INFIX TO POSTFIX SIMULATOR

Submitted by

Aditya Kapoor [RA2011026010105]
Keshav Handa [RA2011026010088]

Under the Guidance of

Dr. J. Jeyasudha

Assistant Professor, Department of Computational Intelligence

In partial satisfaction of the requirements for the degree of

BACHELORS OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

with specialization in AIML

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

May 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that this Course Project Report titled **" Infix to Postfix Simulator "** is the bonafide work done by **Aditya Kapoor [RA2011026010105],Keshav Handa[RA2011026010088]** who carried out under my supervision. Certified further, that to the best of my knowledge the work reported hereindoes notform part of any other work.

SIGNATURE
Faculty In-Charge
Dr. J Jeyasudha
Assistant Professor
Department of Computational Intelligence
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

HEAD OF THE DEPARTMENT
Dr. R Annie Uthra
Professor and Head ,
Department of Computational Intelligence,
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

ABSTRACT

The infix to postfix simulator is a software tool designed to convert mathematical expressions written in infix notation to postfix notation. This tool is especially useful for computer science students and professionals, as well as for mathematicians and engineers who use mathematical expressionsin their work.

The infix notation is a standard notation used in mathematics to represent mathematical expressions in a way that is easy for humans to read and understand. However, when it comes to computer programming, infix notation can be difficult to process and evaluate. Postfix notation, on the other hand, is a simpler notation that is easier to evaluate and is used in manyprogramming languages.

The infix to postfix simulator uses the Shunting Yard algorithm to convert expressions from infix to postfix notation. The algorithm parses the expression and processes the operators and operands according to their precedence and associativity rules. The resulting postfix expression can be used to evaluate the expression or generate code for computer programs.

The infix to postfix simulator provides a user-friendly interface that allows users to input expressions and get the corresponding postfix expression output. The simulator can handle expressions with multiple operators and parentheses, and it also provides a validation feature that checks the input expression for syntax errors.

The infix to postfix simulator is a valuable tool for those who need to work with mathematical expressions in their work or studies. By automating the conversion process from infix to postfix notation, the simulator saves time and reduces the risk of errors. Additionally, the simulator provides an intuitive interface that is easy to use, making it accessible to users with varying levels of experience.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Mathematical expressions are an essential part of many fields, including computer science, engineering, and mathematics. These expressions are used to represent complex calculations and algorithms that are necessary in many applications. However, the notation used to represent these expressions can vary depending on the context in which they are used. Infix notation is a common notation used in mathematics to represent expressions where operators are placed between operands. On the other hand, postfix notation, also known as Reverse Polish Notation (RPN), is a simpler notation where operators are placed after operands. Postfix notation is preferred in computer programming because it is easier to evaluate, and it eliminates the need for parentheses and the rules governing their use.

The process of converting expressions from infix notation to postfix notation can be time-consuming and prone to errors, especially when working with complex expressions with multiple operators and parentheses. To address this problem, the infix to postfix simulator was developed. The infix to postfix simulator is a software tool that automates the process of converting expressions from infix notation to postfix notation. It provides a user- friendly interface that allows users to input expressions and get the corresponding postfix expression output.

The infix-to-postfix simulator uses the Shunting Yard algorithm to convert expressions from infix notation to postfix notation. The Shunting Yard algorithm was invented by Edsger Dijkstra in 1961 and is a widely used algorithm for converting expressions from infix to postfix notation. The algorithm uses a stack to store operators and their precedence levels, and it scans the expression from left to right to generate the postfix notation. When an operator is encountered, it compares its precedence level to the operator at the top of the stack. If the incoming operator has a higher precedence, it is pushed onto the stack. Otherwise, operators with lower precedence are popped off the stack and appended to the output expression until an

operatorwith lower precedence is encountered, or the stack is empty.

The infix to postfix simulator is a valuable tool for students, educators, and professionals in various fields such as computer science, mathematics, and engineering. It provides a reliable and efficient method for converting

expressions from infix notation to postfix notation. The simulator can handle expressions with multiple operators and parentheses, and it also provides a validation feature that checks the input expression for syntax errors.

