Compiler for C Language

### Automata and Compiler Design (IT250) Report

### Submitted in partial fulfillment of the requirements for the degree of

### BACHELOR OF TECHNOLOGY

### In

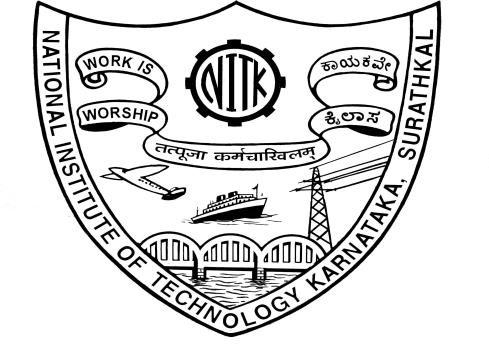
### INFORMATION TECHNOLOGY

### by

### NITHIN S 221IT085

### JAY CHAVAN 221IT020

### AYUSH KUMAR 221IT015



### DEPARTMENT OF INFORMATION TECHNOLOGY NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL, MANGALORE -575025

### March, 2024

**D E C L A R A T I O N**

We hereby *declare* that the *ACD Project Report* entitled **“C Compiler Design”** which is being submitted to the National Institute of Technology Karnataka Surathkal, in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in the Department of Information Technology, is a ***bonafide report of the work carried out by us.*** The material contained in this project report has not been submitted to any University or Institution for the award of any degree.

Nithin S Ayush Kumar Jay Chavan

Signature Signature Signature

Department of IT Department of IT Department of IT

**CERTIFICATE**

This is to certify that the Seminar entitled **“C Compiler Phases”** has been presented by Nithin S (221IT085), Ayush Kumar (221IT015) & Jay Chavan ( 221IT020), students of IV semester B.Tech.(I.T), Department of Information Technology, National Institute of Technology Karnataka, Surathkal, on 25 March, 2024, during the even semester of the academic year 2023 - 2024, in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Examiner-1 Name Examiner-2 Name

Signature of the Examiner-1 with Date Signature of the Examiner-2 with Date

Place :

Date:

**TABLE OF CONTENTS**

**CHAPTER 1 : INTRODUCTION 1**

**CHAPTER 2 : OBJECTIVE 4**

**CHAPTER 3 : METHODOLOGY 4**

**CHAPTER 4 : BLOCK DIAGRAM/FLOW CHART 7**

**CHAPTER 5 : IMPLEMENTATION 10**

**CHAPTER 6 : RESULTS,FUTURE WORKS AND REFERENCES 14**

**LIST OF FIGURES**

[Figure 4.1:](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.gjdgxs) Block Diagram [of Lexical Analyzer](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.1fob9te) 7

FIgure 4.2: Block Diagram [of Syntax Analyzer](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.2et92p0) 8

Figure 4.3[: Block Diagram of Semantic Analyzer](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.tyjcwt) 8

Figure [4.4](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.3dy6vkm) [: Block Diagram o](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.tyjcwt)f Intermediate Code Generator8

Figure [4.5:](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.1t3h5sf)  [Block Diagram of](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.tyjcwt) Code Optimizer9

Figure [4.6:](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.4d34og8)  [Block Diagram of](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.tyjcwt) Target Code Generator [9](https://docs.google.com/document/d/1q5psQkLAqPAQyw8pcpcHVSuNcRxKRIwMA_h4QUW3g-U/edit#heading=h.gjdgxs)

**CHAPTER 1**: **INTRODUCTION**

Our aim is to design a compiler for C Programming Language and present phase by phase output of the Compiler . We are trying to implement the following features

# Modules

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator
5. Code Optimizer
6. Target Code Generator

# 1.1 Lexical Analyzer

* + Identification of Keywords, Identifiers, Operators (Relational, Logical and Arithmetic), Punctuators, Constants (Integer, Character and Float) and String Literals with invalid string error handling
  + Single and Multi Line Comments with error handling
  + Data Types (int, float and char) with modifiers (unsigned and signed) and types (short,long).
  + Procedures with return type int,float and char
  + Parenthesis matching with error reporting.
  + Invalid character identification and error reporting

# 1.2 Syntax Analyzer

In this phase syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers.Checks the expressions from this token are syntactically correct. We are trying to implement the following features.

