C Compiler Design using Lex, Yacc & Python

Nithin S
221IT085
Department of Information Technology, National Institute of Technology Karnataka, Surathkal

Jay Chavan
211IT020
Department of Information Technology, National Institute of Technology Karnataka, Surathkal

Ayush Kumar
211IT015
Department of Information Technology, National Institute of Technology Karnataka, Surathkal

***Abstract*—This paper describes the working of the C Compiler designed in Lex, Yacc and Python, which shows the phase by phase execution of a C Code in a Compiler**

Keywords—C Compiler, Lex, Yacc, Python, Compiler Design, Compiler Design Phases, Grammar, Regular Expressions, Scanner, Optimization, Parser

I. Introduction

Before 1975 writing a compiler was a very time-consuming process. Then Lesk [1975] [6] and Johnson [1975] published papers on lex and yacc. These utilities greatly simplify compiler writing. Implementation details for lex and yacc may be found in Aho [1986]. Lex and yacc are available from.Lex will read your patterns and generate C code for a lexical analyzer or scanner. The patterns in the diagram is a file you create with a text editor. Lex will read your patterns and generate C code for lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representation of strings, and simplify processing. When a lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.Yacc will read your grammar and generate C code for a syntax analyzer or parser. The grammar in the above diagram is a text file you create with a text editor. The syntax analyzer uses grammar rules to allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure of the tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language code.

We'll assume our goal is to write a Basic Compiler. First, we need to specify all pattern matching rules for lex (bas.l) and grammar rules for yacc (bas.y). Commands to create our compiler, bas.exe, are listed below:

```
yacc –d bas.y                    # create y.tab.h, y.tab.c
lex bas.l                        # create lex.yy.c
cc lex.yy.c y.tab.c –obas.exe   # compile/link
```

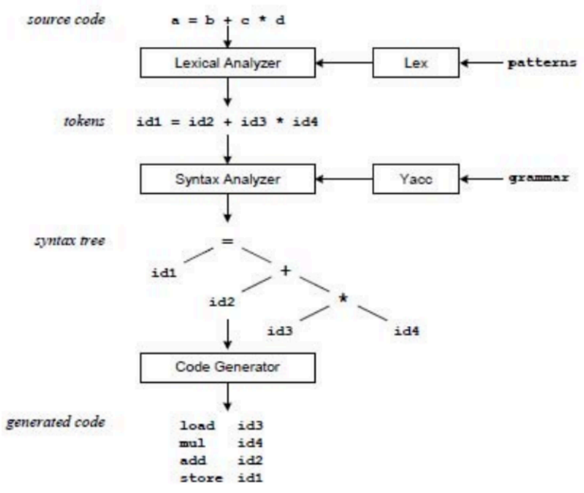

Figure 1 : Compilation Sequence

Yacc reads the grammar descriptions in bas.y and generates a syntax analyzer (parser), that includes function yyparse, in file y.tab.c. Included in file bas.y are token declarations. The –d option causes yacc to generate definitions for tokens and place them in file y.tab.h. Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c. Finally, the lexer and parser are compiled and linked together to create executable bas.exe. From main we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

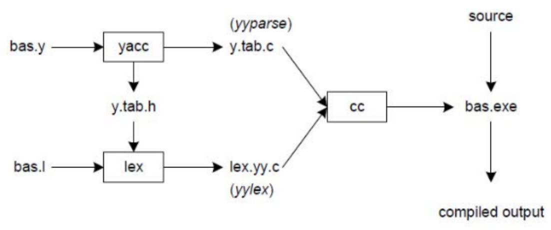

Figure 2 : Building Compiler with Lex and Yacc

With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser.

II. LITERATURE SURVEY

In the field of natural language processing and compiler design, it's been amazing converting English-like words into practical mathematical processes. The development of this discipline may be traced back to several seminal studies that changed our understanding and capabilities.

M. Upadhyaya et al. [1] set out to create a simple calculator compiler using the well-known tools Lex and YACC. This endeavor was motivated by a desire to bridge the gap between plain language and mathematical computing. They successfully tokenized arithmetic expressions using regular expressions and generated grammar rules in YACC by leveraging Flex for lexical analysis and YACC for parsing. Their work demonstrates the power of these technologies, particularly when it comes to dealing with faults and guaranteeing accurate compilation. Brown et al. [2] built on this basis to provide a full introduction to the world of Lex and YACC. Their research goes deeply into fundamental principles, offering insight on advanced topics like mistake recovery and optimization.