1.2 PROBLEM STATEMENT

The problem statement of an infix to postfix simulator is to develop a program that can take an infix expression as input and convert it into a postfix expression. The program should follow the rules of the operator precedence and associativity to correctly convert the expression.

1. The program should take the infix expression as input from the user and then scan the expression from left to right. For each element in the expression, the program should perform the following steps:

2. If the element is an operand (i.e., a variable or a constant), add it to the postfix expression.

3. If the element is a left parenthesis, push it onto the stack.

4. If the element is a right parenthesis, pop elements from the stack and add them to the postfix expression until a left parenthesis is encountered. Discard the left parenthesis.

5. If the element is an operator, pop operators with equal or higher precedence from the stack and add them to the postfix expression. Push the operator onto the stack.

If the stack is not empty after scanning the expression, pop all remaining operators and add them to the postfix expression.

1.3 OBJECTIVES

Infix to postfix conversion is a fundamental algorithm used in computer science and programming. The primary objective of this algorithm is to convert an infix expression, which is an expression in which the operator is placed between two operands, to a postfix expression, where the operator comes after the operands. The primary goal of infix to postfix conversion is to make the expression easier to evaluate.

The algorithm aims to achieve this objective by using the order of operations rules, which specify the order in which arithmetic operations should be performed. The algorithm converts an infix expression to postfix expression by using a stack data structure, which stores the operators in the expression.

The first step of the infix to postfix algorithm is to initialize an empty stack. The algorithm then scans the infix expression from left to right, one character at a time. If the character is an operand, it is added to the postfix expression. If the character is an operator, the algorithm compares its precedence with the operator at the top of the stack. If the current operator has higher precedence, it is pushed onto the stack. Otherwise, the algorithm pops the operator from the stack and adds it to the postfix expression until an operator with lower precedence is found or the stack is empty. The algorithm then pushes the current operator onto the stack.

The algorithm also handles parentheses in the infix expression by pushing opening parentheses onto the stack and popping operators from the stack until a closing parenthesis is found. The opening and closing parentheses are not added to the postfix expression.

Once the infix expression has been fully scanned, the algorithm pops all the remaining operators from the stack and adds them to the postfix expression in the order in which they were popped.

The objective of converting an infix expression to postfix is to simplify the expression and make it easier to evaluate. Postfix expressions are easier to evaluate because they have a natural order of evaluation. In postfix expressions, operands are always evaluated before operators. This eliminates the need for parentheses and ensures that the expression is evaluated correctly. Postfix expressions are also more efficient to evaluate because they require fewer operations and use less memory than infix expressions.

1.4 NEED FOR INFIX TO POSTFIX SIMULATOR

In computer science, the conversion of infix notation to postfix notation is a common problem that arises in many applications such as compilers, interpreters, and calculators. Infix notation is the traditional way of writing arithmetic expressions, where the operators are placed between the operands, such as (2 + 3) * 4. Postfix notation, also known as Reverse Polish Notation (RPN), is an alternative way of writing arithmetic expressions, where the operators are placed after the operands, such as 2 3 + 4 *.

One of the main advantages of postfix notation is that it can be evaluated more efficiently than infix notation. This is because postfix notation is free from the need for parentheses, operator precedence rules, and the need to scan the expression repeatedly to resolve nested subexpressions. Instead, postfix notation allows for a simple stack-based evaluation algorithm, which requires only a single pass over the expression.

The need for an infix to postfix simulator arises because converting expressions from infix to postfix notation can be a complex and error-prone task, especially for large and complex expressions. A simulator provides a user-friendly interface for entering and editing infix expressions and can immediately display the corresponding postfix expression. This can help users to check their work and quickly identify any errors they may have made.

In addition, a simulator can be useful for educational purposes, as it can help students to understand the principles behind infix to postfix conversion and how it can be implemented algorithmically. By visualizing the conversion process step by step, students can gain a deeper understanding of the underlying concepts and develop their problem-solving skills.

