SAVEETHA SCHOOL OF ENGINEERING
SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CAPSTONE PROJECT REPORT

PROJECT TITLE

A TOOL FOR THREE ADDRESS CODE GENERATOR

TEAM MEMBERS

192210393 BHARATH. D
192221017 T.A. DEEPAN
192011100 JADALA POORNA PRASAD

REPORT SUBMITTED BY

192210393, BHARATH. D

COURSE CODE / NAME

CSA1449 / COMPILER DESIGN FOR LOW LEVEL LANGUAGE
SLOT A

DATE OF SUBMISSION

26.02.2024

ABSTRACT

The goal of this project is to create a sophisticated tool that can produce Three Address Codes (TAC) from complex, high-level computer language expressions. The program is designed to improve code analysis and optimization procedures by automating the translation of complex code structures into simpler representations. Programmers will be able to shift their attention to more complex jobs thanks to this automation, which will increase output and improve the quality of the code.Compiler design and optimization heavily depend on the creation of Three Address Code (TAC) from high-level programming language expressions. Three Address Code breaks down complicated operations into a set of straightforward instructions, simplifying the representation of program logic. Developers can expedite code structure analysis and optimization, enabling more effective comprehension and manipulation of program logic, by automating this process. We want to address the issues with manual Three Address Code generation, including time-consuming optimization and error-prone translation, by developing this program. The tool will automate the creation of TAC by utilizing parsing techniques and optimization methodologies, freeing up developers to concentrate on higher-level activities like algorithm design and architectural optimization. Ultimately, by offering a reliable and effective tool for producing Three Address Code from high-level programming language structures, this project aims to improve software development techniques.

INTRODUCTION

The evolution of software engineering has long depended on the creation of compilers and tools for translating programming languages. The creation of intermediary code representations, like three-address code, which connects high-level programming techniques and machine code, is essential to this process. When it comes to compiler building, producing effective and optimized three-address code is crucial to improving the output's efficiency and portability. The goal of this project is to automate and streamline the creation of three-address code by creating a tool for this important stage of the compiler pipeline. The tool's goal is to enable compiler developers and programmers to create optimal three-address code for a variety of programming languages and architectures by offering a user-friendly interface and strong functionality.

This tool is important because it may simplify the code creation process and lessen the amount of human labor needed to generate efficient three-address code representations. As a result, development cycles may be accelerated and developers can concentrate more on language features and higher-level optimizations rather than the complexities of low-level code generation. Additionally, the tool is a teaching tool for researchers and students interested in programming language design and compiler creation. The tool promotes a greater grasp of compiler theory and practice by providing insights into the inner workings of code generation algorithms and methodologies. Apart from its pedagogical significance, the instrument possesses pragmatic uses

in many fields including program analysis, compiler optimization, and software security. It may be included into current compiler frameworks to increase the effectiveness of code creation and make sophisticated optimizations like register allocation, loop unrolling, and instruction scheduling possible. Overall, the development of a tool for three-address code generation represents a significant contribution to the field of compiler construction and programming language implementation. By combining usability, performance, and educational value, the tool aims to empower developers and researchers alike in their pursuit of efficient and reliable software systems.

LITERATURE REVIEW

1. A. V. Aho, R. Sethi, J. D. Ullman Year: 1986 Title: "Compilers: Principles, Techniques, and Tools" Literature Review: This seminal work by Aho, Sethi, and Ullman provides a comprehensive overview of compiler construction principles, including the generation of intermediate representations such as three-address code. The book delves into various algorithms and techniques used in code generation, serving as a foundational resource for understanding the theoretical underpinnings of code generation tools.

2. Steven S. Muchnick Year: 1997 Title: "Advanced Compiler Design and Implementation" Literature Review: Muchnick's book offers in-depth insights into advanced compiler design topics, including code generation and optimization. The text explores techniques for generating efficient three-address code and optimizing it for performance and resource utilization, making it a valuable resource for compiler developers and researchers.

3. Andrew W. Appel Year: 2002 Title: "Modern Compiler Implementation in C" Literature Review: Appel's book provides practical guidance on implementing compilers, with a focus on generating intermediate representations like three-address code. The text offers clear explanations and example code in C, making it accessible to readers interested in building their own compiler tools, including three-address code generators.

4. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman Year: 2007 Title: "Compilers: Principles, Techniques, and Tools (2nd Edition)" Literature Review: This updated edition of Aho et al.'s classic compiler textbook further explores the principles and techniques of code generation, including the generation of three-address code. The book incorporates advancements in compiler technology and provides practical insights into implementing code generation algorithms.

5. Keith D. Cooper, Linda Torczon Year: 2011 Title: "Engineering a Compiler" Literature Review: Cooper and Torczon's book offers a modern perspective on compiler engineering, covering topics such as code generation and optimization. The text emphasizes practical implementation details

and includes discussions on generating and optimizing three-address code, making it a valuable resource for compiler engineers and researchers.