Waite [3] expanded on this underlying knowledge by demonstrating how Lex and YACC could be used to construct a full compiler. Waite's research did not end there; he also looked at the possibilities of other parser generation tools, most notably ANTLR.

Speaking of ANTLR, Ortín et al. [4] and another study in 2019 [5] both embarked on a comparative analysis between Lex/YACC and ANTLR. Their findings were consistent in highlighting ANTLR's superiority in terms of speed, accuracy, ease of use, and flexibility. Such comparative studies are crucial in guiding researchers and developers in choosing the right tools for their specific applications.

Diving into the technical intricacies, a team [6] presented a detailed report on the challenges and solutions related to parsing and scanning MATLAB code within the MATCH compiler. Their insights are invaluable for those looking to develop compilers for MATLAB and other dynamically typed languages.

For those new to the field, Ikotsireas [7], a publication in 2018 [8], and ShareOK [9] all offer concise introductions to Lex and YACC. These works cover the fundamental concepts and provide practical examples, making them perfect starting points for budding researchers.

For those seeking a more comprehensive understanding, "A Compiler-based Approach for Natural Language to Code Conversion" [10] by S. Sridhar and S. Sanagavarapu. This work promises a deep dive into how NLP can be integrated with a compiler for more user accessibility.

III. METHODOLOGY

The work, "Mathematical Operations Using English-Like Sentences," builds a system that reads and executes mathematical operations described in English-like words using Lex and YACC tools. This work's technique is outlined below:

A. Lexical Analysis with Lex: The first step is to use Lex, a lexical analysis tool. The process of breaking down incoming text into meaningful pieces known as Tokens are known as lexical analysis. Tokens can represent numbers, arithmetic operations, keywords, punctuation marks, and other information.

Lex identifies these tokens using a set of predefined regular expressions. Each token is assigned to a certain token type, which is then used in the parsing process. The Lex code has numerous tokens, each associated with a specific mathematical operation or command, such as ADD for addition, SUBTRACT for subtraction, MULTIPLY for multiplication, DIVIDE for division, MODULO for modulus operation, and several others.

B. Parsing with YACC: The second step is parsing, it is done with YACC (Yet Another Compiler Compiler). Parsing involves developing a grammar for English- like sentences. Grammar rules determine sentence structure, including the order of operations, operands, and keywords.

The YACC code defines these language rules as well as the actions that must be taken when a rule is matched. The grammar contains rules for adding numbers, removing numbers from operands, multiplying numbers, dividing the result by a number, and modulus operations on the result by a number.

C. Syntax and Semantics: The grammatical rules created during the parsing process ensure that the English-like sentences are formatted correctly. Through related actions, the semantics of the phrases are captured. When the tool sees a "ADD" statement, for example, it conducts the addition operation and modifies the output accordingly. Similarly, the grammar sets rules for various arithmetic operations and control flow statements, allowing them to be executed and their impact on the entire computation to be evaluated.

D. Conditional Statements: Conditional statements are also supported by the system, allowing users to execute commands based on situations. The YACC code defines the syntax for an "if" statement, which includes conditions for less than, greater than, equal to, and not equal to. These conditions' actions are also described, allowing the system to undertake different activities dependent on the condition's outcome.

E. Integration and Testing: After developing the lexical analyzer and parser, they are integrated and tested. The goal is for the system to be able to correctly comprehend and execute English-like statements describing mathematical operations. User feedback is collected and used to make any necessary system improvements. Table 1 illustrates how Operations are treated as Tokens.

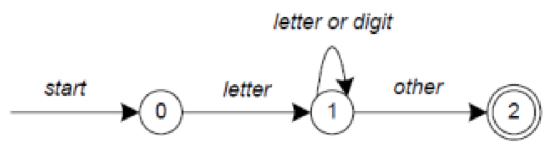

Figure 3 : Finite State Machine

IV. IMPLEMENTATION

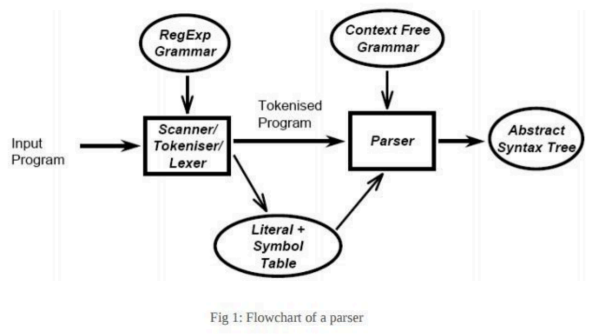

FIgure 4: Flowchart of a parser

The implementation of each phase of the compiler involves a systematic approach to handling different aspects of source code processing, analysis, and transformation.

Starting with the lexical analyzer, the focus lies on pattern recognition and tokenization. Regular expressions are utilized to define the lexical structure of the programming language, enabling the identification and extraction of tokens such as keywords, identifiers, literals, and symbols. The Lex tool provides a convenient way to specify these patterns and generate efficient lexical analyzers automatically.

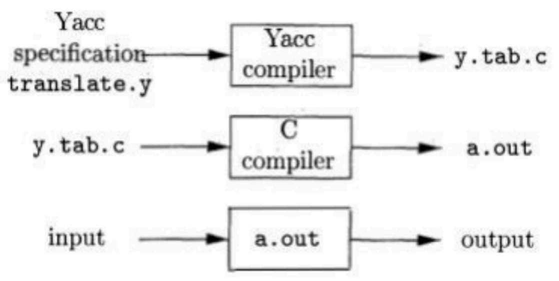

FIgure 5: Flowchart of a Lex

Moving on to the syntax analyzer, the emphasis shifts towards parsing the sequence of tokens to verify the syntactic correctness of the source code. This phase typically involves constructing a parse tree or an abstract syntax tree (AST) to represent the hierarchical structure of the program. Using tools like Yacc, grammar rules are defined to guide the parsing process, ensuring adherence to the language's syntax rules. Error handling mechanisms are integrated to detect and report syntax errors encountered during parsing.

Following syntax analysis, the semantic analyzer delves deeper into the meaning and context of the source code. This phase involves type checking, scope resolution, and semantic rule enforcement to ensure program correctness and consistency. Symbol tables are constructed to store information about identifiers and their attributes, facilitating semantic analysis and code generation. Semantic actions embedded within the parsing rules are executed to perform semantic checks and annotate the AST with additional information.

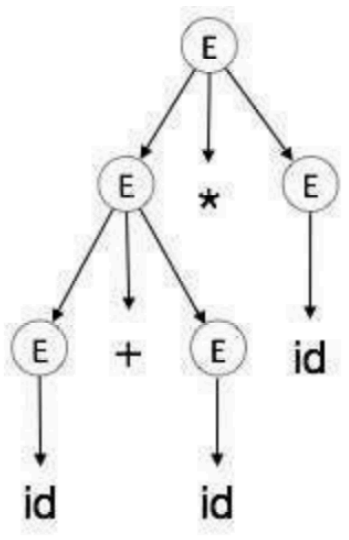

FIgure 6: Sample Parse Tree

The intermediate code generator is responsible for translating the high-level source code into an intermediate representation (IR) that is independent of the source and target languages. This IR serves as a bridge between the front-end and back-end of the compiler, facilitating optimization and target code generation. Three-address code or a similar intermediate representation is typically employed, capturing the essential semantics of the source code while abstracting away language-specific details.

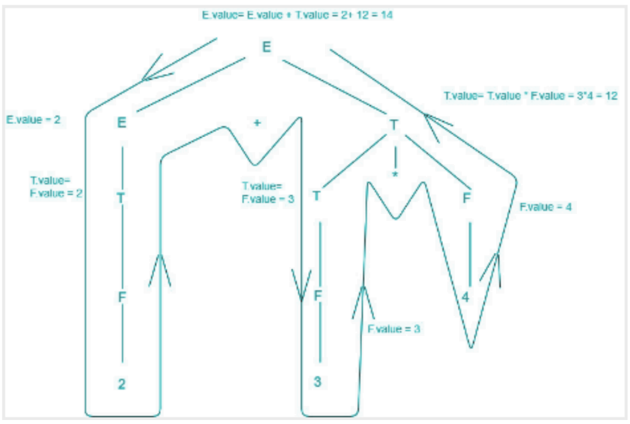

FIgure 7: Preorder of a Parse Tree

Code optimization is a critical phase aimed at improving the efficiency and performance of the generated code. Various optimization techniques are applied to the intermediate representation to eliminate redundant computations, reduce code size, and enhance runtime performance. These optimizations may include constant folding, common subexpression elimination, loop optimization, and control flow restructuring. The effectiveness of optimization strategies is evaluated through analysis of code metrics and performance benchmarks.

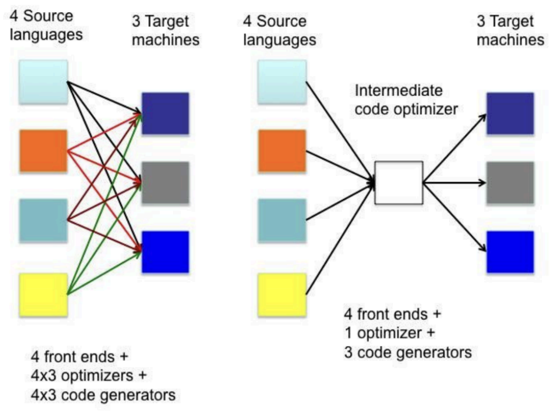

Figure 8 : Schema of TCG

Finally, the target code generator translates the optimized intermediate code into machine-readable instructions specific to the target architecture. This phase involves instruction selection, register allocation, and code scheduling to generate efficient assembly or machine code. Target-specific optimizations may be applied to further enhance code performance and exploit architectural features. The generated code is then assembled and linked to produce the final executable program.

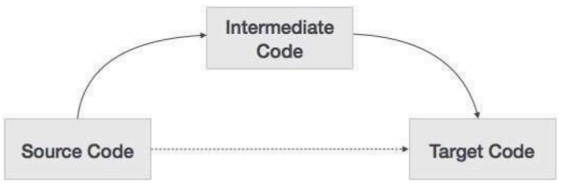

FIgure 9:   ICG and TCG

Throughout the implementation process, rigorous testing and validation are essential to ensure the correctness, reliability, and efficiency of the compiler. Unit tests, integration tests, and regression tests are conducted to verify the functionality of each phase and detect any regressions or defects. Continuous integration and automated testing pipelines streamline the testing process and facilitate early detection of errors. Documentation and user guides are prepared to aid developers and users in understanding and utilizing the compiler effectively. Collaboration with the programming language community and open-source contributions foster innovation and drive continuous improvement in compiler technology.

The entire code for the project can be found in the Final Report as well as our github repo.

V. ANALYSIS

In the realm of analysis within compiler development, a multifaceted approach is essential to comprehensively evaluate the performance, correctness, and usability of the compiler across various dimensions.

Performance analysis involves assessing the efficiency of each phase of the compiler in terms of execution time, memory usage, and scalability. Profiling tools such as Valgrind, gprof, and perf are employed to measure resource consumption and identify performance bottlenecks. By profiling the compiler on a diverse set of input programs ranging from small snippets to large-scale applications, developers can gain insights into areas for optimization and enhancement. Benchmark suites like SPEC CPU, LLVM Test Suite, and NAS Parallel Benchmarks provide standardized workloads for evaluating compiler performance across different architectures and optimization levels. Additionally, instrumentation techniques such as timing and memory profiling enable fine-grained analysis of individual components within the compiler pipeline.

Correctness analysis focuses on ensuring the accuracy and reliability of the compiler's output. Formal verification techniques such as static analysis, model checking, and theorem proving are employed to mathematically verify the correctness of compiler transformations and optimizations. Tools like LLVM's bugpoint and Csmith assist in generating test cases to uncover compiler bugs and corner cases. Furthermore, differential testing compares the output of the compiler under test with that of a reference compiler to detect discrepancies and potential errors. Symbolic execution and constraint solving techniques enable symbolic testing of the compiler against a set of symbolic inputs, uncovering bugs that may not be revealed by concrete testing alone.

Furthermore, comparative analysis compares the performance, features, and limitations of the developed compiler with existing compiler frameworks and toolchains. This includes evaluating factors such as compilation speed, generated code quality, optimization capabilities, and language support. Comparative studies shed light on the strengths and weaknesses of the compiler.

In summary, analysis within compiler development encompasses a spectrum of activities aimed at evaluating the performance, correctness, usability, and competitiveness of the compiler. Through rigorous testing, verification, benchmarking, and user feedback, developers can iteratively refine and enhance the compiler to meet the evolving needs of developers and users in the software development ecosystem.

VI. CONCLUSION

With the lex and yacc tool one can create its own compiler, wherever one is required. It is basically procedural language compiler tools and to support object oriented one need to work on structure of C language to support object oriented which makes the compiler quite complex. To use lex and yacc on UNIX is easy as compared to other operating systems. Gcc is the basic compiler to generate the executable from lex and yacc compiled files. By studying these tools one can understand the basic structure of the compiler designed in C and go forward from it.

