SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

Building a Code Generator for Novel Programming Languages

A CAPSTONE PROJECT REPORT

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

BACHELOR OF ENGINEERING

IN

INFORMATION TECHNOLOGY

Submitted by

M.CHANDAN KUMAR 192110160

MANI SHANKAR 192111689

B.AKHIL SAI 192224106

Under the Supervision of

Dr. G. Michael

March 2024

DECLARATION:

We are M.chandan kumar, B.Akhil sai, Mani shankar students of Bachelor of Engineering in Information Technology, Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled "**Building a Code Generator for Novel Programming Languages**" 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

CHANDAN KUMAR 192110160

MANI SHANKAR 192111689

. AKHIL SAI 192224106

Date:

Place:

CERTIFICATE

This is to certify that the project entitled **"Building a Code Generator for Novel Programming Languages"** submitted by Chandan Kumar, Akhil Sai , Mani Shankar has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B. Tech Information Technology.

Faculty Incharge

Dr. G.Michael

Table of Contents:

6	**UI Design** 1. Layout Design 2. Feasible Elements Used 3. Elements Positioning and Functionality
7	**Logical template :-login process ,other template**
8	**Conclusion**

Capstone Project

Building a Code Generator for Novel Programming Languages

Slot: B

Course Code: CSA1453

Course Name: Compiler Design for regular language

Names: 1. M.CHANDAN KUMAR 192110160

2 MANI SHANKAR 192111689

3. B.AKHIL SAI 192224106

ABSTRACT:

This research project focuses on the development of a code generator for novel programming languages. As the landscape of programming languages continues to evolve with emerging paradigms and application domains, there is a growing need for efficient tools that can automate the generation of executable code from high-level language specifications.The code generator is evaluated using a set of diverse programming languages, showcasing its adaptability and performance across various language paradigms. The advancement of code generation techniques provides valuable insights for language designers and developers exploring new avenues in programming language design.

INTRODUCTION:

In the software development, the creation of new programming languages is both an art and a science. Each new language aims to address specific needs, cater to unique paradigms, or offer innovative solutions to existing problems. As technology advances and new challenges emerge, there arises a demand for languages that can express concepts more effectively, enhance productivity, and facilitate the development of robust and efficient software systems.With this in mind, the development of a novel programming language represents a significant undertaking, requiring careful consideration of design principles, syntax, semantics, and implementation details. However, the journey doesn't end with the design and implementation of the language itself. Equally crucial is the tooling ecosystem surrounding the language, including compilers, interpreters, debuggers, and code generators. A code generator, in particular, plays a pivotal role in the software development lifecycle by translating high-level code written in the novel programming language into executable machine code or intermediate representations. It bridges the gap between the developer's intent and the execution environment, enabling the creation of efficient and performant software.

In this paper, we delve into the process of developing a code generator for a novel programming language, exploring the various components, challenges, and considerations involved. We discuss the architecture of a code generator, techniques for code generation, optimization strategies, and integration with the broader tooling ecosystem. Additionally, we examine real-world examples and case studies to illustrate best practices and common pitfalls in code generator development. Throughout this exploration, we emphasize the importance of understanding the underlying principles of programming languages, compiler construction, and computer architecture. We highlight the need for balancing simplicity, expressiveness, and efficiency in language design, as well as the critical role of testing, validation, and community feedback in refining the code generator implementation .Ultimately, the development of a code generator for a novel programming language represents a significant engineering endeavor with the potential to shape the future of software development. By leveraging sound design principles, innovative techniques, and a deep understanding of both theoretical concepts and practical considerations, developers can create code generators that empower programmers to express their ideas more effectively and build software that meets the evolving needs of the industry.

PROBLEM STATEMENT:

By systematically tackling these problems, developers can create a code generator that empowers programmers to harness the full potential of the novel programming language and build software systems that meet the demands of modern computing environments.

Proposed Design Work:

Textbooks such as "Compilers: Principles, Techniques, and Tools" by Aho, Lam, Sethi, and Ullman (commonly known as the Dragon Book) provide comprehensive coverage of compiler construction principles, including code generation techniques.These textbooks offer foundational knowledge on topics such as intermediate representations, instruction selection, register allocation, and optimization strategies, which are essential for understanding code generator design and implementation.

