*A Report*

*On*

ECHO

*Submitted to*

***University of Petroleum and Energy Studies***

*In Partial Fulfilment for the award of the degree of*

BACHELORS IN TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING (with specialization in CCVT)

By

Manav Verma

500091766

*Under the guidance of*

*Saurabh Shanu Sir*



**University of Petroleum and Energy Studies**

**Dehradun-India**

November 2023

|  |  |
| --- | --- |
| **S. No** | **Content** |
| 1. | Compiler Details |
| 2. | Compiler Basics |
| a) | Problem Statement |
| b) | Background |
| c) | Motivation for the compiler |
| d) | Objective |
| e) | Sub-Obejctive |
| 3. | Mode of achieving objective |
| 4. | Methodology |
| a) | Theoretical framework – explains the model or the set of theories related to the compiler. |
| b) | Sources of data – Primary or secondary data |
| c) | Schematic flow Diagram |
| 5. | Review of Literature |
| 6. | Key Bibliography |

# Compiler Details of Echo

* “echo” is a C-based developer-friendly programming language designed for simplicity and ease of understanding. It is designed to efficiently translate high-level source code into an optimized structured table.
* It focuses on achieving fast compilation times and generating reliable executables. The modular architecture of "echo" comprises essential components, such as a lexer for efficient tokenization, a parser ensuring adherence to grammar rules, and semantic analysis for deeper code understanding.
* Robust error handling, coupled with clear messages and graceful recovery strategies, enhances the compiler's reliability.
* Rigorous testing, including both unit and integration tests, ensures the efficiency and dependability of "echo."
* Despite facing challenges related to language complexity during development, "echo" has evolved into a refined, robust, and efficient C-based compiler.
* It stands as a dependable tool for translating source code into semantically correct, optimized structured tables.

Parsing Technique used (Bottom-Up Parsing):

The code is implementing bottom-up parsing using Yacc, a tool for generating parsers. It starts from the input symbols and works towards the start symbol of the grammar.

LALR Parsing:

* Look-Ahead: LALR parsers, including the one generated by Yacc, use a fixed number of lookahead symbols to make parsing decisions. In the code, lookahead symbols are utilized implicitly in the parsing decisions made by the generated parser based on the grammar rules.
* Left-to-Right Reading: The parser, as specified in LALR parsing, reads the input symbols from left to right. This left-to-right reading is inherent in the nature of how the parser processes the input tokens and constructs the parse tree.
* Rightmost Derivation (Reiteration): In LALR parsing, the rightmost derivation is emphasized. The parser builds the parse tree from the leaves (input symbols) towards the root (start symbol). This aligns with the structure of the code, where the parser processes the input symbols and constructs the parse tree following the rightmost derivation.

Keywords unique to ECHO language:

“Echo”: A keyword, corresponds to void.

Loop Keywords:

* loop: Represents a loop construct.
* whilst: Indicates the beginning of a while loop.

Control Flow Keywords:

* interrupt: Represents an interrupt statement.
* resume: Marks the resumption continue in the code.
* condition: Signifies the beginning of a conditional block.
* else: Used in conjunction with condition for an alternative code block.
* result: Indicates a result statement.

Data Type Keywords:

* number: Represents a numeric data type.
* character: Represents a character data type.
* decimal: Represents a decimal data type.
* precision: Represents a precision data type.

Modifier Keywords:

* lengthy: Used as a modifier for a data type (e.g., lengthy number).
* very lengthy: Used as a modifier for a data type (e.g., very lengthy number).
* short: Used as a modifier for a data type (e.g., short number).
* negative: Represents the negative sign.
* positive: Represents the positive sign.

Grammer used (Type 2):

A context-free grammar (CFG) is a formal grammar (Type 2 in the Chomsky hierarchy) used to describe the syntax of a formal language. It comprises terminals (basic symbols), non-terminals (representing sets of strings), production rules (specifying replacements for non-terminals), and a start symbol (representing the entire language). Production rules define the syntactic structure.

A -> B

**Terminal Symbol -> Terminal and Non terminal symbols**

For instance, an arithmetic expression CFG might have rules like:

E → E + T | T

T → T \* F | F

F → (E) | id

Here, E, T, F are non-terminals, and +, \*, (, ), id are terminals. The start symbol is E. These rules dictate how expressions, terms, and factors can be constructed.

# Compiler Basics

A compiler translates C source code into machine-readable instructions through multiple phases. Initially, lexical analysis breaks the code into tokens, stripping whitespace and comments. Parsing ensures correct syntax and generates object code. The symbol table, a vital data structure, stores information about variables and functions, aiding both analysis and synthesis. Our implementation uses lex for C's lexical analysis, producing a token stream with types and line numbers. Error indications for comments aid debugging. The included symbol table enhances understanding by providing information about tokens and types in the compiled code.

## Problem Statement

The need for the ECHO compiler arises from the demand for a programming language that prioritizes simplicity and ease of understanding. In a landscape where developers seek straightforward syntax and semantics, existing languages often pose challenges due to their complexity. The decision to create the ECHO compiler stems from recognizing these challenges and the desire to offer a developer-friendly alternative. As the ECHO language, built using C, gains traction for its simplicity, a reliable compiler becomes essential to seamlessly translate high-level source code into optimized structured tables.