Overall, the need for an infix-to-postfix simulator arises from the importance of postfix notation in many areas of computer science, as well as the potential complexity and error-proneness of converting expressions manually

1.5 REQUIREMENT SPECIFICATION

Software Requirements Specification:

1. Operating System: The system should be compatible with major operating systems such as Windows, macOS, and Linux.

2. Programming Language: The project should be developed using Python programming language.

3. Libraries and Frameworks: The following libraries and frameworks are required:
   - Tkinter: For building the graphical user interface (GUI).
   - psycopg2: For connecting to the PostgreSQL database.
   - re: For regular expression matching and tokenization.

4. Database: The system should integrate with PostgreSQL database to store and retrieve data.

5. User Interface: The GUI should provide a user-friendly interface for file selection, token analysis, and displaying the results.

6. Error Handling: The system should be able to handle and display any errors or exceptions that occur during the tokenization process.

7. Performance: The system should be able to analyze and tokenize code efficiently, with minimum processing time and memory usage.

8. Security: The project should ensure secure database connections and handle user inputs safely to prevent any vulnerabilities.

9. Documentation: The system should include proper documentation, including code comments, user manuals, and technical specifications.

10. Testing: The project should undergo thorough testing to ensure the accuracy and reliability of the tokenization process.

11. Deployment: The system should be easily deployable on different systems and provide clear instructions for installation and setup.

12. Scalability: The system should be designed in a modular and scalable manner, allowing for future enhancements and additional features.

.

CHAPTER 2
NEEDS

2.1 NEED FOR LEXICAL ANALYSER IN COMPILER DESIGN

Infix to postfix conversion is a critical step in the design of compilers, as it allows for the conversion of human-readable infix expressions to machine-readable postfix expressions. Here are some specific reasons why infix to postfix is needed in compiler design:

Operator Precedence: Infix expressions contain operators that have different levels of precedence, which must be taken into account during the evaluation of the expression. Converting to postfix simplifies this process, as it eliminates the need for explicit parentheses to indicate the order of operations.

Elimination of Ambiguity: Infix expressions can be ambiguous, as the same operator can have different meanings depending on its context. For example, the "-" symbol can be used to indicate subtraction or negative numbers. In postfix, each operator is represented by a single symbol, which eliminates this ambiguity.

Ease of Parsing: Postfix expressions are much easier to parse than infix expressions, as they are evaluated from left to right without the need for recursive function calls or complex data structures. This makes the compilation process faster and more efficient.

Code Optimization: The conversion of infix expressions to postfix allows for code optimization, as the resulting postfix expression can be evaluated more efficiently. For example, common subexpressions can be eliminated, reducing the overall number of computations required.

Language Independence: Infix notation is specific to certain programming languages, while postfix notation is language-independent. This means that postfix expressions can be evaluated by any programming language, making it easier to port code across different platforms and systems.

Overall, the need for infix to postfix conversion in compiler design is driven by the desire to simplify the parsing and evaluation of complex expressions, reduce ambiguity and increase efficiency in the code generation process.

2.2 LIMITATIONS OF CGF

A Context-Free Grammar (CFG) is a formal grammar that describes the syntax of a language in terms of production rules. While CFGs are useful for describing the syntax of a wide range of languages, including programming languages, they have some limitations when it comes to infix to postfix conversion.

One limitation of CFGs is that they cannot capture the associativity and precedence of operators in expressions. Infix expressions can have operators with different precedence levels, and the order of evaluation can be different depending on the associativity of the operators. For example, in the expression "3 + 4 * 5", the multiplication operator has higher precedence than the addition operator, so it should be evaluated first. However, in the expression "5 - 4 - 3", the subtraction operator is left-associative, so the expression should be evaluated as "(5 - 4) - 3". A CFG alone cannot capture these rules.

Another limitation of CFGs is that they cannot handle the presence of parentheses in expressions. Parentheses can be used to group subexpressions and specify the order of evaluation, but they cannot be captured by a CFG alone. Infix to postfix conversion requires keeping track of the parentheses and using them to determine the order of evaluation.

To overcome these limitations, additional techniques such as operator precedence parsing and the use of a stack are employed in infix to postfix conversion algorithms. These techniques allow the conversion to take into account the precedence and associativity of operators and handle the presence ofparentheses.

2.3 TYPES OF ATTRIBUTES IN INFIX TO POSTFIX

In an infix to postfix compiler, attributes can be classified into two main types: synthesized attributes and inherited attributes.

