THE COMPILER (SYNTAX ANALYSIS)

Submitted by

Viswajith Rajan R [RA2011026010108]
Arvind S[RA2011026010064]

Under the Guidance of

Dr. J. Jeyasudha

Assistant Professor, Department of Computational Intelligence

In partial satisfaction of the requirements for the degree of

BACHELORS OF TECHNOLOGY in COMPUTER SCIENCE ENGINEERING

with specialization in Artificial Intelligence & Machine Learning

INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

SCHOOL OF COMPUTING COLLEGE OF ENGINEERING AND TECHNOLOGY SRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR - 603203

May 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that **18CSC304J – COMPILER DESIGN** project report titled **“THE COMPILER – LEXICAL ANALYSIS**” is the bonafide work of **Viswajith Rajan R [RA2011026010108] and Arvind S [RA2011026010064]** who carried out project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not perform any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE
Faculty In-Charge
Dr. J. Jeyasudha
Assistant Professor
Department of Computational Intelligence
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

SIGNATURE
HEAD OF THE DEPARTMENT
Dr. R Annie Uthra
Professor and Head ,
Department of Computational Intelligence,
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

ABSTRACT

This project is a compiler design project based on the generation of LR(0) items and the construction of SLR Table. The first component of the project focuses on the lexical analysis phase, where the application analyzes the source code and breaks it down into a sequence of tokens. The implementation begins with an overview of the theory behind Shift-Reduce parsing, including the concepts of shift, reduce, and parsing tables. It then delves into the step-by-step process of constructing a parsing table based on a grammar, utilizing techniques such as item sets, closure, and goto functions. The syntax analyzer is the main focus of the project. It takes the grammar as input and generates the required item sets and SLR Table accordingly. Overall, this paper provides a comprehensive overview of the implementation of a Shift-Reduce Parsing Algorithm for natural language processing. The presented implementation offers a valuable tool for and syntactic analysis in various domains.

TABLE OF CONTENTS

1.1 INTRODUCTION

Developing a desktop application that provides developers with the ability to input their code and instantly view the compiler's output is a valuable and efficient tool. As software development grows more complex, the importance of testing code and identifying errors prior to deployment becomes increasingly critical. This application offers developers a convenient way to quickly and effectively test their code, eliminating the need to install and configure an entire development environment.

Imagine a scenario where a developer is working on a new feature that requires implementing a complex algorithm they are not familiar with. After writing the code, they encounter an error when attempting to run it. Without the ability to test the code separately, the developer must spend a significant amount of time isolating and fixing the error.

However, with the website tool we have created, developers can simply input their code, select the relevant part of the compiler to test (e.g., the lexical analyzer), and quickly identify and correct any syntax errors. They can then run the code again and see the compiler output. This saves developers time and effort during the debugging process.

Furthermore, this tool is also beneficial for teaching programming. Beginners can use the website to learn programming basics without the need to set up a development environment. The userfriendly interface allows users to write code, view the compiler's output, and make necessary changes. The immediate and clear feedback provided by the tool helps beginners identify and fix

1.2 PROBLEM STATEMENT

As a software developer, we might have encountered situations where you want to test your code against different compilers, or we might have to compile your code on different platforms. But it can be time-consuming and challenging to set up different compilers and platforms to compile your code manually. This is where the tool you have developed comes in handy. The tool allows developers to input their code and see the output of what the compiler would produce without worrying about installing and configuring different compilers and platforms. With your tool, developers can quickly test their code against different compilers and platforms without leaving their development environment. This tool can also be beneficial for developers who are just starting with programming, as they can see how their code is being compiled and understand the different stages of the compilation process, such as lexical analysis and intermediate code generation. Moreover, the tool can help developers to identify and fix errors in their code during the development phase, making it easier for them to deliver bug-free code. Hence, the tool can save developers a lot of time and effort by providing them with a convenient and efficient way to test their code against different compilers and platforms.

1.3 OBJECTIVES

- To develop an application that allows programmers to input their code and see the output of what the compiler would produce.

- To implement a lexical analyzer, syntax analyzer, intermediate code generation, and quadruple triple in Python.

- To use .NET framework as frontend and C# as the backend.

- When the input is served, the C# script runs based on the input, the result is stored in a buffer and served to the front-end.

- To utilize Visual Studio’s Windows Form UI, to make an intuitive and simple UI for the compiler application.

1.4. HARDWARE REQUIREMENTS

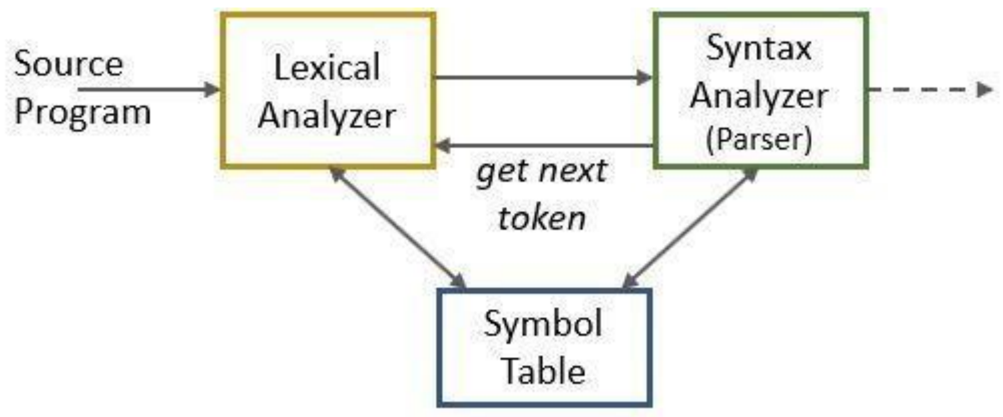
- A server or cloud infrastructure to host the website and the backend logic.
- Sufficient RAM and CPU power to handle multiple user requests
- Sufficient disk space to store the code files and other resources.
- Sufficient cloud storage to store the code files and other resources