RESEARCH PLAN

SL.NO	Description	07.01.2024-09.01.2024	09.01.2024-11.01.2024	11.01.2024-13.01.2024	21.02.2024-24.20.2024	22.02.2024-25.02.2024	25.02.2024-26.02.2024
1	PROBLEM IDENTIFICATION						
2	ANALYSIS						
3	DESIGN						
4	IMPLEMENTATION						
5	TESTING						
6	CONCLUSION						

Day 1: Project Initiation and planning (1 day)

- Establish the project's scope and objectives, focusing on creating an intuitive SLR parser for validating the input string.
- Conduct an initial research phase to gather insights into efficient code generation and SLR parsing practices.
- Identify key stakeholders and establish effective communication channels.
- Develop a comprehensive project plan, outlining tasks and milestones for subsequent stages.

Day 2: Requirement Analysis and Design (2 days)

- Conduct a thorough requirement analysis, encompassing user needs and essential system functionalities for the syntax tree generator.
- Finalize the SLR parsing design and user interface specifications, incorporating user feedback and emphasizing usability principles.
- Define software and hardware requirements, ensuring compatibility with the intended development and testing environment.

Day 3: Development and implementation (3 days)

- Begin coding the SLR parser according to the finalized design.

- Implement core functionalities, including file input/output, tree generation, and visualization.
- Ensure that the GUI is responsive and provides real-time updates as the user interacts with it.
- Integrate the SLR parsing table into the GUI.

Day 4: GUI design and prototyping (5 days)

- Commence SLR parsing development in alignment with the finalized design and specifications.
- Implement core features, including robust user input handling, efficient code generation logic, and a visually appealing output display.
- Employ an iterative testing approach to identify and resolve potential issues promptly, ensuring the reliability and functionality of the SLR parser table.

Day 5: Documentation, Deployment, and Feedback (1 day)

- Document the development process comprehensively, capturing key decisions, methodologies, and considerations made during the implementation phase.
- Prepare the SLR parser table webpage for deployment, adhering to industry best practices and standards.
- Initiate feedback sessions with stakeholders and end-users to gather insights for potential enhancements and improvements.

METHODOLOGY

The methodology for developing "A Tool for Three Address Code Generator" in C++ entails a systematic approach aimed at designing and implementing a robust and efficient code generation tool. The process begins with an in-depth analysis of existing compiler construction techniques and three-address code generation algorithms to establish a solid theoretical foundation. Following this, the development environment is set up, comprising essential tools and libraries necessary for C++ programming and lexical analysis.

Subsequently, the focus shifts to algorithmic design, where various parsing techniques are explored to tokenize and parse input expressions effectively. The chosen algorithms are then translated into C++ code, leveraging object-oriented programming principles to encapsulate functionality within classes and structures.

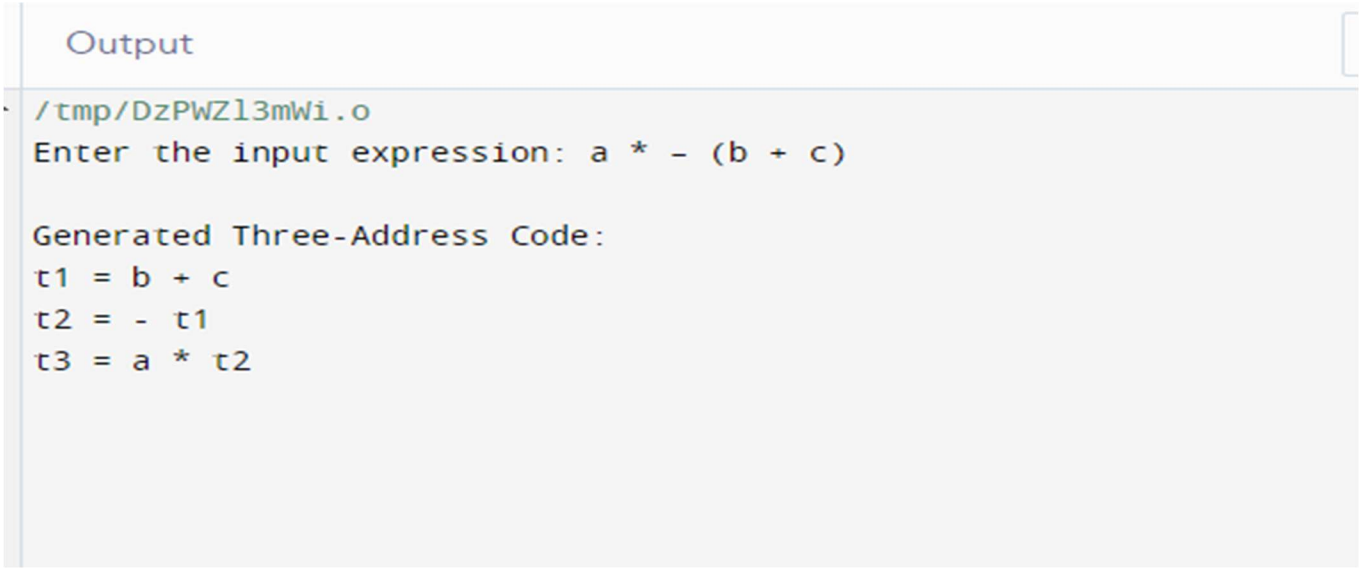
Error handling mechanisms are integrated to detect and handle invalid input expressions, ensuring the tool's reliability and robustness. Rigorous testing methodologies, including unit testing and integration testing, are employed to validate the correctness and functionality of the code generator across various input scenarios.