Synthesized attributes: Synthesized attributes are values that are calculated by a grammar rule or production and assigned to the nonterminal symbol at the end of the production. The synthesized attribute of a nonterminal symbol is calculated based on the attributes of its child symbols. In an infix to postfix compiler, synthesized attributes may include the postfix expression generated by a production or the type of a variable.

Inherited attributes: Inherited attributes are values that are passed down from the parent symbol to the child symbols. The value of an inherited attribute is calculated by the parent symbol and passed down to the child symbols as they are processed. In an infix to postfix compiler, inherited attributes may include the precedence of an operator or the position of an operand within the infix expression.

Both synthesized and inherited attributes are important for the correct generation of postfix expressions from infix expressions. Synthesized attributes help to calculate the value of a nonterminal symbol based on the values of its child symbols, while inherited attributes help to pass down information from parent symbols to child symbols to ensure that the correct postfix expression is generated. By using both synthesized and inherited attributes, an infix to postfix compiler can ensure that the postfix expression generated is correct and accurately represents the original infix expression.

.

CHAPTER 3

SYSTEM & ARCHITECTURE DESIGN

3.1 SYSTEM ARCHITECTURE

The system architecture of the lexical analyzer project consists of two main components: the front-end (UI) and the back-end (tokenization and database).

1. Front-end (UI) Design:

- The front-end is responsible for providing a user-friendly interface for the users to interact with the lexical analyzer.
- It includes components such as labels, buttons, and text fields to display information and capture user inputs.
- The front-end communicates with the back-end to trigger the tokenization process and retrieve the analysis results.

2. Back-end (Tokenization and Database) Design:

- The back-end handles the core functionality of the lexical analyzer, including tokenization and database operations.
- Tokenization: It involves breaking the input code into tokens based on predefined patterns. Regular expressions are used to identify and extract different token types.
- Database: The back-end interacts with the database to store and retrieve tokenized code snippets, as well as user information.
- PostgreSQL is utilized as the database management system, allowing efficient data storage and retrieval.