VII. FUTURE SCOPE

The future scope of compiler research and development holds exciting possibilities for advancing the state-of-the-art in programming language implementation, optimization techniques, and tooling support. Several key areas present opportunities for innovation and exploration:

Machine Learning-Based Optimization: Integrating machine learning techniques into compiler optimization opens up new avenues for automatic program analysis and transformation. Deep learning models can be trained to recognize patterns in code and make informed decisions for optimization, such as loop unrolling, vectorization, and parallelization. Reinforcement learning algorithms can dynamically adapt optimization strategies based on feedback from program execution, leading to more efficient and adaptive compilers.

Quantum Computing Compilation: With the advent of quantum computing, there is a pressing need for compilers tailored to quantum programming languages and architectures. Compiler research in this domain focuses on translating high-level quantum algorithms into executable quantum circuits, optimizing quantum gate placement and resource utilization, and addressing error correction and noise mitigation challenges. Quantum-aware compilation techniques aim to exploit the unique properties of quantum hardware to maximize performance and scalability.

Heterogeneous Computing Compilation: As computing architectures become increasingly heterogeneous, compilers must support efficient code generation for diverse hardware accelerators such as GPUs, TPUs, and FPGAs. Compiler research in this area focuses on polyhedral optimization techniques, domain-specific languages (DSLs), and compiler frameworks for expressing and optimizing parallelism across heterogeneous computing resources. Additionally, tools for automatic offloading and task scheduling help leverage the full potential of heterogeneous systems for accelerating compute-intensive workloads.

High-Level Synthesis (HLS): HLS compilers enable the synthesis of hardware designs from high-level programming languages such as C, C++, and OpenCL. Future research in HLS focuses on improving the productivity, performance, and scalability of hardware design synthesis by integrating advanced optimization techniques, automated design space exploration, and architectural modeling. HLS tools that target emerging technologies such as neuromorphic computing and approximate computing enable rapid prototyping and exploration of novel hardware architectures.

Domain-Specific Compilation: Domain-specific languages (DSLs) and compilers tailored to specific application domains enable developers to express complex algorithms and optimizations concisely and efficiently. Future research in domain-specific compilation focuses on language design, compiler optimization, and tooling support for emerging domains such as machine learning, data analytics, bioinformatics, and quantum computing. Additionally, techniques for cross-domain optimization and interoperability facilitate the integration of domain-specific languages with general-purpose programming languages and libraries.

Compiler Tooling and Developer Productivity: Improving compiler tooling and developer productivity is essential for enhancing the software development experience. Future research in this area includes the development of interactive programming environments, intelligent code completion and refactoring tools, and compiler diagnostics for detecting common programming errors and performance bottlenecks. Integration with IDEs, version control systems, and collaborative development platforms enhances collaboration and code quality assurance.

VIII. INDIVIDUAL CONTRIBUTIONS

A) Nithin S : Implementation, Report
B) Ayush Kumar:: Analysis, Report
C) Jay Chavan: Report, Analysis

IX. ACKNOWLEDGEMENTS

This project titled “C Compiler Design using Lex, Yacc & Python” was an opportunity for us to explore the field of Compiler Design. We would like to extend our heartfelt thanks to **Prof. Anupama**. for providing us with invaluable guidance and support throughout the course of this project. Her constant encouragement and constructive feedback helped us stay focused and motivated.

X. REFERENCES

[1] M. Upadhyaya, "Simple calculator compiler using Lex and YACC," 2011 3rd International Conference on Electronics Computer Technology, Kanyakumari, India, 2011, pp. 182-187, doi: 10.1109/ICECTECH.2011.5942077.

[2] D. A. Ladd and J. C. Ramming, "A*: a language for implementing language processors," in IEEE Transactions on Software Engineering, vol. 21, no. 11, pp. 894-901, Nov. 1995, doi: 10.1109/32.473218.

[3] A. N. Likhith, K. Gurunadh, V. Chinthapalli and M. Belwal, "Compiler For Mathematical Operations Using English Like Sentences," 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), Bangalore, India, 2023, pp. 1-6, doi: 10.1109/CSITSS60515.2023.10334225

[4] M. Mernik and V. Zumer, "An educational tool for teaching compiler construction," in IEEE Transactions on Education, vol. 46, no. 1, pp. 61-68, Feb. 2003, doi: 10.1109/TE.2002.808277.

[5] S. Sridhar and S. Sanagavarapu, "A Compiler-based Approach for Natural Language to Code Conversion," 2020 3rd International Conference on Computer and Informatics Engineering (IC2IE), Yogyakarta, Indonesia, 2020, pp. 1-6, doi: 10.1109/IC2IE50715.2020.9274674