**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**FROM INFIX TO POSTFIX: IMPLEMENTING EXPRESSION CONVERSION IN COMPILER DESIGN**

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

**BACHELOR OF ENGINEERING**

**IN**

**COMPUTER SCIENCE ENGINEERING**

**Submitted by**

1. **JAITHRA(192210259)**
2. **MAHISREEJA(192210438)**
3. **CHERISHMA(192211840)**

**Under the Supervision of**

**Dr.G.MICHAEL**

MARCH (2024)

**DECLARATION**

We, **L. Jaithra, P. Mahisreeja, A. Cherishma ,** students of **‘Bachelor of Engineering in COMPUTER SCIRNCE’**, Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented **FROM INFIX TO POSTFIX: IMPLEMENTING EXPRESSION C ONVERSION IN COMPILER DESIGN** in this Capstone Project Work entitled is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

1. Jaithra 192210259)
2. Mahisreeja 192210438 )
3. Cherishma 192211840)

Date:

Place:

**CERTIFICATE**

This is to certify that the project entitled **“FROM INFIX TO POSTFIX: IMPLEMENTING EXPRESSION C ONVERSION IN COMPILER DESIGN”** submitted by L. Jaithra, P. Mahisreeja, A. Cherishma has been carried out under our supervision. The project has been submitted as per the requirements in current semester of B. Tech Information Technology.

Faulty-in-charge

Dr.G.MICHAEL

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| **S.NO** | **TOPICS** |
| 1 | **Abstract** |
| 2 | **Introduction** |
| 3 | **Problem Statement** |
| 4 | **Proposed Design**   1. Input Handling 2. Conversion Algorithm 3. Testing and Validation |
| 5. | **Functionality**   1. Operator Support 2. Integration with Compiler Framework 3. Documentation |
| 6 | **UI Design**   1. Input Section 2. Conversion Button 3. Accessibility |
| 7 | **Conclusion** |

**ABSTRACT**

Converting infix expressions to postfix expressions is a fundamental operation in computer science, essential for parsing mathematical expressions efficiently. This capstone project focuses on developing a software implementation of an algorithm to convert infix expressions to postfix form. The project encompasses several key steps and considerations to ensure robustness and versatility.

The primary objective of the project is to implement the selected algorithm in a programming language of choice. The implementation must effectively handle various types of infix expressions, including arithmetic operations, parentheses, and precedence rules. By accommodating different expression formats and operator priorities, the software aims to provide accurate and efficient conversion to postfix notation.

The project involves a systematic approach, starting with algorithm selection based on its suitability for infix to postfix conversion. Following this, the algorithm will be translated into executable code, incorporating error handling and optimization techniques to enhance performance and reliability. Rigorous testing methodologies will be employed to validate the software's functionality across diverse input scenarios and edge cases.

Furthermore, the project emphasizes documentation and clarity in code structure to facilitate understanding and future maintenance. Detailed documentation will outline the algorithm's principles, the rationale behind design decisions, and instructions for utilization.

In conclusion, this capstone project offers a comprehensive exploration of converting infix expressions to postfix notation through software implementation. By addressing various complexities and considerations inherent in the conversion process, the project aims to deliver a robust and adaptable solution applicable in diverse computational contexts.

**INTRODUCTION**

This capstone project focuses on developing a software implementation for converting infix expressions to postfix, a crucial operation in computational algorithms.The first phase involves the selection of a programming language suited to the project's requirements, balancing factors such as performance, familiarity, and library support. Once the language is chosen, a thorough understanding of infix notation and its associated rules becomes paramount, guiding subsequent design and implementation decisions. Extensive research on conversion algorithms follows, exploring methodologies like the shunting-yard algorithm and recursive descent parsing to determine the most suitable approach. With a clear understanding of the theoretical underpinnings, the project proceeds to design the software architecture, delineating key components such as expression parsing, operator handling, and output generation for efficient development and maintenance. The core implementation begins with parsing infix expressions into constituent tokens, requiring meticulous attention to detail to accommodate various input formats and error scenarios effectively.