* + Syntax Checking for arithmetic , logical and relational expressions
  + Productions rules for control statements such as if,if else and nested if statements
  + Production rules for unary plus and minus
  + Production rule and syntax checking for ‘for’ loop and nested for loop
  + Syntax checking for function declaration and parameter passing
  + Array indexing syntax errors
  + Scanf and printf syntax errors
  + Missing semicolon and unbalanced parenthesis

# 1.3 Semantic Analyzer

In this phase, we extract necessary semantic information from the source code which is impossible to detect in parsing. We are trying to implement following features

* + Scope of the identifiers declared
  + Duplicate declaration of identifiers
  + Function declaration and its scope
  + Actual and formal parameter matching(number and type of parameters)
  + Matching function return type
  + Checking array indexing with in the given bound
  + Invalid array indexing while declaration (array limit less than one)
  + Type mismatch of variables

# 1.4 Intermediate Code Generator

In this phase , We are trying to generate language independent three-address code for a given source program which is lexically,syntactically and semantically correct .

It receives input in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, (postfix notation).Intermediate code tends to be machine independent code. Therefore, code generators assume an unlimited number of memory storage (registers) to generate code.This three-address code can be converted to MIPS assembly code .

**1.5 Code Optimizer**

In this phase,the machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means

faster, but other objectives may be desired, such as shorter code, or target code

that consumes less power.

* The optimizer can convert the integer 60 to floating point at compile time, just once.
* Different compilers vary greatly in how much they optimize code.
* Simple optimizations can greatly enhance program running time without much impact on compilation speed.

**1.6 Target Code Generator**

The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces semantically equivalent target programs.

* Code generators must ensure the target program accurately preserves the semantic meaning of the source program.
* The target program must be of high quality, efficiently using the target machine's resources.
* Mathematically, generating an optimal target program from a given source program is an undecidable problem.
* The intermediate representation (IR) may undergo multiple passes during the optimization and code generation phases.

**CHAPTER 2** : **OBJECTIVE**

The objective of designing a compiler for the C Programming Language is to systematically translate C code into an efficient, machine-understandable format through several phases. Initially, the Lexical Analyzer identifies basic language constructs like keywords and operators, handling errors such as invalid characters. The Syntax Analyzer then ensures the grammatical correctness of the code, focusing on expressions, control statements, and syntax errors like missing semicolons. The Semantic Analyzer adds a layer of context, checking for scope, type mismatches, and array bounds to ensure logical consistency. The Intermediate Code Generator converts validated code into a machine-independent format, optimizing for simplicity and efficiency. The Code Optimizer enhances this intermediate code by improving performance and reducing resource consumption without altering the program's semantics. Finally, the Target Code Generator produces the final machine code, optimizing for the target hardware's specifics while ensuring the output remains true to the original program's intent. This multi-phase process aims to create efficient, optimized, and error-free machine code from high-level C language input, balancing speed, resource use, and maintainability.

**CHAPTER 3** : **METHODOLOGY**

**3.1 Lexical Analyzer**

The word “lexical” in the traditional sense means “pertaining to words”. In terms of programming languages, words are objects like variable names, numbers, keywords etc. Such words are traditionally called tokens.

Lexical analysis is the first phase of compiler which is also termed as scanning. A token is a sequence of characters that represent lexical units, which matches with the pattern, such as keywords, operators, identifiers etc. Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces a token as output.

* **Tokens** : It is a valid sequence of characters which are given by lexeme. In a programming language, keywords, constants, identifiers, numbers, operators and punctuation symbols are possible tokens to be identified.
* **Pattern** : A pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules.
* **Lexeme** : A lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a token. Eg: c=a+b\*5.

**3.2 Syntax Analyzer**

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation. The aim of this phase of a compiler is to implement a parser for C language.

The lexical analyzer generated in the first phase reads the source program and generates tokens that are given as input to the parser which then creates a syntax tree in accordance with the grammar, consequently leading to the generation of intermediate code that is fed into the synthesis phase, to obtain the correct, equivalent machine level code. We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis.

Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

**3.3 Semantic Analyzer**

Part of semantic analysis is producing some sort of representation of the program, either object code or an intermediate representation of the program. One - pass compilers will generate object code without using an intermediate representation; code generation is part of the semantic actions performed during parsing. Other compilers will produce an intermediate representation during semantic analysis; most often it will be an abstract syntax tree or quadruples.

Semantic analysis typically involves:

● Type checking - Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)

● Label Checking - Labels references in a program must exist.

● Flow control checks - Control structures must be used in their proper fashion (no GOTOs into a FORTRAN DO statement, no breaks outside a loop or switch statement, etc.)