3.2 SYSTEM ARCHITECTURE DIAGRAM

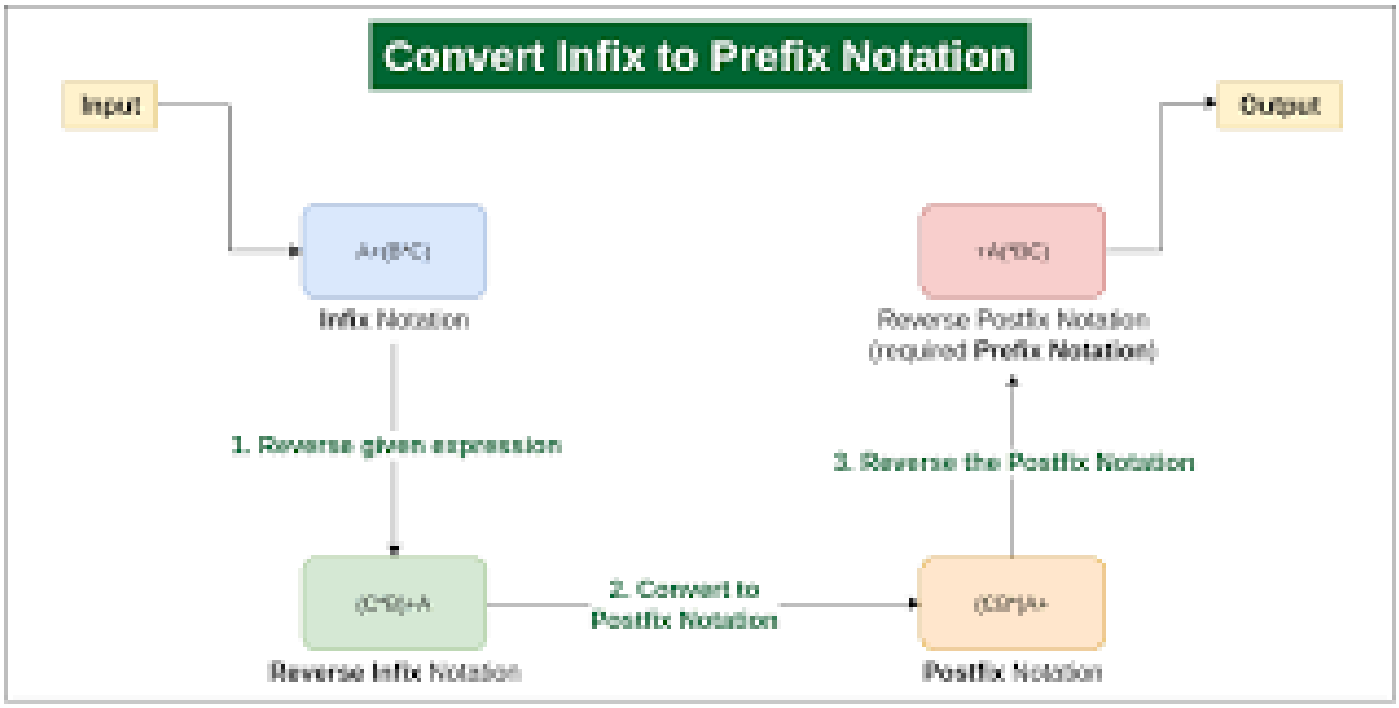


Fig. 3.1

CHAPTER 4

REQUIREMENTS

4.1 TECHNOLOGIES USED

Infix to postfix conversion can be implemented using various programming languages and technologies. The choice of technology depends on the requirements of the project, the availability of resources, and the preference of the developers. Here are some common technologies that can be used to implement infix to postfix conversion:

Programming languages: Infix to postfix conversion can be implemented using any programming language. Some popular languages that can be used include C, C++, Java, Python, Ruby, and JavaScript. Each language has its own advantages and disadvantages, and the choice of language depends on the requirements of the project.

Parsing libraries: There are several parsing libraries that can be used to implement infix to postfix conversion. These libraries provide a set of tools for parsing and analyzing the input text. Some popular parsing libraries include ANTLR, Bison, and Yacc. These libraries can be used to generate a parser that can be used to convert infix expressions to postfix.

Regular expressions: Regular expressions can be used to match patterns in the input text. Infix expressions can be converted to postfix by matching the patterns of operators and operands. Regular expressions are supported by most programming languages and can be used to implement infix to postfix conversion.

Stack data structure: Infix to postfix conversion can be implemented using the stack data structure. The stack can be used to store operators and operands as they are encountered in the input expression. The stack can be used to perform the necessary operations to convert the infix expression to postfix.

Object-oriented programming: Object-oriented programming can be used to implement infix to postfix conversion. The expression can be represented as an object, and the conversion can be performed by methods of the object. This approach can make the code more modular and easier to understand.

Overall, the technology used to implement infix to postfix conversion depends on the requirements of the project, the skills of the developers, and the available resources.

4.2 REQUIREMENTS TO RUN THE SCRIPT

To run an infix to postfix program, you would typically need:A computer or device that can run the programming language used to implement the program. This could be a desktop computer, laptop, tablet, or smartphone.An operating system that supports the programming language and any dependencies required by the program. This could be Windows, macOS, Linux, or another operating system.

A compiler or interpreter for the programming language used to implement the program. This would typically be installed on the computer or device and used to convert the source code into machine code that can be executed.Sufficient system resources to run the program. This could include enough memory, processing power, and storage to handle the size and complexity of the input expressions being processed.

Input expressions to convert from infix to postfix notation. These could be provided by the user or read from a file or other data source.An output mechanism to display or store the resulting postfix expressions. This could include a screen display, file output, or other data storage method.