The development of a software solution for converting infix expressions to postfix entails a structured approach encompassing research, design, implementation, testing, and optimization. By adhering to these principles, this capstone project endeavors to deliver a reliable and efficient tool capable of handling diverse expression types with ease, contributing to the advancement of computational algorithms and software development practices.

**PROBLEM STATEMENT**

In the face of increased difficulty in problem solving the project helps to streamline the process of parsing and evaluating mathematical expressions. In a compiler, converting infix expressions to postfix form facilitates efficient expression evaluation, aiding in subsequent stages of compilation such as syntax analysis and code generation. By converting infix expressions to postfix, the compiler can simplify expression parsing and reduce computational overhead. This conversion process enables compilers to effectively handle operator precedence, associativity, and parentheses, ensuring accurate interpretation of mathematical expressions. Ultimately, integrating infix to postfix conversion into compiler design enhances the efficiency and performance of the compilation process, leading to optimized code generation and improved overall compiler functionality.

**PROPOSED DESIGN**

**Input Handling**: Develop a module to accept infix expressions from user input or files.

**Conversion Algorithm**: Employ the Shunting Yard algorithm or a similar approach for converting infix expressions to postfix notation.

**Operator Support:** Support standard arithmetic operators (e.g., +, -, \*, /) and handle unary operators if necessary.

**Parentheses Handling:** Manage parentheses to enforce precedence and maintain the correct evaluation order of expressions.

**Output Generation**: Generate well-formatted postfix expressions as output, ready for further processing within the compiler pipeline.

**Testing and Validation:** Develop a comprehensive test suite comprising various infix expressions to validate the correctness and robustness of the conversion module. **Optimization and Performance:** Optimize the conversion algorithm for efficiency and scalability to handle expressions of varying complexity.

**FUNCTIONALITY**

**Conversion Algorithm:** Implement an efficient algorithm to convert infix expressions to postfix expressions. Ensure that the algorithm correctly handles operator precedence and associativity rules to maintain the semantics of the original expressions. Manage parentheses to enforce the correct order of operations during conversion.

**Operator Support:** Support standard arithmetic operators (e.g., addition, subtraction, multiplication, division) and any additional operators defined by the language grammar.

Handle unary operators, if applicable, according to the language specification.

**Output Generation:** Generate well-formatted postfix expressions as output, ready for further processing within the compiler pipeline.

**Integration with Compiler Framework:** Integrate the infix to postfix conversion module seamlessly into the compiler framework.

Ensure compatibility with other components of the compiler, such as lexical analysis, parsing, and code generation.

**Documentation:** Provide comprehensive documentation detailing the implementation approach, algorithms used, and integration steps.

**ARCHITECTURAL DESIGN**

**Module Pattern:** Each component (input, conversion, error handling, integration) is encapsulated as a separate module with well-defined interfaces. Promotes modularity and reusability of code.

**Layered Architecture:** Components are organized into layers based on their functionalities (input layer, conversion layer, error handling layer, integration layer). Supports separation of concerns and facilitates maintainability.

**UI DESIGN**

**Input Section:** Provide a text area or input field where users can enter infix expressions.

**Conversion Button:** Add a button labelled "Convert" or "Convert to Postfix" to initiate the conversion process. Ensure that the button is visually prominent and easily clickable.

**Output Section:** Display the resulting postfix expression in a read-only text area or output field.

**Error Handling:** Provide clear and concise error messages to guide users in correcting input mistakes.

**Accessibility:** Ensure that the UI is intuitive and easy to navigate for users of all levels. Design the interface to be accessible, considering factors such as contrast, font size, and keyboard navigation.

**Feasible Element Used**

**Stack Data Structure:** A stack is typically used to store operators during the conversion process. This data structure follows the Last In, First Out (LIFO) principle, which aligns well with the nature of expression parsing.

**Operand Handling:** Operands (such as variables or constants) are outputted directly to the resulting postfix expression.

**Operator Precedence:** Operators have different precedence levels (e.g., multiplication/division have higher precedence than addition/subtraction). It's crucial to handle operators according to their precedence to ensure the correctness of the postfix expression.

**Element Positioning and Functionality**

**Scan the Infix Expression:** Iterate through each character/token of the infix expression from left to right.

**Operand Handling:** If an operand (variable or constant) is encountered, append it directly to the postfix expression.

**Parentheses Handling:** If a left parenthesis is encountered, push it onto the stack.

**Post-conversion Stack Cleanup:** After scanning the entire infix expression, pop any remaining operators from the stack and append them to the postfix expression.

**Result:** The postfix expression obtained after the conversion process is complete.

**Usage:** The resulting postfix expression can then be used for further processing, such as evaluation using a stack-based algorithm or code generation in the compiler.

|  |  |
| --- | --- |
| operator | precedence |
| id | 1 |
| () | 2 |
| unary minus | 3 |
| exponential | 4 |
| \* , / | 5 |
| + , - | 6 |
| $ | 7 |

**Algorithm**

1. Create an empty stack to hold operators.

2. Create an empty list to store the postfix expression.

3. Scan the infix expression from left to right:

a. If the current token is an operand (a number or variable), add it to the postfix expression list.

b. If the current token is a left parenthesis '(', push it onto the stack.

c. If the current token is a right parenthesis ')':

i. Pop operators from the stack and add them to the postfix expression list until a left parenthesis '(' is encountered.

ii. Pop and discard the left parenthesis '(' from the stack.

d. If the current token is an operator:

i. While the stack is not empty and the precedence of the current operator is less than or equal to the precedence of the operator at the top of the stack:

- Pop operators from the stack and add them to the postfix expression list.

ii. Push the current operator onto the stack.

4. After scanning the entire infix expression, pop any remaining operators from the stack and add them to the postfix expression list.

5. Return the postfix expression list.

**Conclusion**

The process of converting infix expressions to postfix, an essential component in compiler design, offers significant advantages in parsing and evaluating mathematical expressions efficiently. Through the shunting-yard algorithm or other similar methods, compilers can seamlessly transform infix notation into postfix, simplifying the parsing process and reducing complexity. This conversion facilitates easier evaluation of expressions, particularly in stack-based architectures, while also enhancing readability and maintainability of compiler code.

**C Program**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 100

int isOperator(char c) {

return (c == '+' || c == '-' || c == '\*' || c == '/');

}

int getPrecedence(char op) {

if (op == '+' || op == '-')

return 1;

else if (op == '\*' || op == '/')

return 2;

else

return 0;

}

void infixToPostfix(char \*infix, char \*postfix) {

char stack[MAX\_SIZE];

int top = -1;

int i, j;

for (i = 0, j = 0; infix[i] != '\0'; i++) {

if (infix[i] == '(') {

stack[++top] = '(';

} else if (infix[i] == ')') {

while (top != -1 && stack[top] != '(') {

postfix[j++] = stack[top--];

}

top--;

} else if (isOperator(infix[i])) {

while (top != -1 && getPrecedence(stack[top]) >= getPrecedence(infix[i])) {

postfix[j++] = stack[top--];

}

stack[++top] = infix[i];

} else {

postfix[j++] = infix[i];

}

}

while (top != -1) {

postfix[j++] = stack[top--];

}

postfix[j] = '\0';

}

int main() {

char infix[MAX\_SIZE];

char postfix[MAX\_SIZE];

printf("Enter the infix expression: ");

fgets(infix, MAX\_SIZE, stdin);

infix[strlen(infix) - 1] = '\0';

infixToPostfix(infix, postfix);

printf("Postfix expression: %s\n", postfix);

return 0;

}

**Input:**

Enter infix expression: a\*b-c+8

**Output:**

Postfix expression: ab\*c-8+