## Background

Traditional languages, while powerful, often introduce complexities in syntax and semantics, posing challenges for programmers aiming for straightforward and intuitive code. Recognizing this gap, the ECHO programming language emerges as a solution designed explicitly to cater to the needs of developers seeking a user-friendly and accessible programming experience. The goal is to provide developers with an efficient tool that facilitates fast compilation times, robust error handling, and error-free executables. The factors driving the development of the ECHO compiler include the demand for simplicity in programming languages and the need for a reliable solution that aligns with these preferences.

## Motivation/need for the compiler

The development of the "echo" compiler also served as a valuable learning process, involving a comprehensive understanding of each phase of the compiler. This hands-on experience was crucial for gaining insights into the intricacies of compiler construction. Beyond addressing the challenges posed by the complexities of C programming, the project provided an opportunity for personal growth and skill development. By creating a compiler from scratch, the developer not only aimed to overcome language intricacies but also honed their proficiency in compiler design, contributing to a deeper understanding of the intricacies of programming languages.

## Objective

The primary objective of the "echo" compiler is to efficiently translate high-level C source code into optimized structured tables, emphasizing fast compilation times and the generation of reliable, error-free executables.

## Sub-Objectives

* Efficiency in Compilation: Optimize algorithms and processes for swift translation of source code to machine-readable format.
* Reliability of Executables: Ensure correctness and stability in the compiled code, minimizing errors and unexpected behaviors.
* Modular Architecture: Implement a modular structure with components like a lexer, parser, and semantic analyzer for maintainability and extensibility.
* Robust Error Handling: Develop robust mechanisms for error detection, providing clear messages and graceful recovery strategies.
* Testing Procedures: Conduct thorough unit and integration tests to verify the efficiency and dependability of the compiler.
* Evolution and Refinement: Continuously improve the compiler based on user feedback and emerging language features, evolving it into a refined and efficient tool.
* Dependability: Establish the compiler as a dependable tool, consistently translating source code into semantically correct, optimized tables.

# 

# Mode of achieving objective

# To fulfill its objectives, the ECHO compiler adopts a systematic approach that leverages a modular architecture. The Lexer plays a pivotal role in tokenizing the source code, enabling efficient processing. The Parser enforces well-defined C grammar rules, constructing a coherent syntax tree that serves as the foundation for subsequent stages. Semantic Analysis is employed to guarantee code correctness, ensuring that the translated code aligns with the intended logic.

# The compiler places a strong emphasis on robust error handling, providing clear and informative messages to developers. Additionally, it incorporates graceful recovery strategies to minimize disruptions in the compilation process, enhancing user experience.

# A comprehensive testing strategy is integral to the ECHO compiler's development. Unit tests are meticulously designed for individual components, validating their functionality in isolation. Optimization techniques are applied to the generation of structured tables, ensuring that the compiled code is not only correct but also optimized for efficiency.

# By combining these elements, the ECHO compiler strives to achieve its objectives of delivering a reliable, efficient, and developer-friendly tool for translating high-level source code into optimized structured tables. This approach is designed to meet the demands of simplicity and ease of understanding while maintaining a high standard of performance and correctness.

# Methodology

**Requirements Analysis:**

Objective: Understand the specific needs and goals of the compiler, considering fast compilation, reliability, and optimized code generation.

**Design Modular Architecture:**

Objective: Develop a modular structure to facilitate component-based development and maintenance.

Activities:

* Design the Lexer to tokenize source code.
* Implement the Parser to analyze syntax and construct a syntax tree.
* Develop Semantic Analysis for type checking and correctness verification.

**Implement Robust Error Handling:**

Objective: Ensure effective identification and resolution of errors during compilation.

Activities:

* Design clear error messages for various scenarios.
* Implement strategies for graceful error recovery.

**Conduct Rigorous Testing:**

Objective: Verify the correctness, reliability, and efficiency of the compiler.

Activities:

* Perform unit tests on individual components (Lexer, Parser, Semantic Analysis).
* Conduct integration tests to validate interactions between components.
* Execute performance tests to optimize and evaluate compilation speed.

**Refine Compiler for Language Complexities and Structured Tables:**

Objective: Address intricacies of the C language and optimize the generated structured tables.

## Theoretical framework

The theoretical framework of a compiler serves as the intellectual foundation, incorporating fundamental models and theories that inform its design and construction. Here's a detailed breakdown:

* Formal language theory provides the basis for understanding the syntax and structure of programming languages. Concepts such as regular and context-free grammars, as well as formal language hierarchies, guide the definition of valid language constructs.
* Automata theory explores abstract machines and their capabilities in language recognition. Finite automata and pushdown automata are often employed to model the lexical and syntactic aspects of compilers, aiding in tokenization and parsing.
* Parsing algorithms, such as LALR (Look-Ahead LR) and LR (Left-to-Right), are instrumental in analyzing the syntactic structure of source code. These algorithms facilitate the construction of parse trees or syntax trees, aiding in the comprehension of code structure.
* Semantic analysis involves understanding the meaning of code beyond its syntax. Principles from this theory guide the creation of symbol tables, type-checking mechanisms, and other processes that ensure the correctness and meaningful interpretation of the code.
* Theoretical frameworks address strategies for effective error handling during compilation. Clear understanding of error recovery mechanisms, syntax-directed error reporting, and strategies for maintaining program integrity contribute to robust error handling.

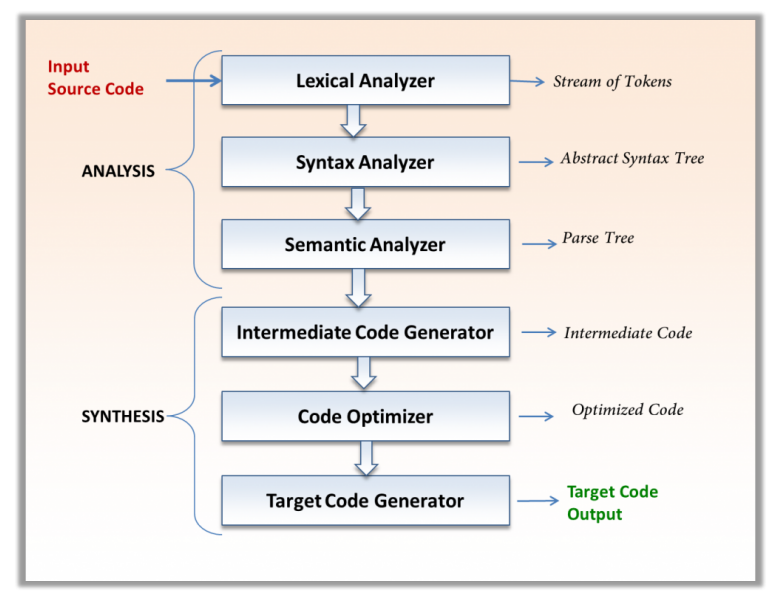
## Sources of data – Primary or secondary data:

In the context of the provided information about the "echo" compiler and its development, it seems like the details are based on primary data. Primary data refers to information collected firsthand from original sources. In this case, details about the compiler's architecture, components (such as Lexer, Parser, Semantic Analysis), error handling strategies, testing methodologies, and optimization techniques appear to be specific to the development process of the "echo" compiler.

The information which is derived from existing documents, publications, or external sources, it would be considered secondary data. Secondary data involves the use of pre-existing data for analysis or reference. (Refer to the bibliography)

## Schematic flow Diagram

The schematic flow diagram visually represents the sequential process of the compiler. It typically includes stages such as lexical analysis, parsing, semantic analysis and error handling. The arrows indicate the flow of data and control between these stages, illustrating the transformation of high-level source code into optimized machine code. The flow diagram serves as a comprehensive guide for understanding the compiler's internal structure.



# 

**Achieved by “ECHO”**

# Review of literature

**Compiler Construction Principles:**

Explore foundational literature on compiler construction, covering lexical analysis, parsing, semantic analysis, and code generation, to understand established methodologies.

**Efficiency Strategies:**

Investigate research on techniques for achieving fast compilation times, including optimizations and parallel processing, to enhance the efficiency of the "echo" compiler.

**Reliability and Error Handling:**

Examine studies emphasizing reliable compiler design, focusing on robust error handling and graceful recovery mechanisms to produce error-free executables.

**Modular Architecture Best Practices:**

Study literature on modular compiler architectures, particularly the design of lexers, parsers, and semantic analyzers, to implement a modular structure for "echo" that enhances maintainability.

**User-Centric Development and Optimization:**

Explore cases where user feedback has influenced compiler development and optimization strategies, ensuring that the "echo" compiler meets practical programming needs and evolves over time.

# Key Bibliography

PDF resources:

* <https://drive.google.com/file/d/1O393wxgxhiZ8BRoccjrrt_2J8r19DDPj/view?usp=drive_link>
* <https://drive.google.com/file/d/1zI9tlKxFsUdZ6jvYp-ifUuAxcoPke-3N/view?usp=sharing>
* <https://drive.google.com/file/d/1MHSW1s5GA438130sn4EsKBL8Q25UxxUN/view?usp=sharing>
* <https://drive.google.com/file/d/1ZiooMAnwLNVKevZhTwKCqKqds1gC5TPy/view?usp=sharing>

Other textual resources:

* The Flex Manual: The official documentation for Flex (Fast Lexical Analyzer Generator) is crucial for understanding how to use Flex to generate lexical analyzers. It covers the syntax of Flex specifications and usage guidelines.
* Bison Documentation: The official documentation for Bison is an essential reference for understanding the capabilities and features of the Bison tool. It provides detailed information on grammar specification and parser generation.
* Lesk, M. E., & Schmidt, E. (1975). Lex: A Lexical Analyzer Generator: The original paper introducing Lex, written by Mike Lesk and Eric Schmidt, provides insights into the design and functionality of Lex. It's a foundational reference for those interested in lexical analysis