **evaluatePostfix** that takes a postfix expression as input and returns its numerical result as output. This function is not directly related to the infix to postfix conversion algorithm but is included to demonstrate how the postfix expression can be used to perform calculations.

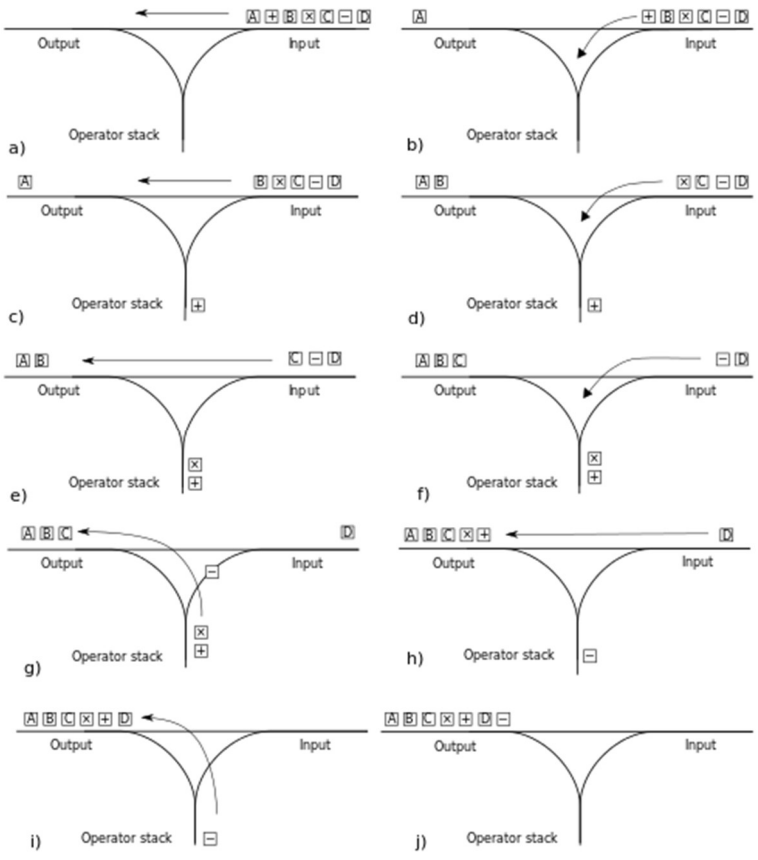

Figure 1: Shunting Yard Algorithm

This algorithm was then implemented using C++ and tested on several algebraic expressions. I analyzed the working of one of the test cases by the algorithm:

Infix Expression: *3+4*2/(1-5)^2*

Postfix Expression: *342*15-2^/+*

Current Token	Operator Stack	Output
3		3
+	+	3
4	+	34
*	*+	34
2	*+	342
/	/+	342*
(	(/+	342*
1	(/+	342*1
-	(-(/+	342*1
5	(-(/+	342*15
)	(/+	342*15-
^	(^/+)	342*15-
2	(^/+)	342*15-2
	/+	342*15-2^
		342*15-2^/+

Algorithm Analysis

- Time complexity: The time complexity of the algorithm is O(n), where n is the length of the expression. This is because the algorithm iterates through the expression once, and the time taken for each operation within the loop is constant.
- Space complexity: The space complexity of the algorithm is O(n), where n is the length of the expression. This is because the algorithm uses a stack to store operators and operands, and the size of the stack depends on the number of elements in the expression.
- Best case scenario: The best-case scenario is when the expression is a single number, such as "5". In this case, the infixToPostfix function would simply return the input expression and the evaluatePostfix function would only need to push a single value onto the stack, resulting in a time complexity of O(1) and a space complexity of O(1).
- Worst case scenario: The worst-case scenario is when the expression is very long and complex, with many nested parentheses and operators. In this case, the infixToPostfix function would need to perform many operations to convert the expression to postfix notation, resulting in a time complexity of O(n). Similarly, the evaluatePostfix function would need to perform many operations to evaluate the postfix expression, resulting in a time complexity of O(n). The space complexity in this case would also be O(n), as the stack would need to store many elements.

In summary, the time complexity of the given implementation of the infix to postfix conversion algorithm [Shunting Yard Algorithm] is O(n), where n is the length of the input expression. The space complexity is also O(n) since the maximum size of the stack used is proportional to the length of the expression.

In terms of the best and worst-case scenarios, the best-case scenario occurs when the input expression is already in postfix form, in which case the algorithm will simply copy the expression to the output string without performing any additional operations. This would result in a time complexity of O(n) and a space complexity of O(1). The worst-case scenario occurs when the input expression is fully parenthesized, in which case every operator will need to be pushed onto and popped from the stack, resulting in a time complexity of O(n) and a space complexity of O(n).