● Array-bound Checking - Variables being used as an array index should be within the bounds of the array.

**3.4 Intermediate Code Generator**

Compilers generate machine code, whereas interpreters interpret intermediate code. Interpreters are easier to write and can provide better error messages (symbol table is still available). However, they are at least 5 times slower than machine code generated by compilers and also require much more memory than machine code generated by compilers.

While generating machine code directly from source code is possible, it entails two problems :

1) With m languages and n target machines, we need to write m front ends, m × n optimizers, and m × n code generators

2) The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused.

**3.5 Code Optimizer**

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

* The output code must not, in any way, change the meaning of the program.
* Optimization should increase the speed of the program and if possible, the program should demand less resources.
* Optimization should itself be fast and should not delay the overall compiling process.Efforts for an optimized code can be made at various levels of compiling the process.
* After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
* While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

**3.6 Target Code Generator**

Code generation can be considered as the final phase of compilation. Through post code generation, optimization processes can be applied on the code, but that can be seen as a part of the code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

* It should carry the exact meaning of the source code.
* It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

**CHAPTER 4: Block Diagram/Flow Chart**

**4.1 Lexical Analyzer**

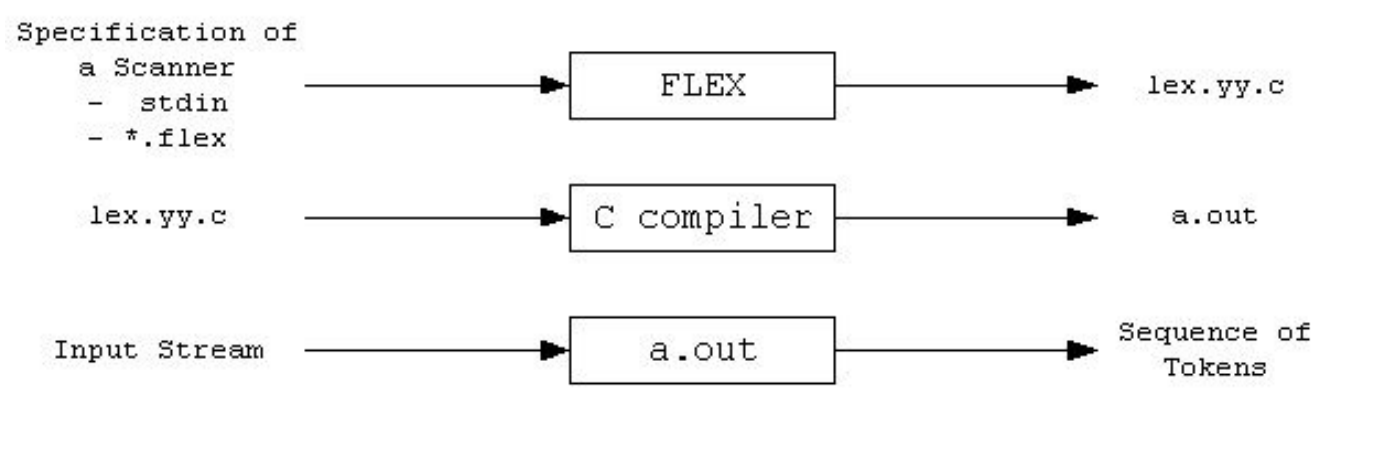


fig. 4.1 Block diagram of Lexical analyzer

**4.2 Syntax Analyzer**

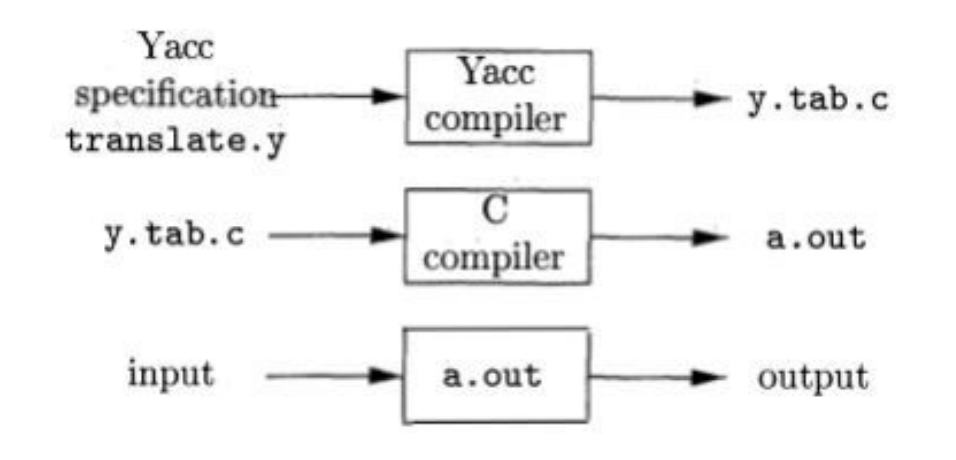


fig. 4.2 Block diagram of syntax analyzer

**4.3 Semantic Analyzer**

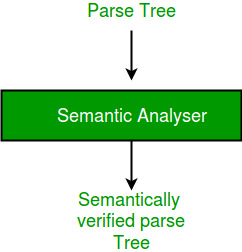


fig. 4.3 Block diagram of semantic analyzer

**4.4 Intermediate Code Generator**

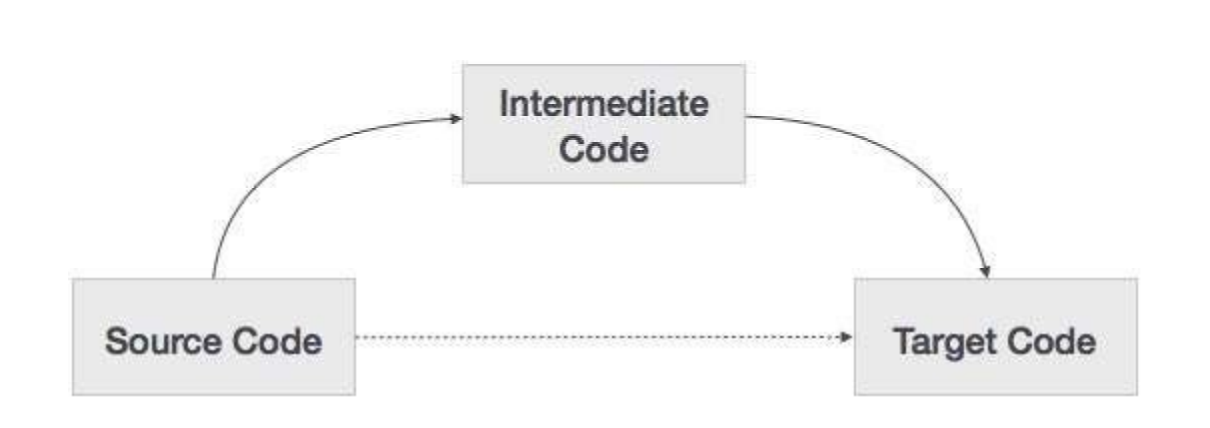


fig. 4.4 Block diagram of Intermediate Code Generator

**4.5 Code Optimizer**

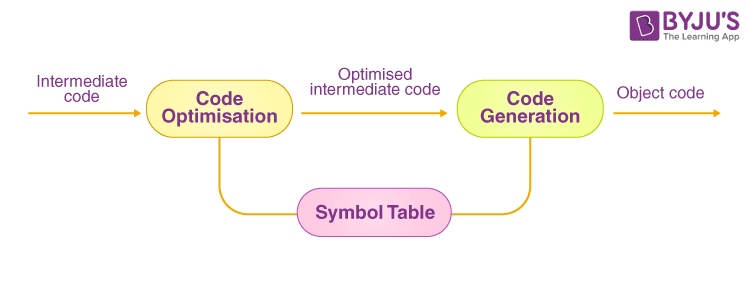