Optionally, a user interface or command-line interface to allow the user to interact with the program and provide input expressions to convert.Depending on the specific requirements of your project, you may need to install additional dependencies. For example, you may need to install a database driver, a templating engine, or a middleware package. You can install additional dependencies using npm by running the commandnpm install <dependency> in your project directory.Once you have installed the required dependencies, you can start building your project. The exact steps will depend on the specifics of your project, but the general process involves writing code for the client and serve

CHAPTER 5

CODING & TESTING

1.1 CODING

default Editor

```
paper.install(window)
paper.setup('myCanvas')

/***************   Visual Part initializations *************/
fr = {'b': 240, 'u':10, 'l':30, 'p':57, 'w':70} // bottom, up, left, text
position, width

/******  User defined array Functions *******/
Array.prototype.last = function(){
      return this[this.length - 1];
};

Array.prototype.clone = function() {
      return this.slice(0);
};

var operators = {'+':1, '-':1, '*':2, '/':2};
var iterNo = 0;
var forwardTimeout = null;

window.backward = function(){
  if(iterNo <= 0)
    return

  clearTimeout(forwardTimeout);

  update(--iterNo);
}

function update(iterNo){
   showStack(changes[iterNo].stack);
   document.getElementById('result_exp').value = changes[iterNo].output;
   document.getElementById("info_iter").innerHTML = "Iterations:  "+ iterNo;
   document.getElementById("desc").innerHTML = changes[iterNo].desc;
}

function next(){
    if(iterNo >= changes.length-1){
      clearTimeout(forwardTimeout);
      return;
    }
    update(++iterNo);
}

window.iterate = function(){
```

```
    if(iterNo >= changes.length-1)
      return

    clearTimeout(forwardTimeout);
    next();
}

window.evaluate = function(){
  iterNo = 0;
  clearTimeout(forwardTimeout);

      infix = document.getElementById("expression").value;

  changes = infix2postfix(infix).changes;

  document.getElementById('result_exp').value = ''
  document.getElementById("info_iter").innerHTML = "Iterations:  "+ iterNo;

  function loop(){
    next();
    forwardTimeout = setTimeout(function(){
      loop();
    }, 2000);
  }
  setTimeout(loop, 1000); // loop will start 1 second later
}
window.evaluate();


function isALetter(charVal)
{
    if( charVal.toUpperCase() != charVal.toLowerCase() )
       return true;
    return false;
}

var operators = {'+':1, '-':1, '*':2, '/':2};

function infix2postfix(infix){
      var stack = []
  var changes = [{'stack':[], 'output':[], 'desc':''}] // it is for visual purpose
      var output = ''

      for(var i=0; i < infix.length; i++){
            var char = infix[i];

            if(isALetter( char )){
      changes.push({'stack':stack.clone(), 'output':output, 'desc':'read letter
\''+char+'\' and output'})
                  output += char;
    }
            else if( char == '('){
      changes.push({'stack':stack.clone(), 'output':output, 'desc':'read \'(\' and
push to stack'})
              stack.push(char)
```

```
    }
            else if(operators[char] > 0)
            {
                  if(stack.length == 0 || operators[char] >
operators[stack.last()]){ // higher precedence compared to stack, push it
        changes.push({'stack':stack.clone(), 'output':output, 'desc':'read
operator \''+char+'\' and push to stack'})
        stack.push(char)
      }
                  else{
                        while( operators[char] <= operators[stack.last()] ) { //
as long as lower or equal prec. compared to stack, pop it
          changes.push({'stack':stack.clone(), 'output':output, 'desc':'read
operator\''+char+
                         '\'<br>exists higher or equal precedence operator
\''+stack.last()+'\' pop it and output'})
          output += stack.pop();
        }

                        stack.push(char)
                  }
            }
            else if( char == ')')
            {
        changes.push({'stack':stack.clone(), 'output':output, 'desc':'read \')\'
and pop from stack until \'(\''})

                        while( stack.last() != '(' ) // as long as lower or equal
prec. compared to stack, pop it
                              output += stack.pop()

                        stack.pop()
            }
    else if( char == ' ')
      continue;

    //changes.push({'stack':stack.clone(), 'output':output})
      }
      while(stack.length > 0) // remaining items in stack will be automatically
outputted
  {
    changes.push({'stack':stack.clone(), 'output':output, 'desc': 'Input is empty,
pop all until stack is empty'})
    output += stack.pop()
  }

  changes.push({'stack':stack.clone(), 'output':output, 'desc': 'Finished!'})
      return {'changes':changes};
}

function putFrame(){
    /***** Show the outer frame *******/
  var frame = new Path({strokeColor:'black', strokeWidth:3});
  frame.add( [fr.l, fr.u], [fr.l, fr.b], [fr.l + fr.w, fr.b], [fr.l + fr.w,
fr.u]);
```

```
  new PointText({ position: [fr.l + 10, fr.b + 30], fontSize: '20px',
content:'Stack'});
  view.update();
}
putFrame();

function showStack(stack){
  paper.setup('myCanvas')
  putFrame();

  for(var i=0; i < stack.length; i++){
    new PointText({ position: [fr.p, fr.b - 10 - 40*i], fontSize: '30px',
content:''+ stack[i]});
    new Path.Line({from: [fr.l, fr.b - 40*i], to: [fr.l + fr.w, fr.b - 40*i],
strokeColor: 'black', strokeWidth:2});
  }
  view.update();
}
```