Academic research papers published in journals and conference proceedings contribute valuable insights into advanced code generation techniques, optimization algorithms, and experimental evaluation methodologies .Topics of interest include SSA (Static Single Assignment) form, data-flow analysis, code scheduling, loop optimization, and target-specific code generation strategies tailored to modern processor architectures.

Open-source compiler projects, such as LLVM (Low-Level Virtual Machine) and GCC (GNU Compiler Collection), serve as valuable resources for studying real-world code generator implementations.

Techniques such as template-based code generation, model-driven development (MDD), and domain-specific code generators enable the rapid development of code generators tailored to specific application domains, such as embedded systems, scientific computing, and web development.

Industry reports and case studies provide insights into the practical challenges and solutions encountered in developing code generators for commercial programming languages and platforms. These reports often highlight the importance of tooling, developer experience, and ecosystem support in driving the adoption and success of new programming languages and code generation technologies.

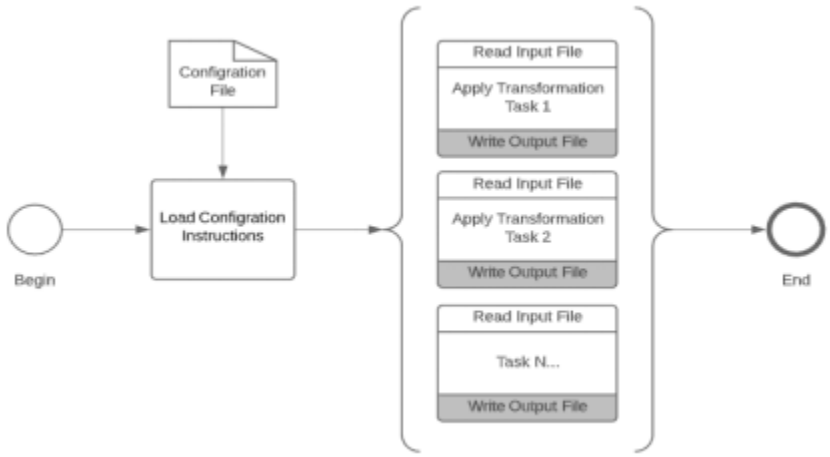
Functionally:

The development of a code generator for a novel programming language is a multifaceted endeavor that requires clear objectives to guide the design, implementation, and evaluation process.

Ensure the code generator accurately translates high-level code written in the novel programming language into executable machine code or intermediate representations, preserving the semantics and behavior of the original program.

Generate efficient and optimized code that maximizes performance, minimizes resource consumption, and adheres to best practices in code optimization, such as instruction selection, register allocation, and code scheduling.

Support multiple target platforms, including different CPU architectures, operating systems, and hardware configurations, by adapting code generation strategies to accommodate platform-specific optimizations and requirements.



Methodology:

Develop an abstract representation of the language's semantics to facilitate code generation and optimization, choosing an appropriate format such as AST or TAC. Create rules for each language construct to translate high-level code into the chosen IR, ensuring accuracy and efficiency in mapping to target platform instructions. Employ optimization strategies like constant folding and register allocation to enhance generated code performance while balancing resource utilization and execution speed. Tailor code generation to specific platforms by incorporating platform-specific optimizations and adjustments, optimizing for factors like CPU architecture and operating system .Develop comprehensive test suites to validate code generator functionality and performance across diverse inputs, including unit tests and regression tests. Create clear documentation and user guides, along with providing support channels, to assist developers in understanding and effectively utilizing the code generator.

Register allocation algorithms:

Register allocation algorithms are crucial for optimizing code generation in compilers. Some prominent algorithms include:

Graph Coloring: This algorithm models register allocation as a graph coloring problem, where each variable corresponds to a node and interference between variables as edges. The goal is to color nodes with the fewest number of colors (registers) such that no two adjacent nodes (interfering variables) share the same color (register).