****

fig. 4.5 Block diagram of Code Optimizer

**4.6 Target Code Generator**

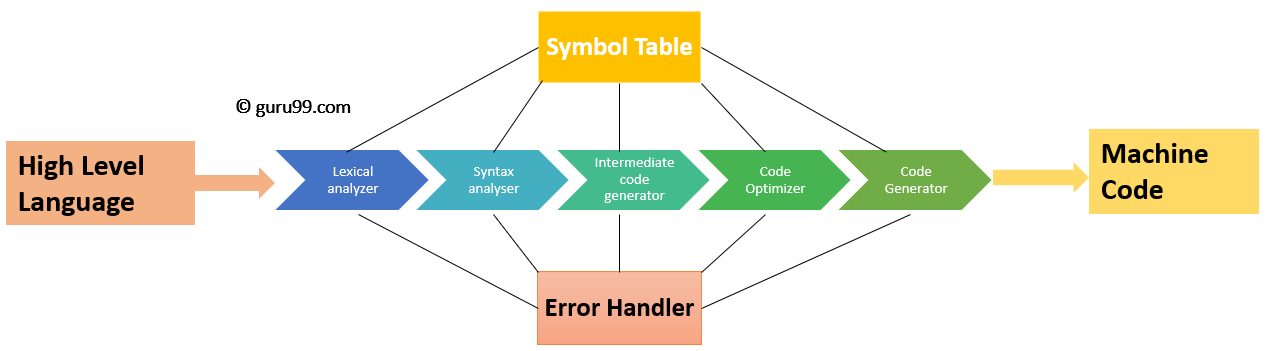
****

fig. 4.6 Block diagram of Target Code Generator

**CHAPTER 5** : **IMPLEMENTATION**

**5.1 Lexical Analyzer**

The scanner performs lexical analysis of a certain program. It reads the source program as a sequence of characters and recognizes "larger" textual units called tokens.

FLEX stands for Fast Lexical Analyzer Generator. It is a tool for generating scanners. Instead of writing a scanner from scratch, you only need to identify the vocabulary of a certain language, write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a scanner for you. FLEX is generally used in the manner depicted here:

First, FLEX reads a specification of a scanner either from an input file \*.lex, or from standard input, and it generates as output a C source file lex.yy.c. Then, lex.yy.c is compiled and linked with the "-lfl" library to produce an executable a.out. Finally, a.out analyzes its input stream and transforms it into a sequence of tokens.

● \*.lex is in the form of pairs of regular expressions and C code.

● lex.yy.c defines a routine yylex() that uses the specification to recognize tokens.

● a.out is actually the scanner.

**5.2 Syntax Analyzer**

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. Every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%'' marks. (The percent ``%'' is generally used in Yacc specifications as an escape character.)In other words, a full specification file looks like :

*declarations*

*%%*

*rules*

*%%*

*programs*

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

*%%*

*rules*

The rules section is made up of one or more grammar rules. A grammar rule has the form:

*A : BODY ;*

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals.

**5.3 Semantic Analyzer**

An attribute is a property whose value is assigned to a grammar symbol. Attribute computation functions (or semantic functions) are associated with the production of a grammar and are used to compute the values of an attribute.