1.5 SOFTWARE REQUIREMENTS

- Operating system: Windows or any compatible OS.
- C# interpreter on the backend
- .NET support to use and access the compiler application

CHAPTER 2 ANATOMY OF A COMPILER

2.1 LEXICAL ANALYZER



Lexical Analyzer's Interaction with Parser

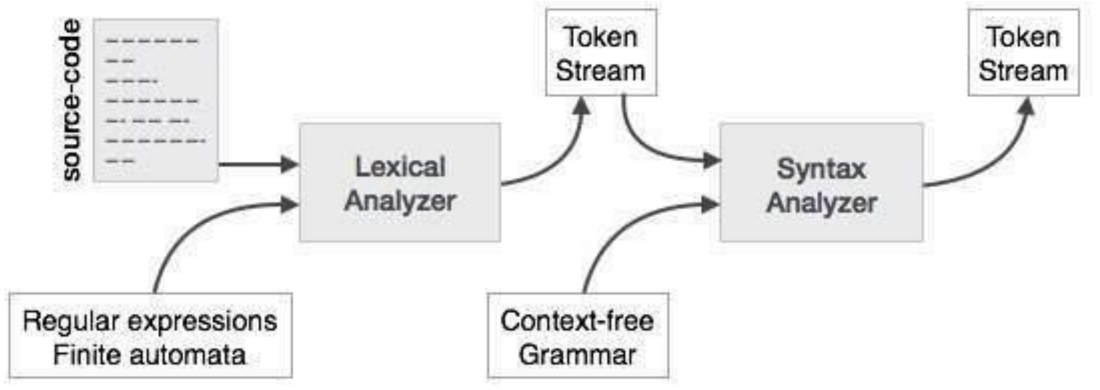
Lexical analysis is the first phase of the compilation process in which the input source code is scanned and analyzed to generate a stream of tokens that are subsequently used by the compiler in the later stages of the compilation process.

The process of lexical analysis is also known as scanning, and the component of the compiler that performs this task is called a lexer or a tokenizer. The primary goal of lexical analysis is to identify the individual lexical units (tokens) of the source code, such as keywords, identifiers, literals, operators, and punctuation marks, and produce a stream of tokens that can be processed by the compiler's parser.

The process of lexical analysis involves several steps. The first step is to read the source code character by character and group them into lexemes, which are the smallest meaningful units of the programming language. Next, the lexer applies a set of rules or regular expressions to identify the lexemes and classify them into different token types.

During this process, the lexer also discards any comments or white spaces that are not significant to the language's syntax. For example, the lexer would ignore any spaces, tabs, or newlines in the source code and focus only on the meaningful tokens.

2.2 SYNTAX ANALYZER

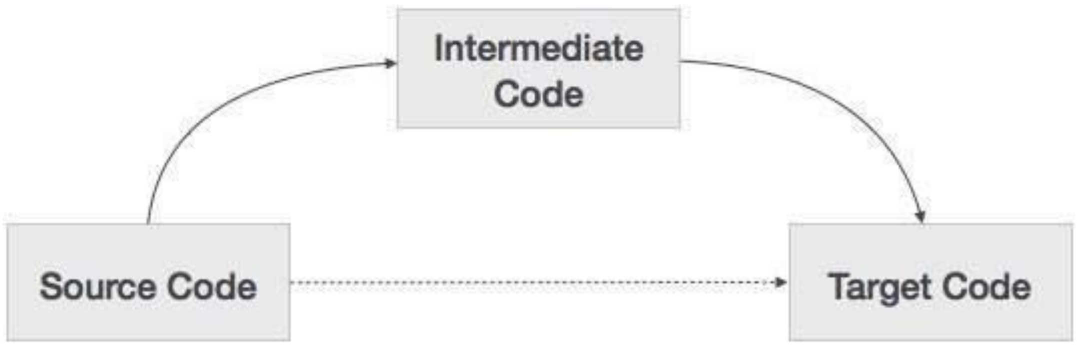


Syntax analysis, also known as parsing, is an essential phase in the compilation process of programming languages. It is responsible for analyzing the structure of the source code and determining if it conforms to the grammar rules of the language. In simpler terms, syntax analysis checks whether the code is written correctly according to the language's syntax rules.

The main objective of syntax analysis is to create a hierarchical structure, such as an Abstract Syntax Tree (AST), from the input source code. This structure represents the relationships and dependencies between different elements of the code, such as statements, expressions, variables, and functions. The AST serves as an intermediate representation of the code and is used for further analysis and code generation phases.

The process of syntax analysis typically involves two important components: a lexer (lexical analyzer) and a parser. The lexer is responsible for tokenizing the source code by breaking it down into a sequence of meaningful units called tokens. These tokens include keywords, identifiers, operators, literals, and punctuation marks. The lexer removes any unnecessary whitespace or comments and outputs a stream of tokens to the parser.