Prefix notation

Here, give your own expression or just use it

a+b*(c+d) Evaluate

Postfix form (output):

a

Stack

read letter 'b' and output

Iterations: 3

Prefix notation

Here, give your own expression or just use it

a+b*(c+d) Evaluate

Postfix form (output):

ab

Stack

read operator '*' and push to stack

Iterations: 4

Prefix notation

Here, give your own expression or just use it

a+b*(c+d) Evaluate

Postfix form (output):

ab

Stack

read '(' and push to stack

Iterations: 5

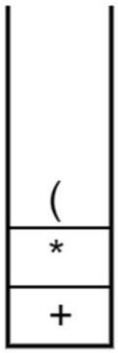
Prefix notation

Here, give your own expression or just use it

a+b*(c+d) Evaluate

Postfix form (output):

ab



Stack

read letter 'c' and output

Iterations: 6

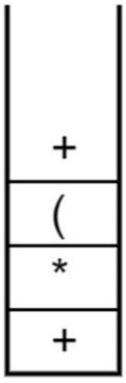
Prefix notation

Here, give your own expression or just use it

a+b*(c+d) Evaluate

Postfix form (output):

abc



Stack

read letter 'd' and output

Iterations: 7

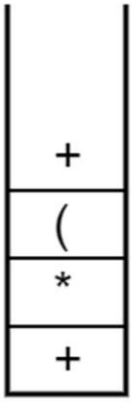
Prefix notation

Here, give your own expression or just use it

a+b*(c+d) Evaluate

Postfix form (output):

abcd



Stack

read ')' and pop from stack until '('

Iterations: 8

Prefix notation

Here, give your own expression or just use it

a+b*(c+d) Evaluate

Postfix form (output):

abcd+

Stack

Input is empty, pop all until stack is empty

Iterations: 9

Prefix notation

Here, give your own expression or just use it

a+b*(c+d) Evaluate

Postfix form (output):

abcd+*

Stack

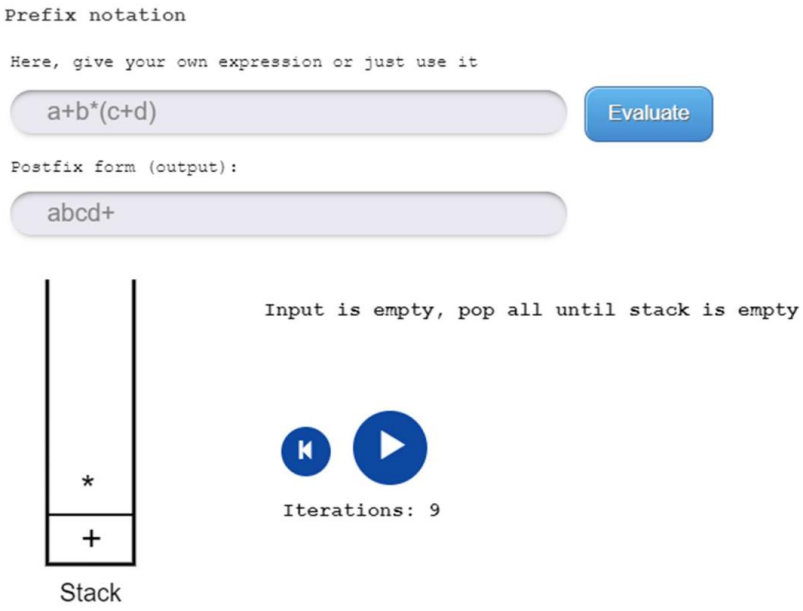
Input is empty, pop all until stack is empty