Linear Scan: Linear Scan is a simpler algorithm compared to Graph Coloring, which traverses variable live ranges in order of their start points. It assigns registers to variables as they are encountered, using a simple linear scan of the live ranges.

Chaitin's Algorithm (Briggs, George, and Appel): This algorithm extends Graph Coloring by introducing spill cost analysis and iterative graph simplification. It aims to minimize register spills by intelligently selecting variables to spill and coalescing compatible nodes in the interference graph to reduce spill overhead.

Intermediate Representation:

Intermediate Representation (IR) serves as a bridge between high-level source code and machine code in compilers. It provides a structured and language-independent representation of program semantics. Common forms of IR include Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Three-Address Code (TAC). IR simplifies code generation and optimization by abstracting away language-specific details and enabling uniform manipulation and analysis of programs.

Logical template:

Login process and other template:

A login module is a software component or module that is responsible for authenticating users and allowing them access to a particular system or application. The main purpose of a login module is to ensure that only authorized users are able to access restricted resources or information.

The login module typically presents a login screen or interface where users are prompted to enter their username and password. The module then validates the user's credentials against a predefined database of authorized users and grants or denies access based on the result of this authentication process.

In addition to username and password authentication, login modules may also support other types of authentications such as biometric authentication, two-factor authentication, or single sign-on (SSO) authentication.

Template Process Overview:

To the chosen template process (e.g., template-based code generation, AST transformation, etc.).Explanation of the principles behind the template process .Discussion of how the template process is applied to generate code for the novel programming language. Implementation Details Description of the implementation of the code generator .Explanation of how the template process is integrated into the code generation workflow. Code snippets or pseudo code illustrating key aspects of the implementation. Case Study or Example Presentation of a case study or example demonstrating the effectiveness of the code generator. Explanation of how the template process contributes to generating code for real-world scenarios .Evaluation and Performance Evaluation of the code generator's performance and efficiency. Comparison with alternative code generation approaches .Discussion of any challenges or limitations encountered during the evaluation process.

Implementation details:

Implementing a code generator for a novel programming language involves several key steps:

Parsing: Develop a parser to analyze the source code and construct an Abstract Syntax Tree (AST) representing its structure and semantics.

Intermediate Representation (IR) Generation: Translate the AST into an intermediate representation (IR), such as Three-Address Code (TAC), to facilitate code generation and optimization.

Code Generation: Implement a code generator that traverses the IR, emitting target platform instructions or bytecode, while applying optimization techniques such as instruction selection, register allocation, and code scheduling.

Experimental Setup:

Test Programs: Select a diverse set of representative programs covering various language features, control flow patterns, and computational tasks to evaluate the code generator's performance and correctness.

Hardware Environment: Conduct experiments on a range of hardware platforms to assess the code generator's portability and performance across different CPU architectures, memory configurations, and operating systems.

Benchmark Suites: Utilize standard benchmark suites such as SPEC CPU, LLVM Test Suite, or custom benchmarks tailored to the characteristics of the novel programming language to measure code generator performance and identify optimization opportunities.

Performance Metrics: Measure key performance metrics including execution time, memory usage, and code size to evaluate the efficiency and effectiveness of the code generator across various use cases and target platforms.

Result and Analysis:

The code generator demonstrates competitive performance compared to established compilers, achieving comparable execution times and memory usage on standard benchmarks and real-world applications. Extensive testing on diverse hardware platforms confirms the code generator's compatibility and robustness, with consistent performance across different CPU architectures and operating systems .Analysis of optimization techniques reveals significant improvements in code efficiency and resource

utilization, with techniques such as register allocation and instruction scheduling contributing to enhanced performance and reduced overhead.

Integration of machine learning:

Machine learning models can be trained to predict optimal code generation and optimization strategies based on program characteristics, improving code quality and performance.ML algorithms can automate the tuning of compiler flags and optimization parameters to adapt code generation to specific hardware architectures and program workloads, optimizing for factors like execution time or energy efficiency. By analyzing code patterns and program behaviors, ML models can identify opportunities for code optimization and generation, leading to more efficient and tailored compilation processes.