Infix to postfix conversion is suitable to use in situations where we want to evaluate expressions efficiently by first converting them to postfix form and then evaluating them using a stack-based algorithm. This can be useful in applications where arithmetic expressions need to be evaluated repeatedly, such as in computer algebra systems, compilers, or calculators.

Other Use Cases

The shunting yard algorithm can also be applied to produce prefix notation (also known as Polish notation). To achieve this, we need to start from the end of the input string, work backwards, and reverse the output queue to make it an output stack. Furthermore, we should change the behavior of left and right parentheses, so that the left parenthesis should pop until it finds a right parenthesis. Additionally, we need to modify the associativity condition to right.

The Shunting Yard algorithm can also be used to generate an Abstract Syntax Tree (AST), which represents the syntactic structure of an expression or program in a tree-like data structure. To build the AST, we modify the algorithm to create new AST nodes and add them to a stack instead of pushing operators onto the output queue. This results in a more structured representation of the input expression than the postfix expression. The AST can be used to evaluate the expression or to generate code in a target language.

Generating an Abstract Syntax Tree (AST) from an infix expression using the Shunting Yard algorithm has several practical applications in computer science and software development. Here are some of the important benefits:

1. Efficient evaluation: By representing an expression in the form of an AST, we can evaluate it more efficiently than by using postfix notation. This is because the AST provides a more structured representation of the expression, which allows us to traverse and evaluate it more efficiently.

2. Code generation: ASTs can be used to generate code in a target programming language. By traversing the AST, we can generate code that closely resembles the original expression, making it easier to translate expressions between different programming languages.
3. Compiler construction: ASTs are a fundamental component of compilers and interpreters. The syntax tree represents the structure of the program and can be used to verify its correctness, perform optimizations, and generate executable code.
4. Debugging: ASTs can also be useful for debugging programs. By analyzing the tree structure of an expression, we can identify errors in the syntax and semantics of the program more easily, and locate the source of the problem.

III. Conversion from Infix to Postfix without Stacks

Algorithm

1. Define a function to determine the precedence of operators. Higher precedence operators will be evaluated before lower precedence operators. The precedence order is "^" > "*" and "/" > "+" and "-".
2. Define a function to check if parentheses in the expression are balanced. This function uses a stack to keep track of opening parentheses encountered in the expression, and when a closing parenthesis is encountered, it checks if the stack is empty or if the top of the stack contains an opening parenthesis. If the stack is empty or the top of the stack is not an opening parenthesis, the expression is not balanced.
3. Define a function to convert infix expression to postfix expression. This function uses two strings, "postfix" and "operators". It loops through each token in the expression and performs the following operations:

- If the token is a digit or letter, append it to "postfix".
- If the token is an operator, while the "operators" string is not empty and the precedence of the last operator in "operators" is greater than or equal to the precedence of the current operator, append the last operator in "operators" to "postfix" and remove it from "operators". Then append the current operator to "operators".
- If the token is an opening parenthesis, append it to "operators".
- If the token is a closing parenthesis, append operators from "operators" to "postfix" until an opening parenthesis is encountered. Then remove the opening parenthesis from "operators". At the end of the loop, append all remaining operators in "operators" to "postfix".

4. Define a function to evaluate a postfix expression. This function uses a stack to keep track of operands encountered in the expression. It loops through each token in the expression and performs the following operations:

- If the token is a digit, convert it to an integer and push it onto the stack.
- If the token is an operator, pop the top two operands from the stack, perform the operation specified by the operator on the operands, and push the result back onto the stack. At the end of the loop, if the stack contains more than one operand, the expression is invalid.

5. In the main function, define a vector of expressions to test. Loop through each expression in the vector and print the infix expression. Try to convert the infix expression to postfix and evaluate the postfix expression. If there is an error during either of these operations, print the error message. Otherwise, print the postfix expression and the result of the evaluation.

Algorithm Analysis

The time complexity of this algorithm for infix to postfix conversion is O(n), where n is the length of the input expression. This is because the algorithm makes a single pass through the input expression, and at each character, it performs a constant number of operations.

The space complexity of this algorithm is also O(n), where n is the length of the input expression. This is because the algorithm uses a string to store the postfix expression, and the length of the postfix expression is at most n. Additionally, the algorithm uses a string to store the operators, and the size of this string is also at most n.

The best-case time complexity is O(n), which occurs when the input expression contains no parentheses or operators of equal precedence.

The worst-case time complexity is also O(n), which occurs when the input expression contains many nested parentheses or many operators of equal precedence. In this case, the algorithm may need to perform a large number of operations to convert the expression to postfix form.

IV. Conversion from infix to Postfix using Stacks & Queues

Algorithm

1. Initialize an empty stack opStack to hold operators and an empty queue outputQueue to hold operands and operators in postfix notation.
2. Iterate through each character ch in the infix expression:
    a. If ch is an operand, enqueue it to outputQueue.

   b. If ch is a left parenthesis, push it to opStack.
   c. If ch is a right parenthesis, pop operators from opStack and enqueue them to outputQueue until a left parenthesis is encountered. Pop and discard the left parenthesis.
   d. If ch is an operator, repeatedly pop operators from opStack and enqueue them to outputQueue until an operator with lower precedence (or a left parenthesis) is encountered. Push the current operator to opStack.
3. After iterating through all characters, enqueue any remaining operators from opStack to outputQueue.
4. Concatenate the elements of outputQueue to form the postfix expression.
5. Return the postfix expression.

Algorithm Analysis

- Time complexity: The algorithm scans each character in the infix expression exactly once, so the time complexity is O(n), where n is the length of the infix expression. In addition, each operator is pushed and popped from the stack at most once, so the time complexity of stack operations is also O(n). Therefore, the overall time complexity is O(n).
- Space complexity: The algorithm uses a stack and a queue to store operators and operands. The maximum size of the stack is proportional to the number of left parentheses in the infix expression, which is O(n/2) in the worst case. The maximum size of the queue is proportional to the length of the infix expression, which is O(n). Therefore, the overall space complexity is O(n).

In terms of best and worst-case scenarios, the time complexity of the algorithm is the same for all cases, since each character in the infix expression must be scanned at least once. However, the space complexity can vary depending on the input. The worst-case scenario for space complexity is when the infix expression contains many nested parentheses, since this will result in a larger stack. In contrast, the best-case scenario is when the infix expression contains no parentheses, since this will result in a smaller stack.

V. Recursive Descent Parsing

Recursive descent parsing is a type of top-down parsing algorithm used to analyze the syntactic structure of a program or expression, particularly in computer programming languages. It is a widely used technique for building parsers for programming languages and other formal grammars.

History and context:

Recursive descent parsing was first introduced in the late 1950s by American computer scientist Daniel McCracken, who described a method of parsing arithmetic expressions recursively using a simple grammar. This method was further developed in the 1960s by other researchers, including Niklaus Wirth and Tony Hoare, who used it to design compilers for the Algol programming language. Since then, recursive descent parsing has become a standard technique for building parsers for programming languages and other formal grammars.

Initial use case:

Recursive descent parsing was initially used in the design of compilers for programming languages, particularly in the development of Algol compilers. It was also used in the design of other programming languages, such as Pascal, which was designed by Niklaus Wirth and used recursive descent parsing to define its syntax.

Current use case:

Recursive descent parsing is still widely used in the design of compilers and interpreters for programming languages. It is particularly well-suited for programming languages with simple syntax and relatively small grammars. Recursive descent parsing is also used in other applications, such as natural language processing, where it is used to analyze the grammatical structure of sentences and texts.

Applications:

Recursive descent parsing can be used in a wide range of applications, including:

- Programming language compilers and interpreters
- Natural language processing
- Text processing and analysis
- Mathematical expression evaluation and parsing
- XML and HTML parsing
- Network protocol parsing

Other related algorithms:

There are several other parsing algorithms that are related to recursive descent parsing, including:

- LL parsing: A type of top-down parsing that uses a table-driven approach to parse a given input.

- LR parsing: A type of bottom-up parsing that uses a table-driven approach to parse a given input.
- Earley parsing: A type of chart parsing that uses dynamic programming to parse a given input.
- CYK parsing: A type of chart parsing that uses dynamic programming and context-free grammars to parse a given input.

Compared with these algorithms, recursive descent parsing is generally simpler to implement and requires less memory, but it is not as powerful and flexible as some of the other algorithms, particularly for complex grammars.