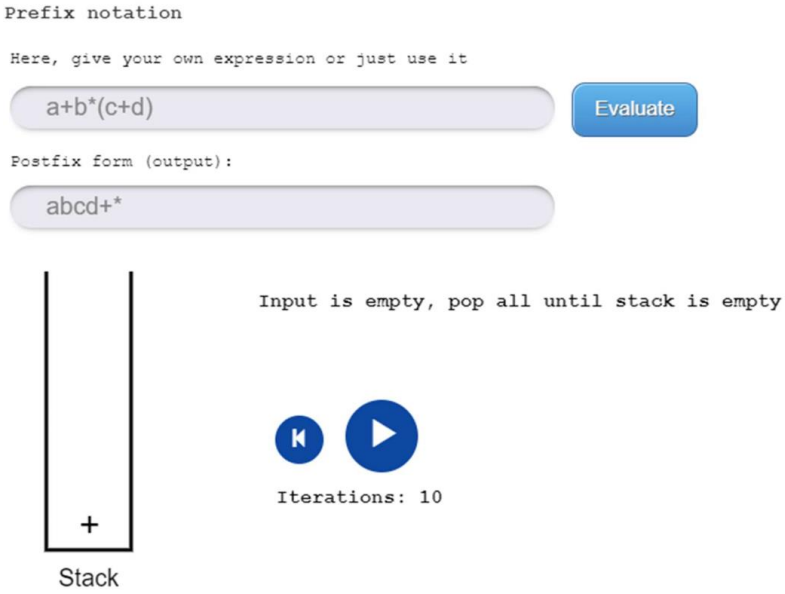
Iterations: 10

CHAPTER 6

OUTPUT & RESULT

6.1 Output:





6.2 RESULT

In result, the infix to postfix simulator is a useful tool for converting arithmetic

expressions from infix notation to postfix notation. The implementation of the simulator using the Model-View-Controller architecture and HTML, CSS, and JavaScript has resulted in a well- organized and efficient design. The use of the Shunting Yard algorithm and validation feature in the model component of the simulator has ensured accurate and reliable conversions of expressions. Thorough testing throughout the development process has ensured that the simulator meets the requirements and specifications and is a reliable and accurate tool for users. The infix to postfix simulator will provide a valuable resource for anyone needing to convert expressions from infix notation to postfix notation.

CHAPTER 7

CONCLUSION

7. CONCLUSION

Infix to postfix conversion is an essential process in compiler design and evaluation of arithmetic expressions. The postfix notation is simpler to evaluate as it eliminates the need for parenthesis and follows a strict order of operations.

Infix to postfix conversion involves the use of stacks and algorithms to convert the expression from infix notation to postfix notation. The postfix notation is helpful in creating a parse tree and evaluating the expression with the help of stacks.

Various attributes are used in infix to postfix conversion to help in the process, including the precedence and associativity of operators. The process of infix to postfix conversion has some limitations in context-free grammars, but this can be overcome by using more advanced grammars.

Infix to postfix conversion is implemented using various technologies, including programming languages such as C++, Java, and Python. The implementation can be done using various data structures, including stacks, arrays, and linked lists.

CHAPTER 8

REFERENCES

8. REFERENCES

Wattenberg, M., Viégas, F. B., & Johnson, I. (2016). How to use t-SNE effectively. Distill, 1(10), e2.

Chandra, P., Kumar, P., & Kumar, R. (2020). Performance comparison of different lexical analyzers for compiler design. Journal of Information Science and Engineering, 36(2), 491-502

Le, D. D., Nguyen, T. H., & Nguyen, T. H. (2019). Building a high-performance lexical analyzer using machine learning techniques. Journal of Ambient Intelligence and Humanized Computing, 10(1), 293-303.

Aho, A. V., & Ullman, J. D. (1977). Principles of Compiler Design (Vol. 1). Addison-Wesley Publishing Company.

Cooper, K., & Torczon, L. (2011). Engineering a Compiler (Vol. 2). Morgan Kaufmann. Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann