

**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**Implementing a Compiler for Functional Programming Language**

**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**

**V. Krishna Sankar (192210630)**

**H. Deepak (192210627)**

**N. Harish (192211336)**

**Under the Supervision of**

**Dr. G. MICHAEL**

**September 2024**

**DECLARATION**

We, **V.Krishna Sankar, H.Deepak, N.Harish,** students of **‘Bachelor of Engineering in Department of Computer Science**’ in Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **Building Compiler for functional programming language** 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.

**V. Krishna Sankar (192210630)**

**H. Deepak (192210627)**

**N. Harish (192211336)**

**CERTIFICATE**

This is to certify that the project entitled **“Building Compiler for functional programming language”** submitted by, **V. Krishna Sankar, H. Deepak, N. Harish**  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.

Teacher-in-charge

Dr. G. MICHAEL

**Table of Contents**

|  |  |
| --- | --- |
| **S.NO** | **TOPICS** |
| 1. | **Abstract** |
| 2. | **Introduction** |
| 3. | **Problem Statement** |
| 4. | **Proposed Design**   * Requirement Gathering and Analysis * Tool selection criteria * Scanning and Testing Methodologies |
| 5. | **Phases of complier** |
| 6. | **Functionality**   * User Authentication and Role Based Access Control. * Tool Inventory and Management * Security and Compliance Control |
| 7. | **UI Design**   * Layout Design * Feasible Elements Used * Elements Positioning and Functionality |
| 8. | **Conclusion** |

**ABSTRACT:**

Creating compilers for functional programming languages presents a critical endeavor in advancing software development paradigms. This paper offers a comprehensive review of the process of building compilers for functional programming languages, elucidating its significance, methodologies, and impact on programming language ecosystems. The abstract delves into the intricacies of compiler construction, emphasizing its role in translating high-level functional code into executable machine instructions.

It outlines the diverse components involved in compiler design, including lexical analysis, syntax parsing, semantic analysis, optimization techniques, and code generation strategies, elucidating their respective functionalities and interdependencies. The abstract also discusses various compiler construction tools and frameworks, such as Lex and Yacc, LLVM, and GCC, highlighting their suitability and effectiveness in building compilers for functional languages.

Moreover, the abstract evaluates the benefits and challenges inherent in developing compilers for functional programming languages, emphasizing their ability to promote code correctness, modularity, and abstraction while addressing issues like performance optimization and type inference complexities. It further examines emerging trends and advancements in compiler technology, such as Just-In-Time compilation and functional programming language extensions, underscoring their potential to revolutionize software development practices and enhance language expressiveness.

**Introduction:**

In the realm of modern software development, the construction of compilers for functional programming languages stands as a pivotal endeavor. These languages, renowned for their emphasis on abstraction, modularity, and correctness, offer unique challenges and opportunities in the domain of compiler design. With the proliferation of functional programming paradigms in various software ecosystems, the need for robust compilers tailored to these languages has become increasingly apparent.

This investigation seeks to explore the intricate process of building compilers for functional programming languages, shedding light on the methodologies, tools, and innovations driving this transformative field of study. By delving into the complexities of compiler construction and examining emerging trends, this investigation aims to equip developers and language enthusiasts with the knowledge and insights necessary to navigate the terrain of functional language compilation effectively.

**Problem Statement:**

As functional programming languages gain prominence in software development, the demand for tailored compilers to support these languages grows. However, navigating the landscape of compiler construction presents a significant challenge. With a plethora of methodologies, tools, and innovations available, selecting the most suitable compiler framework for a given functional language becomes a daunting task for developers and organizations alike.

**Proposed Design:**

**Requirements Gathering and Analysis:** Engage in stakeholder interviews and surveys to ascertain the organization's needs regarding compiler functionality, language support, and performance requirements for the target functional programming language.

**Tool Selection Criteria:** Compile a list of compiler construction tools, considering language compatibility, optimization capabilities, and community support. Evaluate tools based on project objectives using industry research and expert recommendations.

**Scanning and Testing Methodology:** Define a systematic approach to compiler design, covering lexical analysis, parsing, semantic analysis, optimization, and code generation. Implement best practices and methodologies to ensure the compiler's robustness and efficiency for the chosen functional programming language.

**Phases of Compiler:**

### 1. ****Lexical Analysis:****

The first phase of compilation, known as **Lexical Analysis**, involves scanning the source code to break it down into meaningful sequences called **tokens**. Tokens are the smallest units in a program, such as keywords (e.g., if, while), identifiers (e.g., variable names), literals (e.g., numbers, strings), operators (e.g., +, -), and punctuation symbols (e.g., parentheses). The lexical analyzer, or lexer, reads the input character stream and groups these characters into tokens using a predefined set of rules, often expressed using regular expressions. During this process, the lexer also removes comments and unnecessary white spaces, which are irrelevant to the syntax but useful for code readability. If an invalid sequence of characters is encountered, the lexer generates a lexical error. The output of the lexical analysis phase is a stream of tokens, which serves as input for the next phase, **syntax analysis**. Lexical analysis is crucial because it transforms the raw text of the source code into a structured format that the compiler can process more efficiently.

### 2. ****Syntax Analysis (Parsing):****

After the lexical analysis, the token stream is passed to the **Syntax Analysis** phase, also called **Parsing**. The parser takes these tokens and arranges them into a hierarchical structure known as the **Abstract Syntax Tree (AST)**. This tree structure represents the grammatical structure of the program, according to the rules defined by the language's grammar. For example, a simple arithmetic expression like a + b would be parsed into a tree where + is the root node, and a and b are its children. Syntax analysis ensures that the sequence of tokens adheres to the formal grammar of the programming language. If the code violates the grammar (for example, through missing operators or misplaced parentheses), the parser generates syntax errors. The AST produced by this phase is an abstract representation that omits certain details (like parentheses) but retains the overall structure needed for further analysis and translation. Syntax analysis is critical as it verifies that the program's structure is well-formed before moving to the more complex semantic checks.

### 3. ****Semantic Analysis:****

Once the program's syntax has been validated, the **Semantic Analysis** phase begins. This phase ensures that the program is **semantically correct**—that is, it checks whether the operations in the program make sense based on the context. For instance, semantic analysis verifies that variables are declared before use, functions are called with the correct number of arguments, and types are consistent in expressions (e.g., ensuring an integer isn’t added to a string). The semantic analyzer may also check for issues such as type compatibility, scoping rules, and memory allocation, depending on the programming language. This phase often performs **type checking**, which ensures that the types of expressions match the expected types (e.g., ensuring that an integer isn't mistakenly assigned to a variable expecting a string). Semantic analysis may also include resolving identifier names to their definitions, ensuring that there are no naming conflicts or undefined references. Errors detected in this phase are **semantic errors**, and correcting them is essential for the correct execution of the program. The semantic analysis results in a modified AST that reflects the meaning of the program, ready for intermediate code generation.

### 4. ****Intermediate Code Generation:****

In the **Intermediate Code Generation** phase, the compiler translates the validated AST into an **Intermediate Representation (IR)**, which serves as a bridge between high-level source code and low-level machine code. The intermediate representation is a simplified, lower-level code that abstracts away machine-specific details, making it easier to perform optimizations and target different hardware architectures. The IR is typically a set of instructions similar to assembly language but more abstract, allowing the compiler to be portable across different machine architectures. This intermediate form might represent operations as a series of steps that are easier to manipulate during later phases, like optimization. For example, a high-level operation like a + b \* c could be translated into a sequence of intermediate instructions that handle the multiplication and addition separately. The goal of this phase is not to produce efficient or optimized code but rather to create a consistent and flexible representation of the program that can be further refined in the next stages. The IR can come in different forms such as **Three-Address Code (TAC)**, **Static Single Assignment (SSA)**, or other intermediate formats used depending on the compiler's design.

### 5. ****Optimization:****

The **Optimization** phase improves the intermediate code's efficiency, focusing on reducing execution time, memory usage, or power consumption without altering the program's behavior. Optimizations can be either **machine-independent** (global optimizations) or **machine-specific** (targeted optimizations for specific hardware). Machine-independent optimizations include techniques such as **constant folding**, where constant expressions are evaluated at compile time rather than runtime, and **dead code elimination**, which removes parts of the code that are never executed or do not affect the program's output. Other common optimizations include **loop unrolling** (expanding loop iterations to reduce overhead), **inlining** (replacing function calls with the function's body), and **strength reduction** (replacing expensive operations with cheaper ones). Machine-specific optimizations depend on the target architecture and might involve using special hardware features or adjusting instruction sequences to maximize pipeline efficiency. This phase is crucial for improving the performance of the final program, as naive code generation often produces inefficient code. However, optimization is an optional phase; some compilers might skip it for the sake of simplicity or speed of compilation.

### 6. ****Code Generation:****

In the **Code Generation** phase, the optimized intermediate representation is translated into **target machine code** or **assembly code**. This phase is highly dependent on the architecture of the target machine, as the instructions generated must be compatible with the hardware's instruction set. For example, for a processor with an x86 architecture, the generated code will consist of x86 assembly instructions. The code generation process must ensure that all high-level constructs (such as loops, conditionals, and function calls) are correctly translated into machine-level instructions that the processor can execute. The code generator also handles **register allocation**, determining which values should be stored in the limited number of CPU registers and which should be stored in memory. Additionally, the code generator must ensure that resources like memory and processor registers are efficiently utilized. The generated machine code might still include symbolic references to external libraries or variables, which will be resolved during the linking phase. The quality of this phase impacts the program's runtime performance, as poorly generated machine code can result in slower execution or excessive memory use.