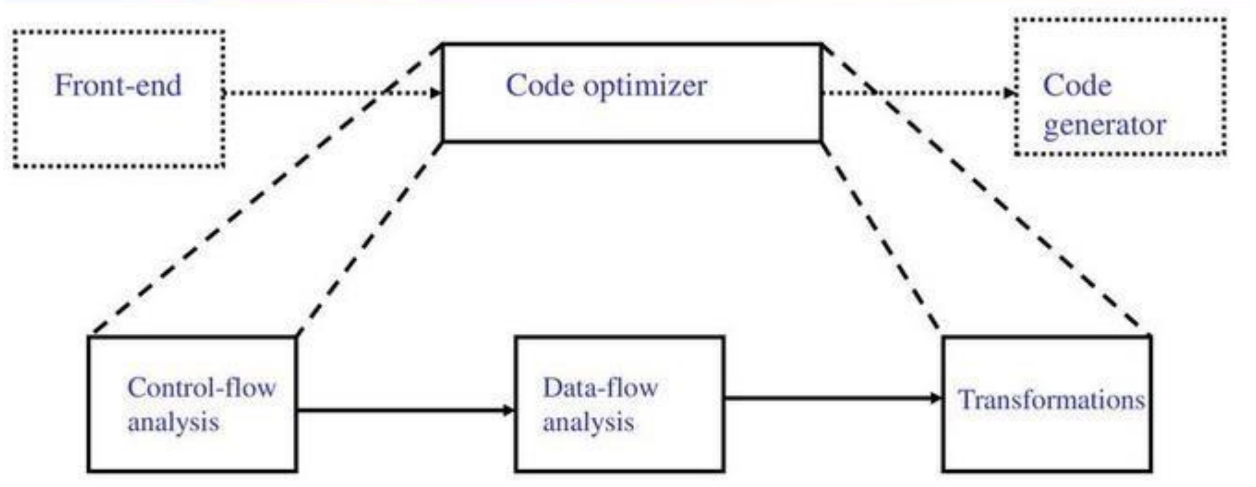
2.3 Intermediate Code Generation



In compiler design, a quadruple is a data structure used to represent an executable instruction or operation in an intermediate representation of a program. It contains four fields, namely the operator or instruction to be performed, the addresses of the operands, and the result. The quadruple is named as such because it has four fields. The operator field specifies the operation or instruction to be performed, such as "add", "subtract", "multiply", "divide", "assign", and so on. The operands and result fields contain memory addresses for the variables or memory locations that contain the values involved in the operation. The quadruple is a simple and uniform representation that can be used during code optimization and code generation phases of the compiler. It can be easily translated into assembly language or machine code.

Quadruples are a common intermediate representation in compilers because they provide a compact and uniform representation of executable instructions. They can be used to represent a wide range of operations, including arithmetic and logical operations, memory accesses.

2.4 Code Optimization



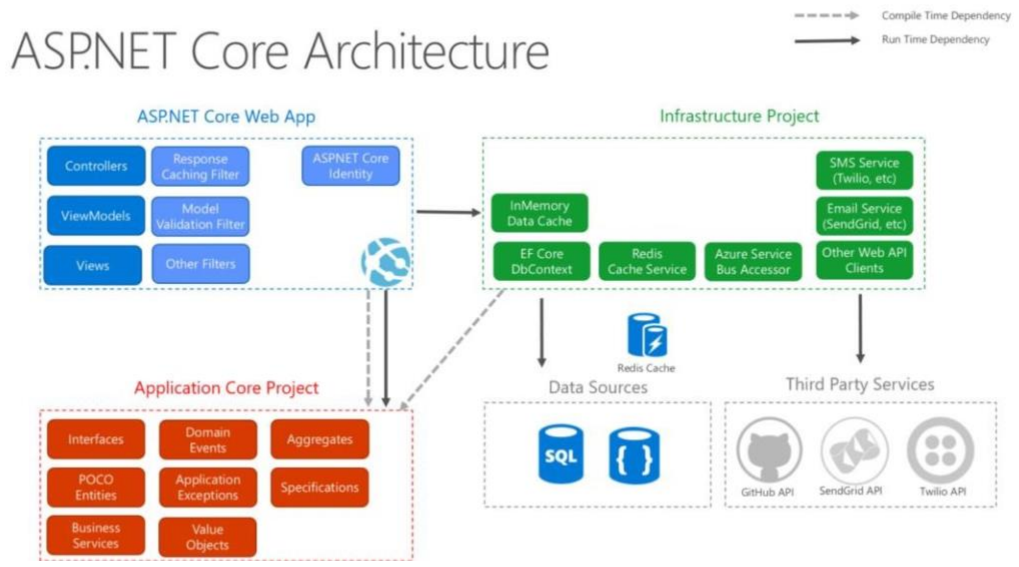
Code optimization is a crucial phase in the compilation process that aims to improve the efficiency, speed, and overall performance of a program while preserving its original functionality. It involves analyzing and transforming the code to reduce its resource consumption, such as memory usage, execution time, or power consumption, without changing its behavior.

Optimizations can be performed at various levels, ranging from high-level language constructs to low-level machine code. The ultimate goal is to produce code that executes faster, uses fewer resources, and produces smaller binaries.

Code optimization is an intricate and complex process that requires a deep understanding of the program, the target architecture, and the trade-offs between code size, execution speed, and resource consumption. Different optimization techniques may interact with each other, and the effectiveness of optimizations can vary depending on the program and the target platform.

CHAPTER 3 ARCHITECTURE AND COMPONENTS

3.1 ARCHITECTURE DIAGRAM



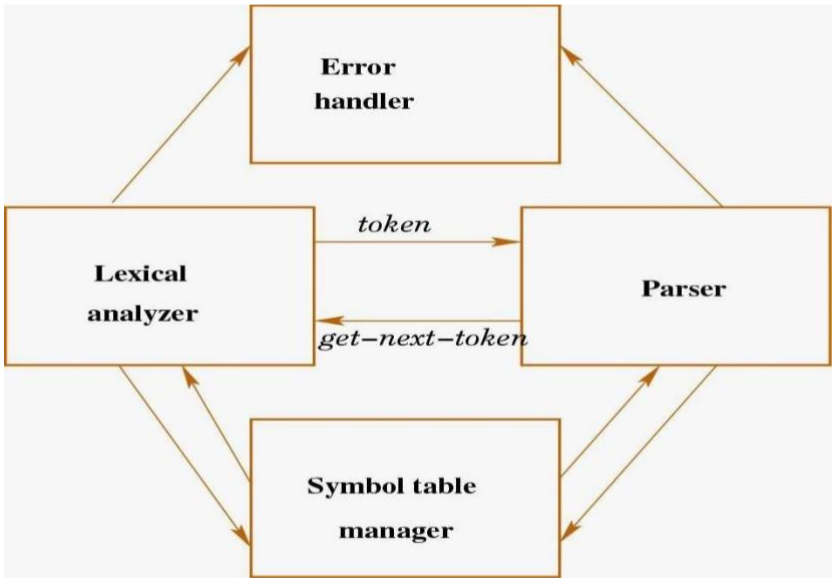
The .NET framework project for the compiler design project follows a layered architecture approach, separating different concerns into distinct layers and components. The architecture consists of the following major components:

1. Presentation Layer:
   - This layer handles the user interface and interaction with the compiler application.
   - It can include a graphical user interface (GUI) or a command-line interface (CLI) for users to input their source code, configure compiler options, and view compilation results.

