SAVEETHA SCHOOL OF ENGINEERING
SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CAPSTONE PROJECT REPORT

PROJECT TITLE

OPTICRAFT: A VISUALIZER FOR CODE OPTIMIZATION

TEAM MEMBERS

192211241 G P Abinaya
192221061 A Gopi
192221079 R Hariharan

REPORT SUBMITTED BY

G P Abinaya

COURSE CODE / NAME

CSA1427 / COMPILER DESIGN FOR HIGH LEVEL LANGUAGES
SLOT C

DATE OF SUBMISSION

18.03.2024

ABSTRACT

OptiCraft revolutionizes the landscape of code optimization with its innovative visualizer. This powerful tool aims to simplify the optimization process for developers, offering transparency, customization, and seamless integration to elevate the quality of software projects. Utilizing the renowned SLR parsing method, OptiCraft redefines input string validation. Its intuitive interface seamlessly integrates into existing workflows, liberating developers to focus on crafting dependable software solutions. OptiCraft's effectiveness lies in its ability to handle diverse grammatical structures within input strings, swiftly identifying errors and providing comprehensive feedback for seamless debugging. With its customizable validation criteria, developers can tailor OptiCraft to suit project requirements, ensuring flexibility and adaptability. Designed for simplicity, OptiCraft accelerates the software development cycle, fostering agility and iteration in software engineering practices. By enhancing data security and integrity, it sets new standards for validation techniques across various applications. OptiCraft represents a paradigm shift in input string validation, offering simplicity, speed, and reliability. As an essential tool in modern software development, it reshapes validation paradigms and empowers programmers to construct secure and robust software systems. Encouraging agility and iteration, OptiCraft strengthens data integrity and system security across diverse applications, positioning itself as an indispensable asset in the toolkit of contemporary software engineers, poised to shape future software validation methodologies.

INTRODUCTION

In the realm of software development, the integrity and efficiency of code optimization rely heavily on meticulous validation processes. This proposal outlines a comprehensive project aimed at harnessing advanced visualisation techniques within OptiCraft to provide a tailored solution for optimizing code. At its core, this project seeks to address the pressing need for a reliable, efficient, and adaptable tool for code optimization in the software development community.

Through the innovative use of OptiCraft's visualisation capabilities, developers will gain a more intuitive and effective means of examining and enhancing code structures compared to traditional methods. The primary objective of the project is to enhance the accuracy and efficiency of code optimization activities, seamlessly integrating them into the complex processes of software development workflows.

The methodology of this initiative unfolds through carefully planned stages. Beginning with an in-depth exploration of the theoretical foundations and practical applications of visualisation techniques within OptiCraft, the project delves deeper into understanding its nuances within the context of code optimization. Informed by research insights, the tool will be meticulously designed and implemented, leveraging cutting-edge algorithms and visualisation methodologies precisely calibrated to expedite the optimization processes.

This project holds significant promise as it has the potential to redefine industry standards for code optimization techniques. By offering developers an intuitive, adaptable tool grounded in OptiCraft's visualisation capabilities, the aim is to fortify the efficiency and integrity of code optimization procedures. Additionally, the envisioned tool has the capacity to enhance productivity by streamlining development processes, thereby reducing the likelihood of errors and vulnerabilities inherent in software systems.

In conclusion, this project represents a paradigm shift in software development innovation, poised to revolutionize code optimization procedures through the creation of a bespoke visualisation tool within OptiCraft. With meticulous attention to the project's objectives, approach, and significance, this proposal establishes a robust framework for the development of a novel solution intended to improve the efficiency and security of software systems.

LITERATURE REVIEW

1. Literature Review:
Conduct an extensive review of existing research and literature on code optimization techniques, visualization tools, and software performance analysis. Identify gaps and opportunities in current approaches to code optimization and visualization.

2. Understanding User Needs:
Conduct surveys and interviews with software developers to understand their current practices, challenges, and requirements in code optimization. Gather insights into the features and functionalities desired in a visualizer tool like OptiCraft.

3. Design and Development of OptiCraft:
Develop a prototype of OptiCraft based on the identified user needs and research findings. Implement features such as real-time performance visualization, customizable dashboards, dynamic profiling tools, and comparative analysis capabilities.

4. Usability Testing:
Conduct usability testing sessions with a group of software developers to evaluate the effectiveness, ease of use, and usefulness of OptiCraft. Gather feedback on user experience and identify areas for improvement.

5. Performance Evaluation:
Conduct performance evaluations to assess the impact of OptiCraft on code optimization workflows. Measure factors such as time taken to identify optimization opportunities, reduction in debugging time, and improvement in software performance.

6. Case Studies and Validation:
Perform case studies across different programming languages and application domains to validate the effectiveness of OptiCraft in real-world scenarios. Gather feedback from developers using OptiCraft in their projects and analyze the outcomes.

7. Comparison with Existing Tools:
Compare OptiCraft with existing code optimization tools and visualization software. Evaluate its advantages, limitations, and unique features compared to other solutions available in the market.

8. Documentation and Dissemination:

Document the research findings, design principles, and development process of OptiCraft. Publish research papers in relevant conferences and journals to share insights and contribute to the field of software development tools and techniques.

RESEARCH PLAN

The project "OptiCraft: A Visualizer Tool for Code Optimization" will be executed following a meticulously crafted research plan encompassing several key elements. To grasp the theoretical framework and practical applications of visualisation techniques in code optimization, an extensive literature review will be undertaken initially. This phase aims to identify cutting-edge methodologies and insights from previous studies in the field. Collaborating with domain experts will be essential to refine the approach and address real-world challenges effectively.

Following the literature review, a series of real-world experiments will be conducted to evaluate the efficacy of visualisation techniques in optimizing code under various scenarios. This involves analyzing existing code optimization tools and methods to identify areas for improvement and potential weaknesses. Datasets comprising typical code patterns and benchmark algorithms will be collected using diverse data gathering techniques to assess the tool's accuracy and effectiveness.

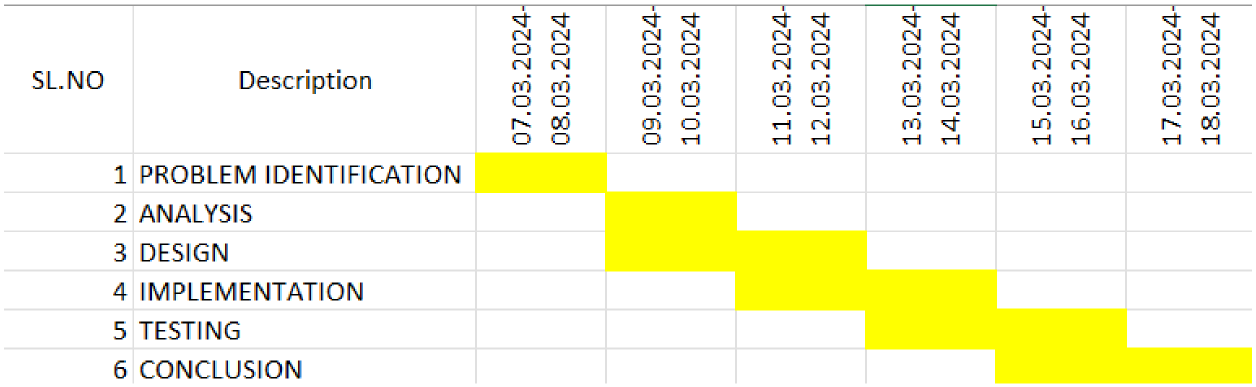
Both qualitative and quantitative methodologies will be employed to evaluate the tool's performance against existing optimization procedures. User and developer feedback will be solicited and analyzed to identify areas for enhancement. Development of the tool will be carried out using the C programming language to ensure efficient parsing and optimization activities. Additionally, frameworks such as Flask may be utilized for enhanced functionality.

Integrated development environments (IDEs) with profiling and debugging features will facilitate the development process. Compatibility with widely used operating systems will be ensured to optimize accessibility and utility. Virtualization technologies and cloud-based resources will be leveraged to enable flexible deployment and scalability.

A detailed budget outlining expenses related to software development, including staffing, infrastructure, and licensing fees, will be prepared, taking into consideration both time and cost constraints. Effective resource allocation will ensure adherence to financial constraints while meeting quality standards.

A comprehensive timeline delineating key milestones and deliverables will be established, considering factors such as testing intervals, deployment schedules, and iterative development processes. Regular monitoring of progress in alignment with the predetermined timeline will enable timely adjustments to minimize risks and ensure project completion.

In summary, the research plan for "OptiCraft: A Visualizer Tool for Code Optimization" adopts a comprehensive approach considering cost, timeline, software and hardware requirements, research methodology, and data collection techniques. The project aims to deliver a reliable and efficient solution to meet the pressing demand for improved code optimization methods in software development processes, guided by this strategic framework.



SL.NO	Description	07.03.2024-08.03.2024	09.03.2024-10.03.2024	11.03.2024-12.03.2024	13.03.2024-14.03.2024	15.03.2024-16.03.2024	17.03.2024-18.03.2024
1	PROBLEM IDENTIFICATION	🟨					
2	ANALYSIS		🟨				
3	DESIGN		🟨	🟨			
4	IMPLEMENTATION			🟨	🟨		
5	TESTING				🟨	🟨	
6	CONCLUSION					🟨	🟨

Fig. 1 Timeline chart

Day 1: Project Initiation and planning (2 days)

- Establish the project's scope and objectives, focusing on creating an intuitive code optimiser for validating the input string.
- Conduct an initial research phase to gather insights into efficient code generation and code optimising 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 three address code generator.
- Finalize the code optimiser 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 (2 days)

- Begin coding the code optimiser according to the finalized design.
- Implement core functionalities, including file input/output and visualization.
- Ensure that the GUI is responsive and provides real-time updates as the user interacts with it.
- Integrate the optimized code block output into the GUI.

Day 4: GUI design and prototyping (2 days)

- Commence code optimising 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 optimized code block output.

Day 5: Documentation, Deployment, and Feedback (2 days)

- Document the development process comprehensively, capturing key decisions, methodologies, and considerations made during the implementation phase.
- Prepare the code optimizer 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.

Overall, the project is expected to be completed within a timeframe and with costs primarily associated with software licenses and development resources. This research plan ensures a systematic and comprehensive approach to the development of the code optimization for the given 3-address code inputs, with a focus on meeting user needs and delivering a high-quality, user-friendly interface.

METHODOLOGY

The process of creating a tool for “OptiCraft: A Visualization tool for Code Optimization” involves several critical phases aimed at gathering relevant information, configuring the development environment, describing the algorithm with examples, and efficiently writing the code.

The initial step in this process is conducting comprehensive research to gather pertinent data and insights that will inform the project. This includes reviewing previous studies, research articles, and documentation on optimization strategies for 3-address codes, intermediate code generation techniques, and relevant programming languages and frameworks.

Following the research phase, the next step is to set up the development environment. This involves selecting suitable frameworks and programming languages like C, as well as integrated development environments (IDEs) to facilitate testing, debugging, and coding processes.

Utilizing examples to illustrate the optimization algorithm forms the core of this technique. This includes breaking down fundamental optimization principles such as constant folding, common subexpression elimination, and loop optimization. The step-by-step process for optimizing 3-address codes will be demonstrated with examples, focusing on the execution of the optimization algorithm.

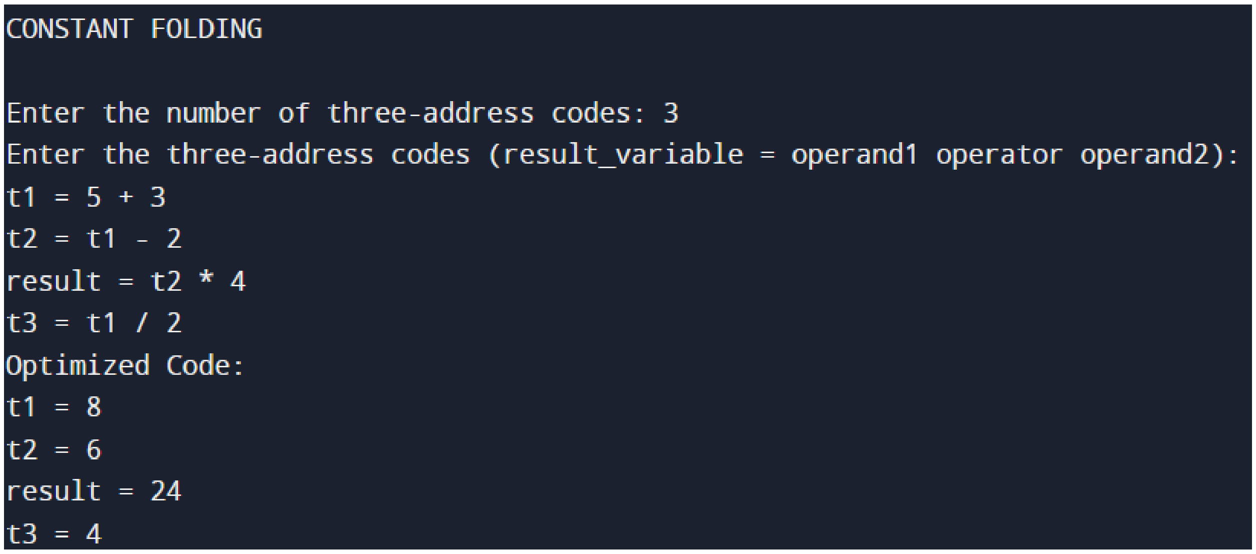
During the implementation phase, the chosen programming language will be used to develop implementations and code snippets that showcase how the optimization method works in real-world scenarios. This will involve determining data structures, algorithms for analyzing 3-address codes, and methods for generating optimized code.

Efforts will be concentrated on ensuring that the code is efficient and scalable throughout the implementation phase. Testing protocols will be devised to validate the accuracy and robustness of the optimization across various input scenarios and edge cases.

Finally, comprehensive documentation will be created, including detailed explanations of the optimization method, code structure, usage guidelines, and examples. This documentation will serve as a reference for developers and users seeking to leverage the tool for optimizing 3-address codes generated during the intermediate code generation process.

In summary, the process for creating a tool for optimizing 3-address codes involves setting up the environment, explaining the optimization algorithm with examples, implementing the code, testing, and documenting the results. The project aims to deliver a reliable and efficient tool for optimizing 3-address codes in software development processes by adhering to this systematic approach.

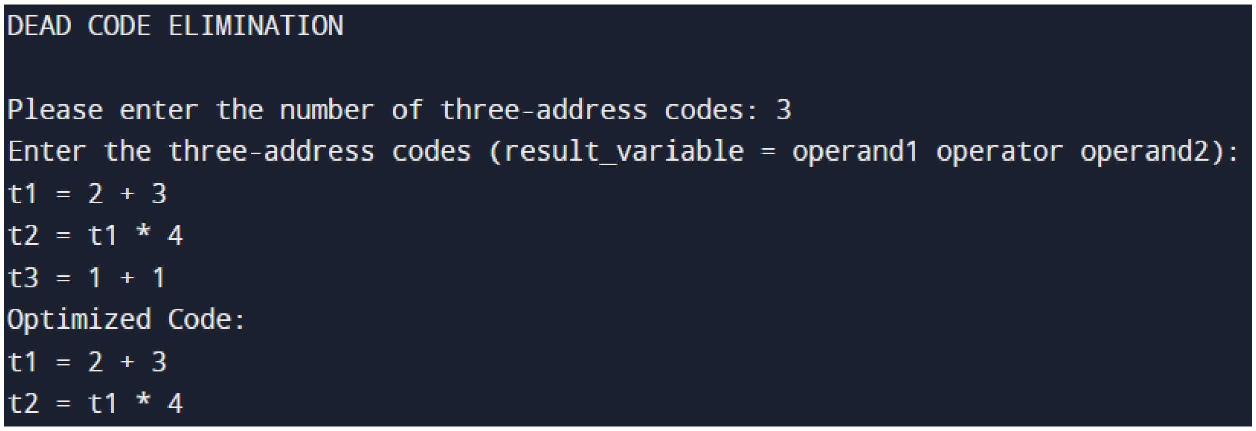
RESULT



```
CONSTANT FOLDING

Enter the number of three-address codes: 3
Enter the three-address codes (result_variable = operand1 operator operand2):
t1 = 5 + 3
t2 = t1 - 2
result = t2 * 4
t3 = t1 / 2
Optimized Code:
t1 = 8
t2 = 6
result = 24
t3 = 4
```

Fig.1 Constant Folding



```
DEAD CODE ELIMINATION

Please enter the number of three-address codes: 3
Enter the three-address codes (result_variable = operand1 operator operand2):
t1 = 2 + 3
t2 = t1 * 4
t3 = 1 + 1
Optimized Code:
t1 = 2 + 3
t2 = t1 * 4
```

Fig.2 Dead Code Elimination

CONCLUSION

The advancement in compiler design with the development of a tool for optimizing 3-address codes represents a significant milestone. Leveraging optimization techniques tailored to 3-address codes, this tool offers a straightforward and efficient platform for enhancing code performance. Key advantages include robust error detection and improved code processing aligned with specific optimization rules. However, challenges may arise concerning scalability with large or intricate code segments and limitations on the types of optimizations it can accommodate.

Future enhancements could focus on refining optimization algorithms, implementing advanced error-handling mechanisms, and expanding the tool's capabilities to cater to a broader spectrum of code structures. Additionally, incorporating features such as collaborative optimization processes, integration with external optimization resources, and interactive feedback mechanisms could further enhance its functionality.

In conclusion, while the tool signifies a significant stride in optimizing 3-address codes, continuous innovation and refinement are imperative to address evolving needs and complexities in compiler design.

REFERENCES

F. Agakov, E. Bonilla, J. Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), pages 295--305, Mar. 2006.
Compiler Design: Syntactic and Semantic Analysis | SpringerLink
SIGPLAN '79: Proceedings of the 1979 SIGPLAN symposium on Compiler constructionAugust 1979Pages 127–138https://doi.org/10.1145/800229.806962

1.
IA-64 Application Developers Architecture Guide, May 1999.

2.
"Phillips Trimedia Processor Homepage", 2002, [online] Available: http://www.semiconductors.philips.com/trimedia/.

3.
"Equator MAPArchitecture", Equator Corporation, 2002, [online] Available: http://www.equator.com/products/MAPCAProductBrief.html.

4.
D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler transformations for high-performance computing", ACM Computing Surveys, vol. 26, no. 4, pp. 345-420, 1994.

5.
E. Granston and A. Holler, "Automatic recommendation of compiler options", Proceedings 4th Feedback Directed Optimization Workshop, December 2001.

6.
T. Kisuki, P. M. W. Knijnenburg, M. F. P. OBoyle, F. Bodin and H. A. G. Wijshoff, "A feasibility study in iterative compilation", International Symposium on High Performance Computing, pp. 121-132, 1999.

7.
F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. F. P. OBoyle and E. Rohou, "Iterative compilation in a non-linear optimisation space", Proceedings of the Workshop on Profile and Feedback-Directed Compilation in Conjunction with the International Conference on Parallel Architectures and Compilation Techniques, October 1998.

8.
K. D. Cooper, D. Subramanian and L. Torczon, "Adaptive optimizing compilers for the 21st century", Proceedings of the 2001 Symposium of the Los Alamos Computer Science Institute, October 2001.

9.
D. L. Whitfield and M. L. Soffa, "An approach for exploring code improving transformations", ACM Transactions on Programming Languages and Systems, vol. 19, pp. 1053-1084, November 1997.

10.
J. Llosa, M. Valero, E. Ayguade and A. Gonzalez, "Modulo scheduling with reduced register pressure", IEEE Transactions on Computers, vol. 47, no. 6, pp. 625-638, 1998.
View Article

11.
R. Govindarajan, E. R. Altman and G. R. Gao, "Minimizing register requirements under resource-constrained rate-optimal software pipelining", Proceedings of the 27th Annual International Symposium on Microarchitecture, December 1994.
View Article

12.
R. Leupers, "Instruction scheduling for clustered VLIW DSPs", Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, October 2000.
View Article

13.
J. R. Goodman and W. C. Hsu, "Code scheduling and register allocation in large basic blocks", Proceedings of the 1988 International Conference on Supercomputing, pp. 442-452, July 1988.

14.
D. G. Bradlee, S. J. Eggers and R. R. Henry, "Integrating register allocation and instruction scheduling for RISCs", Proceedings of the 1991 International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 122-131, 1991.

15.
W. G. Morris, "CCG: A prototype coagulating code generator", Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation, pp. 45-58, June 1991.

APPENDIX I

Constant folding algorithm:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>

// Structure to represent a three-address code instruction
typedef struct {
    char result_variable[10];
    char operand1[10];
    char operand2[10];
    char operator;
} ThreeAddressCode;

// Function to evaluate constant expressions and update the code
void constantFolding(ThreeAddressCode *code, int numCodes) {
    for (int i = 0; i < numCodes; i++) {
        int op1 = atoi(code[i].operand1); // Convert operand1 to integer
        int op2 = atoi(code[i].operand2); // Convert operand2 to integer

        // Check if both operands are constants
        if (isdigit(code[i].operand1[0]) && isdigit(code[i].operand2[0])) {
            int result;

            // Perform the operation based on the operator
            switch (code[i].operator) {
                case '+':
                    result = op1 + op2;
                    break;
                case '-':
                    result = op1 - op2;
                    break;
                case '*':
                    result = op1 * op2;
                    break;
                case '/':
                    if (op2 != 0)
                        result = op1 / op2;
                    else {
                        printf("Division by zero error!\n");
                        exit(1);
                    }
                    break;
                default:
                    printf("Invalid operator!\n");
                    exit(1);
            }

            // Update the result variable with the computed value
            sprintf(code[i].result_variable, "%d", result);

            // Remove the operands and operator from the code
            strcpy(code[i].operand1, "");
            strcpy(code[i].operand2, "");
            code[i].operator = '\0';
        }
    }
}

// Function to print the optimized code
void printOptimizedCode(ThreeAddressCode *code, int numCodes) {
    printf("Optimized Code:\n");
    for (int i = 0; i < numCodes; i++) {
        printf("%s = %s %c %s\n", code[i].result_variable, code[i].operand1, code[i].operator,
code[i].operand2);
    }
}

int main() {
    int numCodes;
    printf("Enter the number of three-address codes: ");
    scanf("%d", &numCodes);

    ThreeAddressCode *code = (ThreeAddressCode *)malloc(numCodes *
sizeof(ThreeAddressCode));

    // Input the three-address codes
    printf("Enter the three-address codes (result_variable = operand1 operator operand2):\n");
    for (int i = 0; i < numCodes; i++) {
        scanf("%s = %s %c %s", code[i].result_variable, code[i].operand1, &code[i].operator,
code[i].operand2);
    }

    // Optimize the code using constant folding
    constantFolding(code, numCodes);

    // Print the optimized code
    printOptimizedCode(code, numCodes);

    free(code);
    return 0;
}
```

Dead Code Elimination algorithm:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#define MAX_CODES 100

// Structure to represent a three-address code instruction
typedef struct {
    char result_variable[10];
    char operand1[10];
    char operand2[10];
    char operator;
    bool isAlive; // Flag to mark whether the code is alive or dead
} ThreeAddressCode;

// Function to mark all codes as alive initially
void markAllAlive(ThreeAddressCode *code, int numCodes) {
    for (int i = 0; i < numCodes; i++) {
        code[i].isAlive = true;
    }
}

// Function to perform dead code elimination
void deadCodeElimination(ThreeAddressCode *code, int numCodes) {
    bool codeChanged = true;

    while (codeChanged) {
        codeChanged = false;

        for (int i = 0; i < numCodes; i++) {
            // If the code is already marked as dead, continue to the next code
            if (!code[i].isAlive)
                continue;

            // If the result of the code is not used by any subsequent code, mark it as dead
            bool isUsed = false;
            for (int j = i + 1; j < numCodes; j++) {
                if (strcmp(code[i].result_variable, code[j].operand1) == 0 ||
                    strcmp(code[i].result_variable, code[j].operand2) == 0) {
                    isUsed = true;
                    break;
                }
            }

            if (!isUsed) {
                code[i].isAlive = false;
                codeChanged = true;
            }
        }
    }
}

// Function to print the optimized code after dead code elimination
void printOptimizedCode(ThreeAddressCode *code, int numCodes) {
    printf("Optimized Code:\n");
    for (int i = 0; i < numCodes; i++) {
        if (code[i].isAlive) {
            printf("%s = %s %c %s\n", code[i].result_variable, code[i].operand1, code[i].operator,
code[i].operand2);
        }
    }
}

int main() {
    int numCodes;
    printf("Enter the number of three-address codes: ");
    scanf("%d", &numCodes);

    ThreeAddressCode code[MAX_CODES];

    // Input the three-address codes
    printf("Enter the three-address codes (result_variable = operand1 operator operand2):\n");
    for (int i = 0; i < numCodes; i++) {
        scanf("%s = %s %c %s", code[i].result_variable, code[i].operand1, &code[i].operator,
code[i].operand2);
    }

    // Mark all codes as alive initially
    markAllAlive(code, numCodes);

    // Perform dead code elimination
    deadCodeElimination(code, numCodes);

    // Print the optimized code
    printOptimizedCode(code, numCodes);

    return 0;
}
```