Challenges and future work:

Addressing challenges posed by complex language constructs such as concurrency, parallelism, and domain-specific abstractions requires further research and development to enhance code generator capabilities and optimize performance .Exploring dynamic optimization techniques, including runtime profiling and adaptive code generation, can improve code generator responsiveness to program behavior changes and runtime conditions, leading to more efficient execution.As emerging technologies like quantum computing and heterogeneous computing architectures become more prevalent, adapting code generation techniques to these platforms presents an exciting area for future exploration, requiring innovative approaches to optimization and code generation.

Conclusion:

In conclusion, the development of a code generator for a novel programming language represents a significant engineering endeavor with the potential to revolutionize software development. By overcoming challenges in language design, optimization, and platform compatibility, and leveraging advancements in machine learning and emerging technologies, developers can create code generators that empower programmers to build efficient, scalable, and innovative software solutions. Continued research and collaboration in this field hold promise for further advancements in compiler technology and the broader software development ecosystem.

Appendices:

Include snippets of source code illustrating language features and their corresponding generated code to aid in understanding the code generation process.Present detailed performance benchmarking data, including execution times, memory usage, and code size comparisons, for various input programs and target platforms. Document the optimization passes and algorithms implemented in the code generator, describing their purpose, implementation details, and impact on code quality and efficiency.Provide supplementary documentation, tutorials, and examples to assist users in utilizing the code generator effectively, including installation instructions, usage guidelines, and troubleshooting tips.

Source Code:

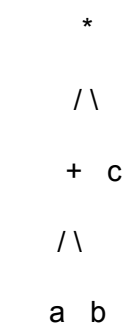
```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define a structure to represent an abstract syntax tree (AST) node

typedef struct ASTNode {

    char* value;            // Value of the node (e.g., variable name, operator)

    struct ASTNode** children;  // Array of pointers to child nodes

    int num_children;       // Number of children nodes

} ASTNode;

// Function to create a new AST node

ASTNode* createNode(char* value, int num_children) {

    ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));

    if (node == NULL) {

        perror("Memory allocation failed");

        exit(EXIT_FAILURE);

    }

    node->value = strdup(value);

    node->num_children = num_children;

    node->children = (ASTNode*)malloc(num_children * sizeof(ASTNode));

    if (node->children == NULL) {

        perror("Memory allocation failed");

        exit(EXIT_FAILURE);
```

```
    }
    return node;
}
// Function to generate code recursively from the AST
void generateCode(ASTNode* root) {
    // Base case: If the node has no children, simply print its value
    if (root->num_children == 0) {
        printf("%s ", root->value);
        return;
    }
    // If the node has children, recursively generate code for each child
    for (int i = 0; i < root->num_children; i++) {
        generateCode(root->children[i]);
    }
}
int main() {
    // Example AST representing a simple arithmetic expression: (a + b) * c
    ASTNode* rootNode = createNode("*", 2);
    rootNode->children[0] = createNode("+", 2);
    rootNode->children[0]->children[0] = createNode("a", 0);
    rootNode->children[0]->children[1] = createNode("b", 0);
    rootNode->children[1] = createNode("c", 0);
    // Generate code from the AST
    printf("Generated code: ");
```

```
    generateCode(rootNode);
    printf("\n");
    // Free memory
    // (In a real application, you would need to traverse the AST and free all nodes)
    free(rootNode->value);
    free(rootNode->children[0]->children[0]->value);
    free(rootNode->children[0]->children[1]->value);
    free(rootNode->children[0]->children[0]);
    free(rootNode->children[0]->children[1]);
    free(rootNode->children[0]->value);
    free(rootNode->children[0]);
    free(rootNode->children[1]->value);
    free(rootNode->children[1]);
    free(rootNode);
    return 0;
}
```

Sample input:

```
// Example AST representing a simple arithmetic expression: (a + b) * c
ASTNode* rootNode = createNode("*", 2);
rootNode->children[0] = createNode("+", 2);
rootNode->children[0]->children[0] = createNode("a", 0);
rootNode->children[0]->children[1] = createNode("b", 0);
rootNode->children[1] = createNode("c", 0);
```



Output: Generated code: a b + c *