  - The presentation layer communicates with the underlying layers to coordinate the compilation process and display relevant information to the user.
2. Backend:
  - The syntax analyzer component works on the backend through C# and ensures the syntactic correctness of the source code by analyzing its structure based on a defined grammar.
  - It employs parsing algorithms, such as SLR parsing, to construct parse trees or abstract syntax trees (AST).

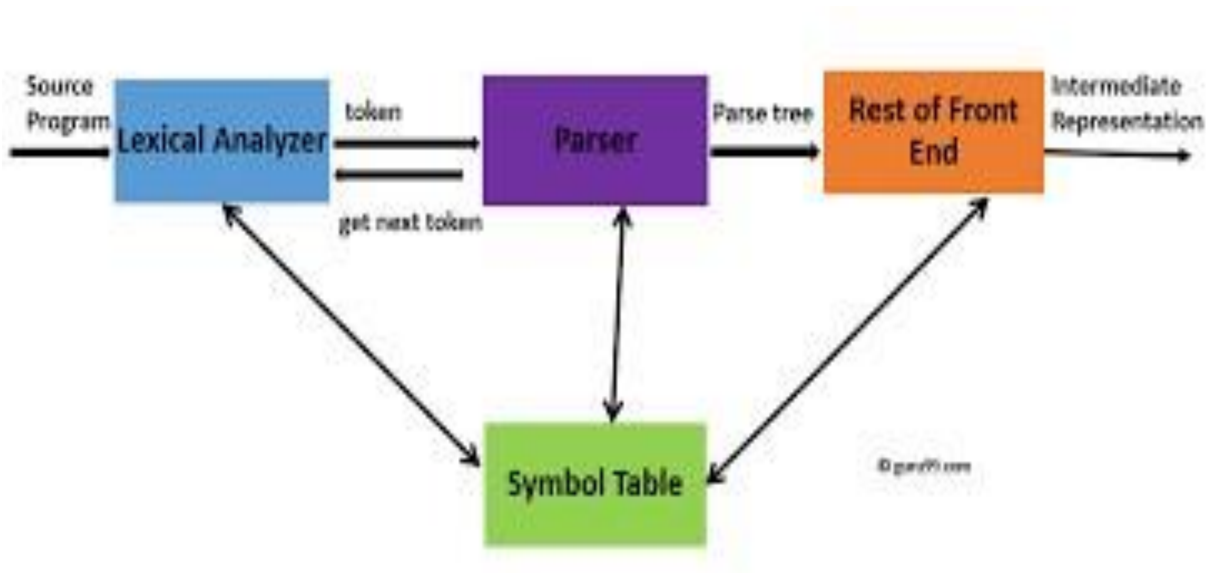
3.2 COMPONENTS DIAGRAMS

3.2.1 LEXICAL ANALYSIS



The component diagram of lexical analysis includes four key components: the lexical analyser, error handler, symbol table, and parser. The lexical analyser breaks down the input source code into a sequence of tokens based on predefined rules and passes them to the parser. The error handler detects and reports any errors that occur during the lexical analysis phase, and communicates with both the lexer and the parser to resolve errors. The symbol table maintains a record of all symbols declared in the source code, and is used by both the lexer and parser. The parser analyses the sequence of tokens generated by the lexer and constructs an abstract syntax tree using a formal grammar. Together, these components work to transform the source code into a structured representation that can be further processed by the compiler

3.2.2 SYNTAX ANALYSIS

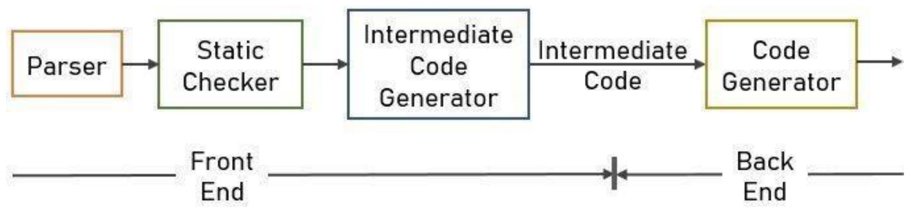


The syntax analysis component is responsible for verifying the syntactic correctness of the source code by analyzing its structure based on a defined grammar. It ensures that the sequence of tokens generated by the lexical analyzer adheres to the grammar rules.

The core component of the syntax analysis is the Syntax Analyzer itself. It processes the token stream and constructs the parse tree or abstract syntax tree (AST) based on the grammar. The syntax analyzer employs parsing algorithms such as recursive descent parsing, LALR(1) parsing, or other techniques to handle different types of grammars.

The Parsing Table is a crucial data structure used by the syntax analyzer. It contains the grammar rules and associated parsing actions. The parsing table is constructed during the compiler's development phase based on the grammar. It provides the necessary information for the syntax analyzer to determine the next action based on the current state and input token.

3.2.3 INTERMEDIATE CODE GENERATION

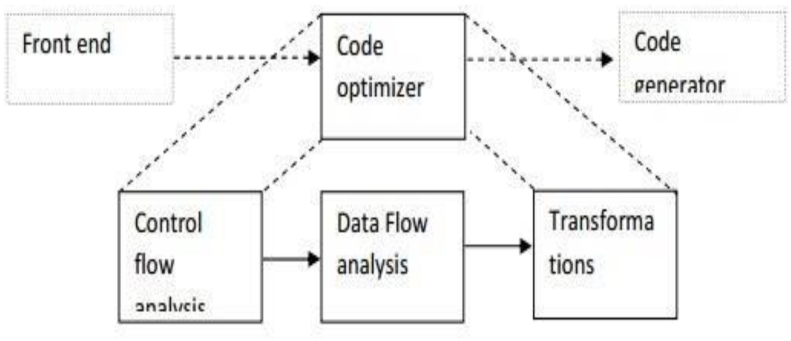


The intermediate code generation component is responsible for translating the parse tree or abstract syntax tree (AST) generated by the syntax analyzer into an intermediate representation of the code. It bridges the gap between the high-level source code and the final target code generation.

The core component of the intermediate code generation is the Intermediate Code Generator. It takes the parse tree or AST as input and traverses it, extracting relevant information about the program's structure and operations. The intermediate code generator applies a set of rules or transformations to convert this information into an intermediate code representation.

The Parse Tree or AST represents the structural hierarchy and relationships between the different elements of the source code. It serves as the input to the intermediate code generator, providing the necessary information about variables, functions, control flow, and other language-specific constructs.

3.2.4 CODE OPTIMIZATION



The code optimization component is responsible for improving the efficiency and performance of the intermediate code generated by the intermediate code generator. It applies various optimization techniques to transform the intermediate code while preserving its semantics.

The core component of the code optimization is the Code Optimizer. It receives the intermediate code as input and performs a series of analysis and transformations to enhance the code's efficiency. The code optimizer applies optimization techniques tailored to the specific characteristics of the intermediate code representation.

The Intermediate Code represents the input to the code optimization component. It is the abstract representation of the program logic generated by the intermediate code generator.

The intermediate code provides a suitable representation that allows the code optimizer to analyze and modify the code to achieve optimization goals.

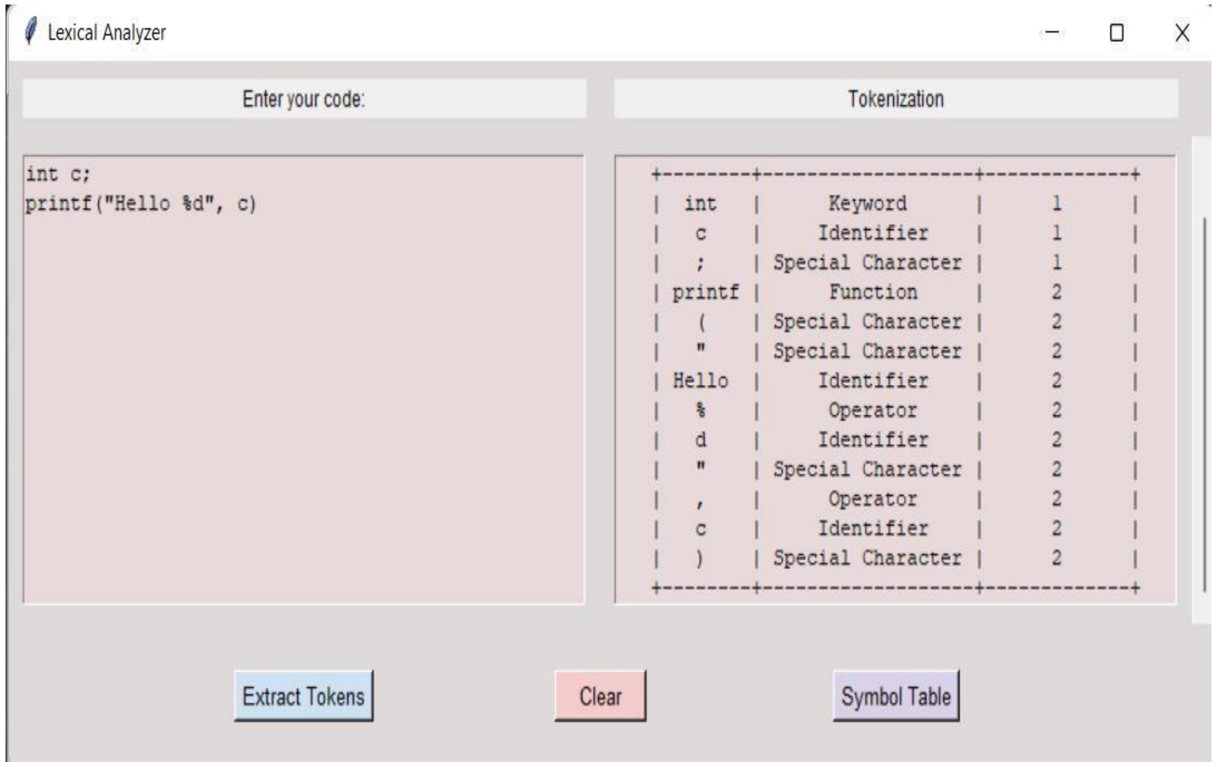
During the code optimization process, the code optimizer performs various optimization techniques such as constant folding, common subexpression elimination, dead code.

CHAPTER 4

CODING AND TESTING

4.1 LEXICAL ANALYSIS

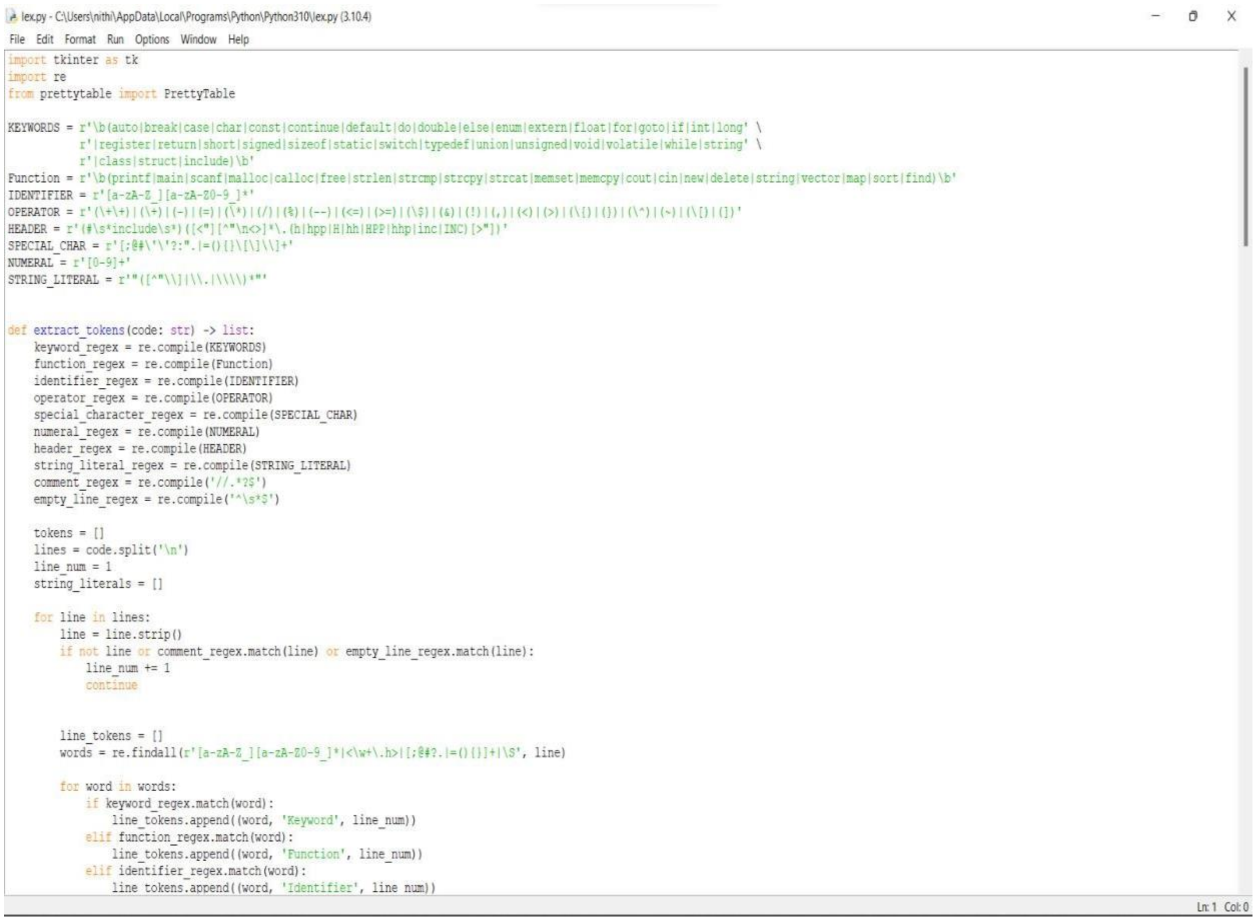
4.1.1 FRONTEND



```
int c;
printf("Hello %d", c)
```

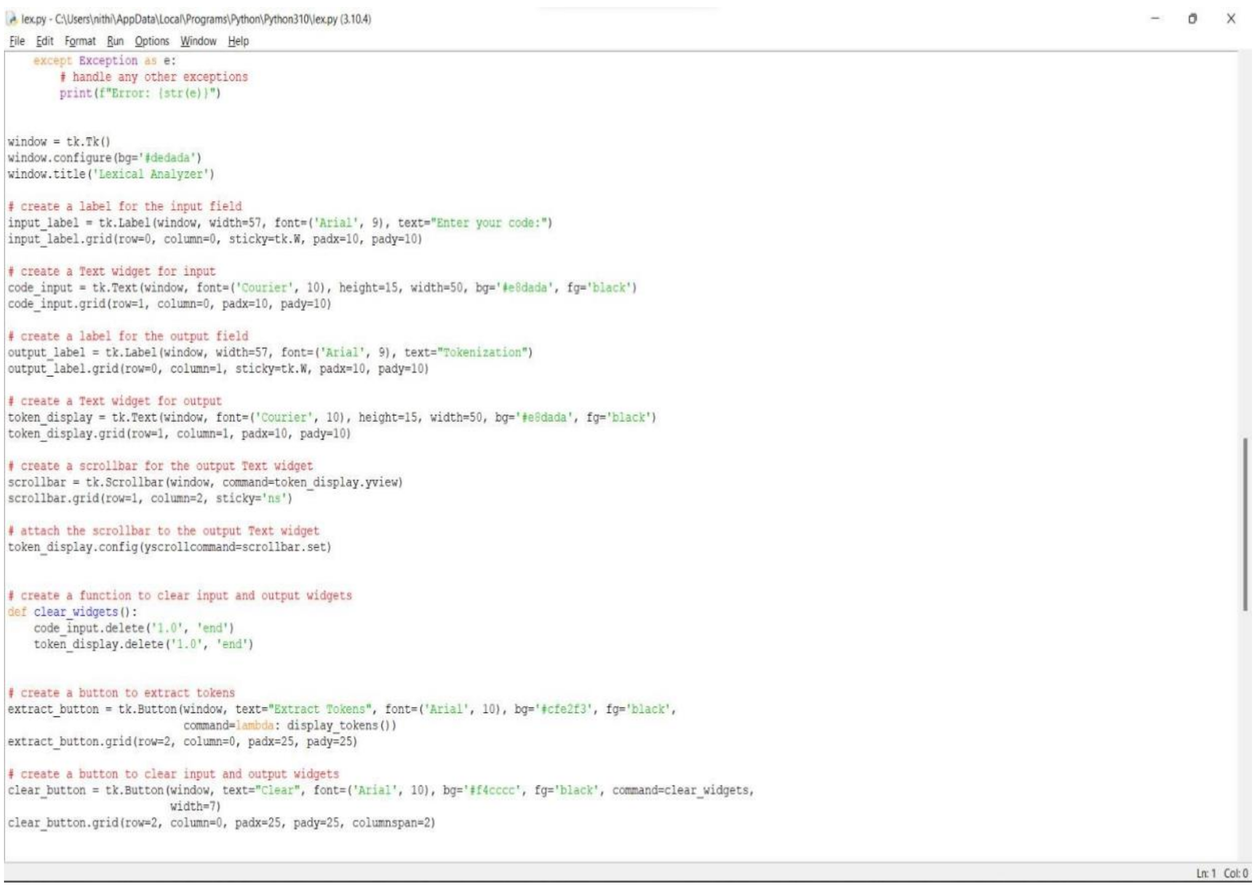
```
+--------+------------------+-------------+
|  int   |     Keyword      |      1      |
|   c    |    Identifier    |      1      |
|   ;    | Special Character |     1      |
| printf |     Function     |      2      |
|   (    | Special Character |     2      |
|   "    | Special Character |     2      |
| Hello  |    Identifier    |      2      |
|   %    |     Operator     |      2      |
|   d    |    Identifier    |      2      |
|   "    | Special Character |     2      |
|   ,    |     Operator     |      2      |
|   c    |    Identifier    |      2      |
|   )    | Special Character |     2      |
+--------+------------------+-------------+
```