An attribute grammar is an extension to a context - free grammar that is used to describe features of a programming language that cannot be described in BNF or can only be described in BNF with great difficulty. Each attribute has a well-defined domain of values, such as integer, float, character, string, and expressions. Attribute grammar is a medium to provide semantics to context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree

*E → E + T { E.value = E.value + T.value }*

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E. Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories :

● Synthesized attributes - These attributes get values from the attribute values of their child nodes. Synthesized attributes never take values from their parent nodes or any sibling nodes.

For example, S → ABC

● Inherited attributes - In contrast to synthesized attributes, inherited attributes can take values from parents and/or siblings.

For example, S → ABC

**5.4 Intermediate Code Generator**

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generators assume an unlimited number of memory storage (register) to generate code. The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

A statement involving no more than three references(two for operands and one for result) is known as three address statements. A sequence of three address statements is known as three address code. Three address statements are of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statements.

Example – The three address code for the expression a + b \* c + d :

● T 1 = b \* c

● T 2 = a + T 1

● T 3 = T 2 + d

where T 1 , T 2 , T 3 are temporary variables.

**5.5 Code Optimizer**

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

* The output code must not, in any way, change the meaning of the program.
* Optimization should increase the speed of the program and if possible, the program should demand less resources.
* Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

* Efforts for an optimized code can be made at various levels of compiling the process.
* After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
* While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

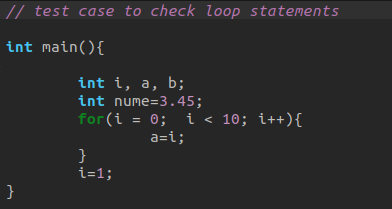
**5.6 Target Code Generator**

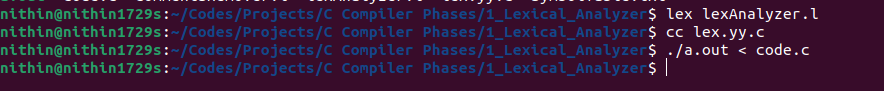
A code generator is expected to have an understanding of the target machine’s runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