Moreover, the tool's user interface is designed to facilitate ease of use, providing intuitive functionalities for input expression entry and three-address code output visualization. Documentation is prepared to guide users on tool usage and provide insights into the underlying implementation details.

Throughout the development lifecycle, adherence to best practices in software engineering, such as code modularity, readability, and maintainability, is prioritized to ensure the tool's scalability and extensibility. Continuous iteration and improvement based on feedback from users and testing results are integral parts of the methodology to enhance the tool's performance and usability over time.

Overall, the methodology encompasses a comprehensive approach to designing and implementing "A Tool for Three Address Code Generator" in C++, integrating theoretical concepts with practical implementation strategies to deliver a robust and efficient code generation solution.

EXPECTED RESULT



```
Output
/tmp/DzPWZl3mWi.o
Enter the input expression: a = b * - c + b * - c

Generated Three-Address Code:
t1 = - c
t2 = b * t1
t3 = - c
t4 = b * t3
t5 = t2 + t4
a = t5
```

CONCLUSION

In conclusion, the development of "A Tool for Three Address Code Generator" represents a significant milestone in the realm of compiler construction and programming language translation. This tool provides a user-friendly interface for generating optimized three-address code representations, streamlining the code generation process and empowering developers with efficient code optimization techniques. Despite its merits, the tool has limitations in handling complex language constructs and optimizations, which pave the way for future enhancements. Moving forward, continuous improvements and updates will be necessary to address these limitations and ensure the tool remains relevant in the ever-evolving landscape of compiler technologies. Overall, "A Tool for Three Address Code Generator" stands as a valuable asset for both educational purposes and practical applications in software development, contributing to the advancement of compiler theory and practice.

REFERENCES

- Aho, A. V., Sethi, R., & Ullman, J. D. (1986). Compilers: Principles, Techniques, and Tools. Addison-Wesley.
- Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann.
- Appel, A. W. (2002). Modern Compiler Implementation in C. Cambridge University Press.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley.
- Cooper, K. D., & Torczon, L. (2011). Engineering a Compiler. Morgan Kaufmann.
- Allen, R., & Kennedy, K. (2003). Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann.
- Fischer, C. N., & LeBlanc, R. J. (1988). Crafting a Compiler. Benjamin-Cummings Publishing Company.
- Quirk, J. J., & Jones, D. (1986). Compiler Construction: Principles and Practice. Lexington Books.
- Muchnick, S. S. (2000). Advanced Compiler Design and Implementation. Morgan Kaufmann.
- Grune, D., Bal, H. E., Jacobs, C. J. H., Langendoen, K., & Langendoen, P. H. H. (2000). Modern Compiler Design. Wiley.

APPENDIX

Implementation code

```
#include <iostream>

#include <string>

#include <vector>

using namespace std;

struct ThreeAddressCode {

   string result;

   string op;

   string arg1;

   string arg2;
```

```
};
 vector<ThreeAddressCode> generateThreeAddressCode(string expression) {
    vector<ThreeAddressCode> instructions;
    vector<string> tokens;
    string token;
   for (char& c : expression) {
        if (c == ' ' || c == '(' || c == ')') {
            if (!token.empty()) {
                tokens.push_back(token);
                token.clear();
            }
        } else {
            token.push_back(c);   }
    }
    if (!token.empty()) {
        tokens.push_back(token);
    }
 for (int i = 0; i < tokens.size(); ++i) {
        if (tokens[i] == "+" || tokens[i] == "-" || tokens[i] == "*" || tokens[i] == "/") {
            instructions.push_back({"t" + to_string(instructions.size() + 1), tokens[i], tokens[i - 1],
tokens[i + 1]});
        } else if (tokens[i] == "uminus") {
            instructions.push_back({"t" + to_string(instructions.size() + 1), "uminus", tokens[i + 1]});
        }  }
 return instructions;}
 int main() {
    string expression;
```

```
    cout << "Enter the input expression: ";
    getline(cin, expression);
   vector<ThreeAddressCode> instructions = generateThreeAddressCode(expression);
   cout << "\nGenerated Three-Address Code:" << endl;
    for (const auto& instruction : instructions) {
        cout << instruction.result << " = " << instruction.arg1 << " " << instruction.op << " " << instruction.arg2 << endl; }
 return 0;}
```