4.1.2 BACKEND



lex.py - C:\Users\nithi\AppData\Local\Programs\Python\Python310\lex.py (3.10.4)
File Edit Format Run Options Window Help
Ln: 1 Col: 0

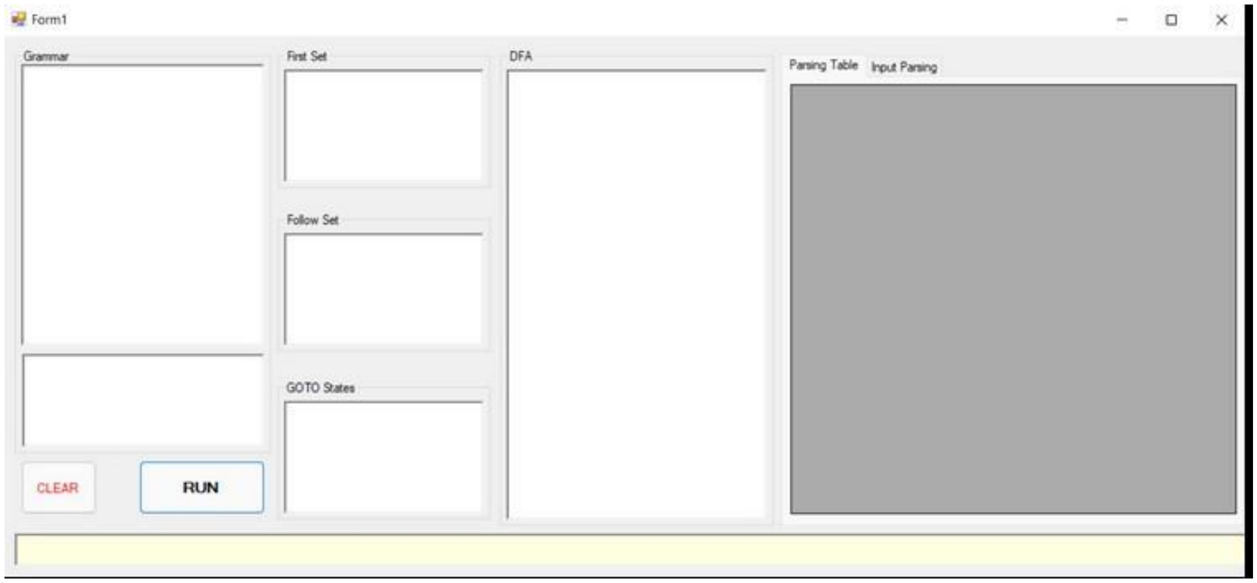
```python
import tkinter as tk
import re
from prettytable import PrettyTable

KEYWORDS = r'\b(auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long' \
           r'|register|return|short|signed|sizeof|static|switch|typedef|union|unsigned|void|volatile|while|string' \
           r'|class|struct|include)\b'
Function = r'\b(printf|main|scanf|malloc|calloc|free|strlen|strcmp|strcpy|strcat|memset|memcpy|cout|cin|new|delete|string|vector|map|sort|find)\b'
IDENTIFIER = r'[a-zA-Z_][a-zA-Z0-9_]*'
OPERATOR = r'(\+\+)|(\+)|(-)|(=)|(\*)|(/)|(%)|(--)|(<=)|(>=)|(\$)|(&)|(!)|(,)|(<)|(>)|(\{)|(})|(\^)|(~)|(\[)|(])'
HEADER = r'(#\s*include\s*)([<"][^"\n<>]*\.(h|hpp|H|hh|HPP|hhp|inc|INC)[>"])'
SPECIAL_CHAR = r'[;@#\'\'?:".|=(){}\[\]\\]+'
NUMERAL = r'[0-9]+'
STRING_LITERAL = r'"([^"\\]|\\.|\\\\)*"'


def extract_tokens(code: str) -> list:
    keyword_regex = re.compile(KEYWORDS)
    function_regex = re.compile(Function)
    identifier_regex = re.compile(IDENTIFIER)
    operator_regex = re.compile(OPERATOR)
    special_character_regex = re.compile(SPECIAL_CHAR)
    numeral_regex = re.compile(NUMERAL)
    header_regex = re.compile(HEADER)
    string_literal_regex = re.compile(STRING_LITERAL)
    comment_regex = re.compile('//.*?$')
    empty_line_regex = re.compile('^\s*$')

    tokens = []
    lines = code.split('\n')
    line_num = 1
    string_literals = []

    for line in lines:
        line = line.strip()
        if not line or comment_regex.match(line) or empty_line_regex.match(line):
            line_num += 1
            continue


        line_tokens = []
        words = re.findall(r'[a-zA-Z_][a-zA-Z0-9_]*|<\w+\.h>|[;@#?.|=(){}]+|\S', line)

        for word in words:
            if keyword_regex.match(word):
                line_tokens.append((word, 'Keyword', line_num))
            elif function_regex.match(word):
                line_tokens.append((word, 'Function', line_num))
            elif identifier_regex.match(word):
                line_tokens.append((word, 'Identifier', line_num))
```