* **Target language** : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
* **IR Type** : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
* **Selection of instruction** : The code generator takes Intermediate Representation as input and converts (maps) it into target machine’s instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
* **Register allocation** : A program has a number of values to be maintained during the execution. The target machine’s architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
* **Ordering of instructions** : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

**CHAPTER 6 : Results**

**6.1 Lexical Analyzer**





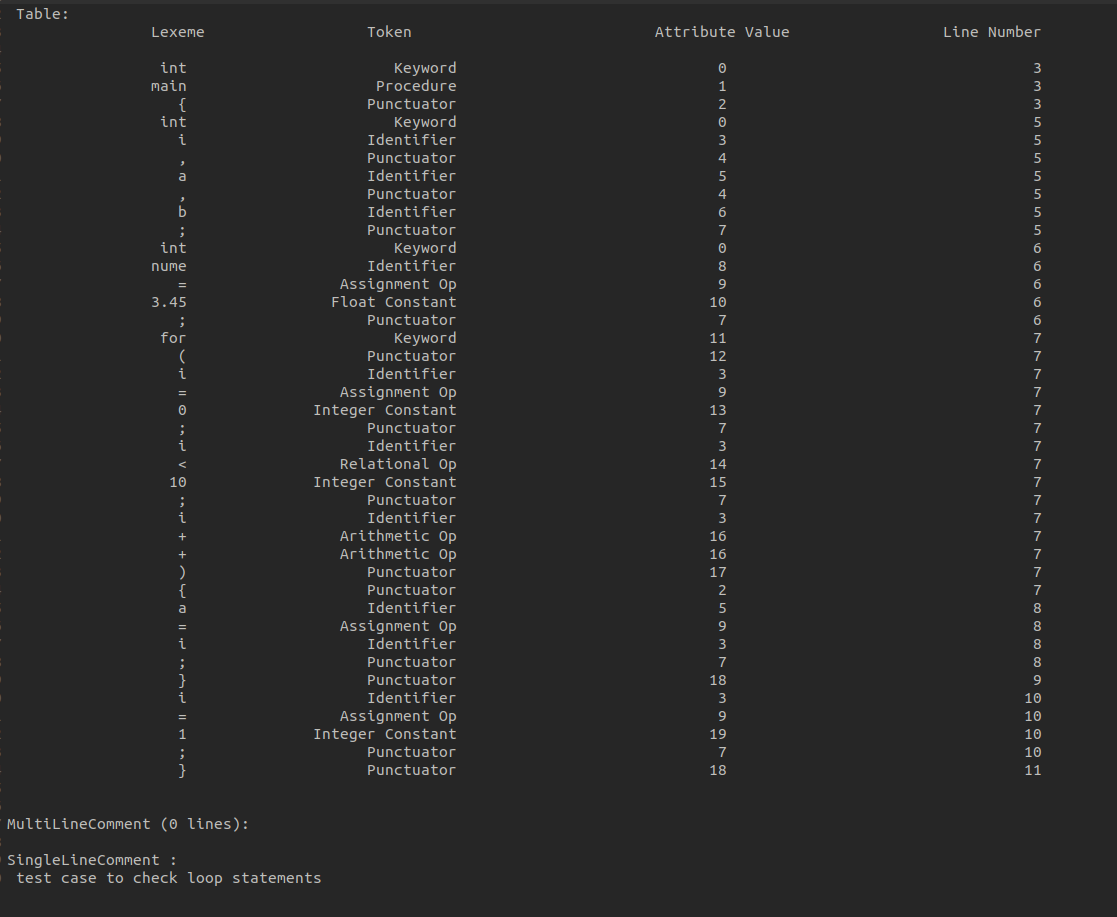
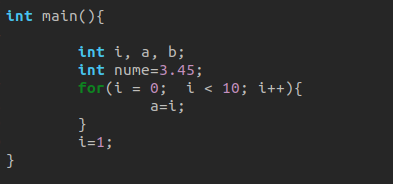