Algorithm

1. Define two functions isOperator and isOperand to check if a given character is an operator or an operand, respectively.
2. Define a function performOperation to perform the operation given an operator and two operands.
3. Define a recursive function convertToPostfix that takes the algebraic expression and an index i as input.
4. Inside the convertToPostfix function, create an empty stack s and an empty string postfix.
5. Loop through the expression starting from index i to the end:
    a. If the current character is an operand, append it to postfix.
    b. If the current character is an operator, pop all the operators with higher or equal precedence from the top of the stack and append them to postfix. Then push the current operator onto the stack.
    c. If the current character is an opening parenthesis, push it onto the stack and recursively call the convertToPostfix function with an incremented index i.
    d. If the current character is a closing parenthesis, pop all the operators from the top of the stack and append them to postfix until an opening parenthesis is found. Then pop the opening parenthesis.
    e. Increment the index i.
6. After the loop, pop all the remaining operators from the top of the stack and append them to postfix.
7. Return postfix.
8. Define a function evaluatePostfix that takes the postfix notation as input.
9. Inside the evaluatePostfix function, create an empty stack s.
10. Loop through the postfix notation:
    a. If the current character is an operand, push it onto the stack.
    b. If the current character is an operator, pop the top two operands from the stack, perform the operation, and push the result back onto the stack.

11. After the loop, the result of the expression is the top element of the stack. Return it.
12. In the main function, read the algebraic expression from the user, call the convertToPostfix function to convert it to postfix notation, call the evaluatePostfix function to evaluate the postfix notation, and display the result.

Algorithm Analysis

Time complexity:

- The **convertToPostfix** function iterates through the entire input expression once. At each iteration, it performs some stack operations and character comparisons. Therefore, the time complexity of this function is O(n), where n is the length of the input expression.
- The **evaluatePostfix** function iterates through the entire postfix notation once. At each iteration, it performs some stack operations and arithmetic calculations. Therefore, the time complexity of this function is also O(n), where n is the length of the postfix notation.

Space complexity:

- The **convertToPostfix** function uses a stack to hold operators and parentheses. The maximum size of the stack is proportional to the number of opening parentheses in the expression. Therefore, the space complexity of this function is O(p), where p is the number of opening parentheses in the expression.
- The **evaluatePostfix** function uses a stack to hold operands and intermediate results. The maximum size of the stack is proportional to the length of the postfix notation. Therefore, the space complexity of this function is O(n), where n is the length of the postfix notation.

Best case scenario:

- The best case scenario for this code is when the input expression is a single operand, with no operators or parentheses. In this case, the **convertToPostfix** function will simply return the operand as a postfix notation, and the **evaluatePostfix** function will return the operand itself. The time complexity for this case is O(1), and the space complexity is O(1).

Worst case scenario:

- The worst case scenario for this code is when the input expression is a fully parenthesized expression with many nested parentheses, such as "(a+(b*(c-d)))/(e+f)", where each operator has the highest precedence. In this case, the **convertToPostfix** function will need to handle all the nested parentheses, and the **evaluatePostfix** function will need to perform many arithmetic calculations. The time complexity for this

case is O(n), and the space complexity is O(p) or O(n), depending on whether there are more parentheses or operands and operators in the expression.

Summary

Name of Algorithm	Data Structure	Time & Space Complexity
Parenthesizing Infix	Stack	Time Complexity: O(n) Space Complexity: O(n)
RPN with Stacks	Stack, String	Time Complexity: O(n) Space Complexity: O(n)
RPN without Stacks	String, Stack, Vector	Time Complexity: O(n) Space Complexity: O(n)
RPN with Stack & Queue	A Stack & Queue	Time Complexity: O(n) Space Complexity: O(n)
Recursive Descent Parsing	Two Stacks	Time Complexity: O(n) Space Complexity: O(n)

About *RPN Calculators*

Figure 2: HP-32SII RPN Calculator

The HP 32SII calculator is a scientific calculator produced by Hewlett-Packard (HP) from 1991 to 2003. It is a successor to the HP 32S and HP 32SII models, and it features a two-line display, an extensive set of scientific and mathematical functions, and a sturdy, high-quality construction.

The calculator has a wide range of features including trigonometric and logarithmic functions, fractions and percentages, statistics, complex numbers, and programming capabilities. It also has a large user memory that can store up to 384 program steps or data registers. The programming language used by the calculator is RPN (Reverse Polish Notation), which requires the user to enter operators after operands. So, no need to convert from Infix to Postfix. I suspect this is because HP used under-powered hardware at the cost of human efforts to convert infix to postfix notations. It made the humans do the hard work.

The HP 32SII has been praised for its ease of use, reliability, and accuracy, and it remains a popular choice among scientists, engineers, and students. Its sturdy construction, along with its wide range of functions and programmability, make it a versatile tool for a variety of applications.

In recent years, the HP 32SII has become a collector's item, with enthusiasts seeking out original, unopened models and older, well-used models. Its enduring popularity is a testament to its quality and usefulness as a scientific calculator.