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

typedef enum { NUMBER, VARIABLE, LAMBDA, APPLY } ExprType;

typedef struct Expr {

ExprType type;

int value;

char varName;

struct Expr \*param, \*body;

struct Expr \*func, \*arg;

} Expr;

Expr \*makeNumber(int value) {

Expr \*e = (Expr \*)malloc(sizeof(Expr));

e->type = NUMBER;

e->value = value;

return e;

}

Expr \*makeVariable(char varName) {

Expr \*e = (Expr \*)malloc(sizeof(Expr));

e->type = VARIABLE;

e->varName = varName;

return e;

}

Expr \*makeLambda(Expr \*param, Expr \*body) {

Expr \*e = (Expr \*)malloc(sizeof(Expr));

e->type = LAMBDA;

e->param = param;

e->body = body;

return e;

}

Expr \*makeApply(Expr \*func, Expr \*arg) {

Expr \*e = (Expr \*)malloc(sizeof(Expr));

e->type = APPLY;

e->func = func;

e->arg = arg;

return e;

}

int interpret(Expr \*expr) {

switch (expr->type) {

case NUMBER:

return expr->value;

case APPLY: {

int argValue = interpret(expr->arg);

return argValue + 1;

}

default:

printf("Unsupported expression type!\n");

exit(1);

}

}

int main() {

Expr \*lambdaExpr = makeLambda(makeVariable('x'), makeNumber(1));

Expr \*applyExpr = makeApply(lambdaExpr, makeNumber(2));

int result = interpret(applyExpr);

printf("Result: %d\n", result);

return 0;

}

**Functionality:**

**User Authentication and Role-Based Access Control:**

* Implement user authentication measures to manage access to the compiler system.
* Define roles and permissions to control access based on user responsibilities and authorization levels, ensuring secure interaction with the compiler's functionalities.

**Tool Inventory and Management:**

* Maintain a centralized catalog of functional programming language development tools, including vendor information, version numbers, and license status.
* Streamline tool management processes such as installation, configuration, and updates, ensuring seamless integration with the compiler development environment.

**Security and Compliance Controls:**

* To safeguard sensitive data, implement robust security measures such as encryption, access controls, and comprehensive audit trails to ensure compliance with relevant standards and regulations.

**Architectural Design:**

**Presentation Layer:**

* Develop a web-based user interface (UI) tailored for interacting with the assessment framework specific to the functional programming language.
* Implement role-based access control (RBAC) to manage user authentication and permissions within the compiler system.

**Application Layer:**

* The business logic layer is dedicated to processing user requests and orchestrating system functionality tailored for the functional programming language.
* The criterion management module is designed to define, store, and manage assessment criteria specifically relevant to the functional programming paradigm within the compiler system.

**Monitoring and Management Layer:**

* Integrate tools for real-time performance monitoring, log analysis, and system health checks optimized for the requirements of the functional programming language compiler.
* Utilize platforms for centralized and aggregated storage and analysis of system logs, ensuring seamless management and insight generation tailored to the compiler's specific needs.

**UI Design:**

**Dashboard:**

* Tiles/cards displaying key metrics about the compilation process, such as the number of source files compiled, errors encountered, and compilation time.
* System status indicators indicating the current state of the compiler e.g. idle, compiling, or error.

**User Management:**

* User account management interface allowing administrators to create, edit, and delete user accounts.
* Role assignment functionality enabling administrators to assign roles to users and define their permissions.

**Help and Support:**

* Help documentation section accessible from the dashboard, containing user manuals, guides, and FAQs.
* Support contact information displayed prominently, allowing users to reach out for assistance when needed.

**Element Positioning and Functionality:**

**Real-time Monitoring:**

* Positioned on the dashboard to provide real-time monitoring of the compilation process.
* Widgets or progress bars display live updates on compilation progress, including the number of files processed, errors encountered, and compilation speed.

**Collaboration Features:**

* Available within the compiler environment, allowing users to collaborate on source code files.
* Features such as comments, annotations, or version control support facilitate collaboration among compiler developers and testers.

**Trend Analysis:**

* Positioned in the reporting and analysis section, offering insights into the compiler's performance.
* Interactive charts or graphs visualize compilation metrics over time, such as compilation speed, error trends, and resource utilization.

**Conclusion:**

Overall, a well-designed UI for a compiler for a functional programming language streamlines the compilation process, supports collaboration and knowledge sharing, and empowers users with the tools and resources needed to effectively compile and manage code.

**References:**

* Aho, Alfred V., et al. "Compilers: Principles, Techniques, and Tools." Pearson Education, 2006.
* Appel, Andrew W. "Modern Compiler Implementation in ML." Cambridge University Press, 1998.
* Jones, Simon Peyton. "The Implementation of Functional Programming Languages." Prentice Hall, 1987