fig. 6.1 Result of lexical analyzer

**6.2 Syntax Analyzer**



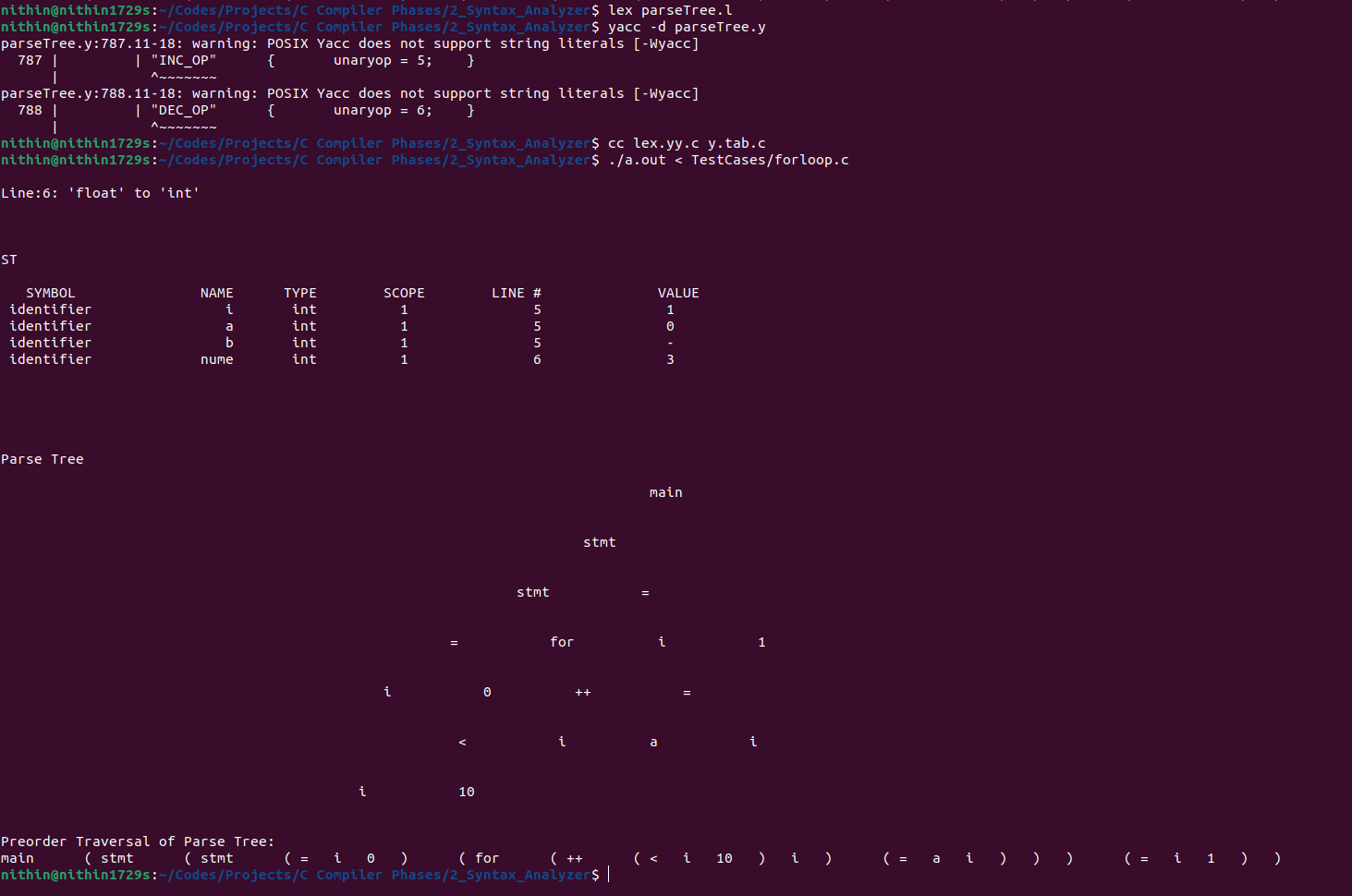
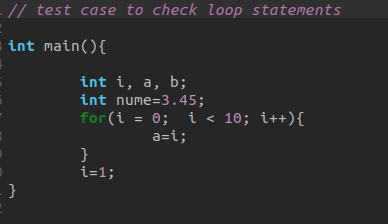
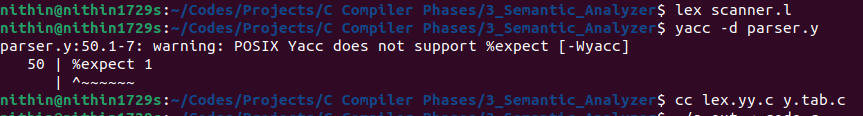


fig. 6.2 Result of syntax analyzer

**6.3 Semantic Analyzer**





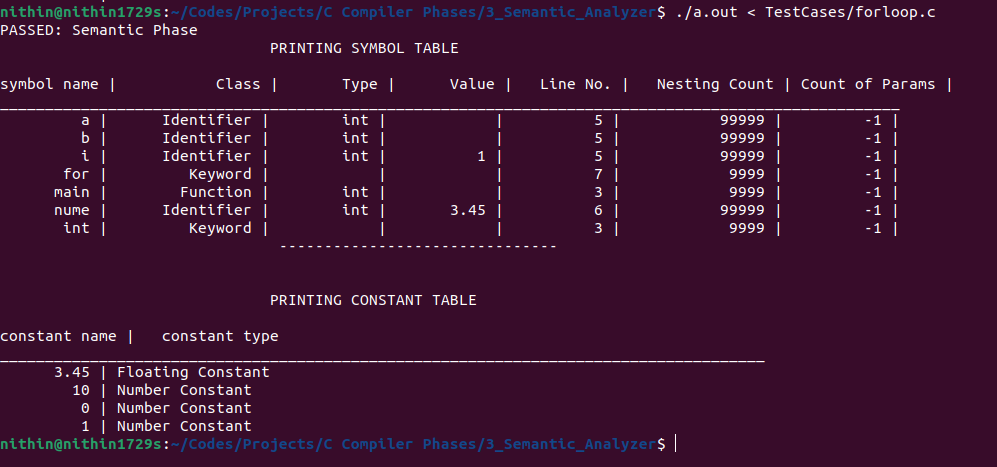
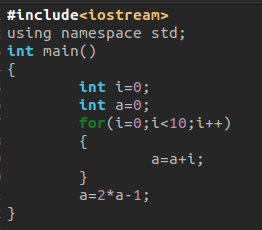
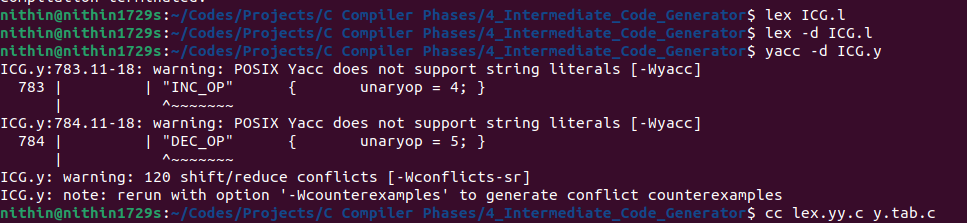


fig. 6.3 Result of semantic analyzer

**6.4 Intermediate Code Generator**



****

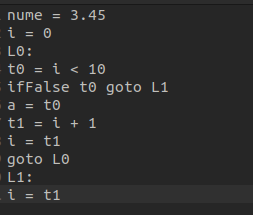
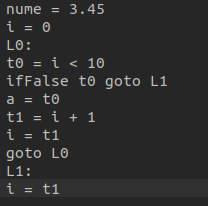


fig. 6.4 Result of intermediate code generator

**6.5 Code Optimization**

ICG Output of previous phase





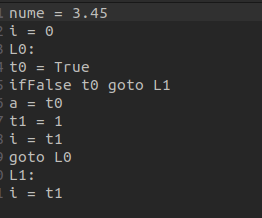
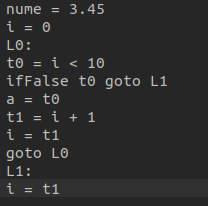
Optimized ICG 

fig 6.5 Results of code optimizer

**6.6 Target Code Generator**



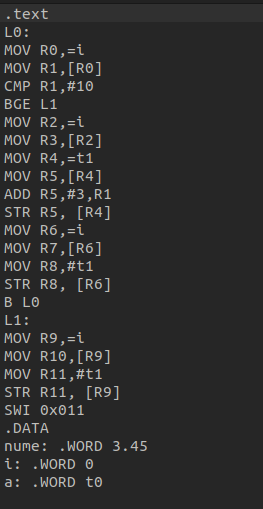
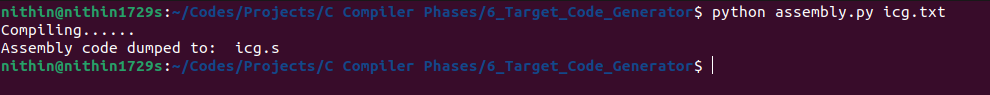


fig 6.6 Result of target code generator

A simple C compiler was successfully built and the output for each phase is shown for sufficient number of test cases..

**GitHub Link:** <https://github.com/Nithin1729S/C-Compiler-Phases>

**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.

**References**

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.

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.

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.

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. ,

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.