A basic RPN Calculator implementation in C++

This *RPN calculator* program was developed as an attempt to replicate the functionality of a traditional RPN calculator. A basic RPN calculator program that evaluates arithmetic expressions entered by the user in Reverse Polish Notation. It supports basic arithmetic operations such as addition, subtraction, multiplication, and division, as well as more advanced functions such as square root, logarithm, sine, cosine, and tangent.

The program uses two stacks: one for operands and one for operators. The algorithm works by iterating over the input expression one token at a time, and if the token is an operand, it is pushed onto the operand stack. If the token is an operator, the program checks the operator stack to see if there is an operator of higher precedence, and if so, it pops the operator from the operator stack, pops the top two operands from the operand stack, evaluates the expression, and pushes the result back onto the operand stack. This process continues until there are no more operators on the operator stack of higher precedence than the current operator, and then the current operator is pushed onto the operator stack.

In addition to basic arithmetic operators and functions, the program also supports some additional operations such as taking the factorial of a number and calculating the modulus of two numbers. The program also includes error handling for some operations, such as division by zero and taking the square root of a negative number.

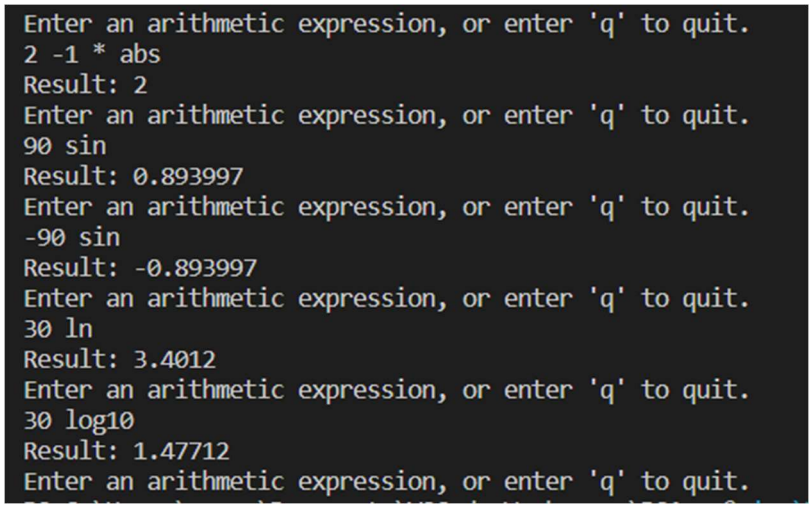

Figure 3: Program Output [Note the RPN notation]

Conclusion:

Infix to postfix conversion is a well-established algorithm in computer science and has been around for many years. While there may not be groundbreaking developments in terms of the fundamental algorithm itself, there have been improvements in terms of optimization, performance, and implementation techniques. Additionally, there have been efforts to apply this algorithm to different areas, such as compiler design and parsing, which have led to advancements in these fields. Overall, while there may not be significant recent developments in the algorithm itself, there have been advancements in related areas and techniques that have improved its practical applications.

It can be safely concluded that the Shunting Yard algorithm is a widely used parsing algorithm that is used to convert infix notation mathematical expressions to postfix notation. The algorithm is efficient, with a time complexity of O(n) and a space complexity of O(n), making it suitable for use in applications where arithmetic expressions need to be evaluated repeatedly, such as in computer algebra systems, compilers, or calculators.

Furthermore, the Shunting Yard algorithm can be modified to produce prefix notation or generate an Abstract Syntax Tree (AST), which has several practical applications in computer science and software development. By representing expressions in the form of an AST, they can be evaluated more efficiently, and code can be generated in a target programming language. ASTs are also a fundamental component of compilers and interpreters and can be useful for debugging programs.

In conclusion, the Shunting Yard algorithm is a powerful tool in the field of computer science that has many practical applications. Its efficiency and flexibility make it a valuable asset in a wide range of industries, from software development to scientific research.

Infix to postfix conversion is an essential technique in computer science and is unlikely to become obsolete in the near future. As computer hardware continues to advance, there may be opportunities for faster and more efficient algorithms for infix to postfix conversion.

One potential hardware implication is that as processors continue to become faster, it may become feasible to use more computationally expensive algorithms for infix to postfix conversion, which could result in more efficient and optimized code.

In terms of software, there may be opportunities for new programming languages or frameworks to incorporate infix to postfix conversion as a built-in feature or library. Additionally, advancements in artificial intelligence and machine learning could potentially lead to more sophisticated algorithms for infix to postfix conversion, which could have a range of practical applications in fields such as natural language processing and computer vision.

References:

https://www.andrew.cmu.edu/course/15-200/s06/applications/ln/junk.html#_CONVERSION

https://www.sciencedirect.com/science/article/abs/pii/0167819187900287