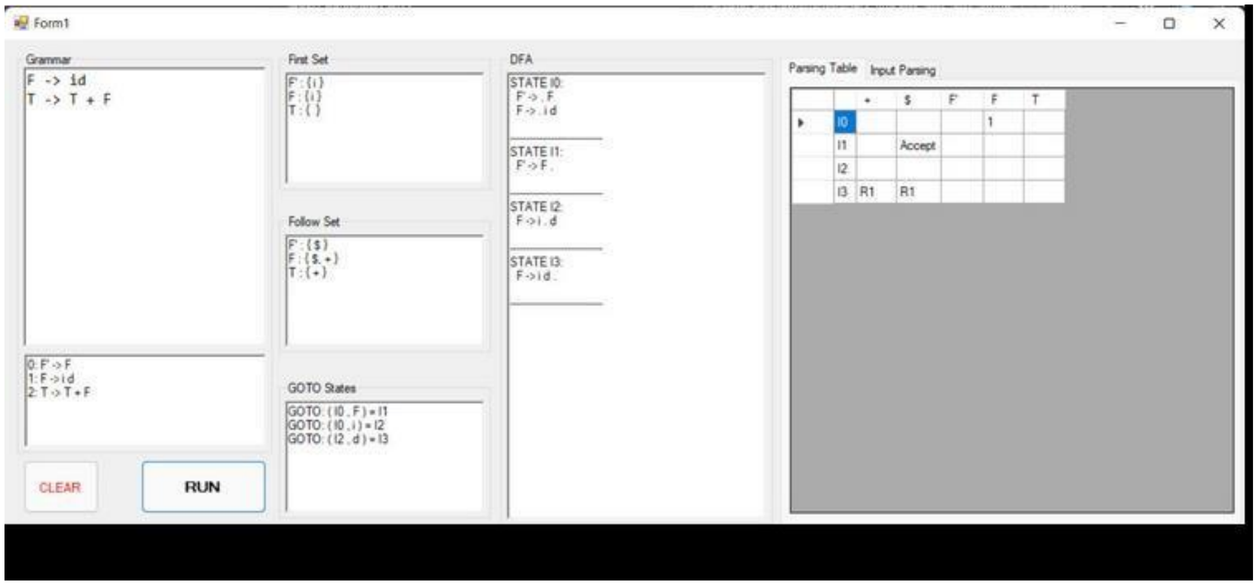

lex.py - C:\Users\nithi\AppData\Local\Programs\Python\Python310\lex.py (3.10.4)
File Edit Format Run Options Window Help
Ln: 1 Col: 0

```python
    except Exception as e:
        # handle any other exceptions
        print(f"Error: {str(e)}")


window = tk.Tk()
window.configure(bg='#dedada')
window.title('Lexical Analyzer')

# create a label for the input field
input_label = tk.Label(window, width=57, font=('Arial', 9), text="Enter your code:")
input_label.grid(row=0, column=0, sticky=tk.W, padx=10, pady=10)

# create a Text widget for input
code_input = tk.Text(window, font=('Courier', 10), height=15, width=50, bg='#e8dada', fg='black')
code_input.grid(row=1, column=0, padx=10, pady=10)

# create a label for the output field
output_label = tk.Label(window, width=57, font=('Arial', 9), text="Tokenization")
output_label.grid(row=0, column=1, sticky=tk.W, padx=10, pady=10)

# create a Text widget for output
token_display = tk.Text(window, font=('Courier', 10), height=15, width=50, bg='#e8dada', fg='black')
token_display.grid(row=1, column=1, padx=10, pady=10)

# create a scrollbar for the output Text widget
scrollbar = tk.Scrollbar(window, command=token_display.yview)
scrollbar.grid(row=1, column=2, sticky='ns')

# attach the scrollbar to the output Text widget
token_display.config(yscrollcommand=scrollbar.set)


# create a function to clear input and output widgets
def clear_widgets():
    code_input.delete('1.0', 'end')
    token_display.delete('1.0', 'end')


# create a button to extract tokens
extract_button = tk.Button(window, text="Extract Tokens", font=('Arial', 10), bg='#cfe2f3', fg='black',
                           command=lambda: display_tokens())
extract_button.grid(row=2, column=0, padx=25, pady=25)

# create a button to clear input and output widgets
clear_button = tk.Button(window, text="Clear", font=('Arial', 10), bg='#f4cccc', fg='black', command=clear_widgets,
                         width=7)
clear_button.grid(row=2, column=0, padx=25, pady=25, columnspan=2)
```

4.2 SYNTAX ANALYSIS

4.2.1 FRONTEND





```
F -> id
T -> T + F
```

```
0: F' -> F
1: F -> id
2: T -> T + F
```

```
F' : { i }
F : { i }
T : { }
```

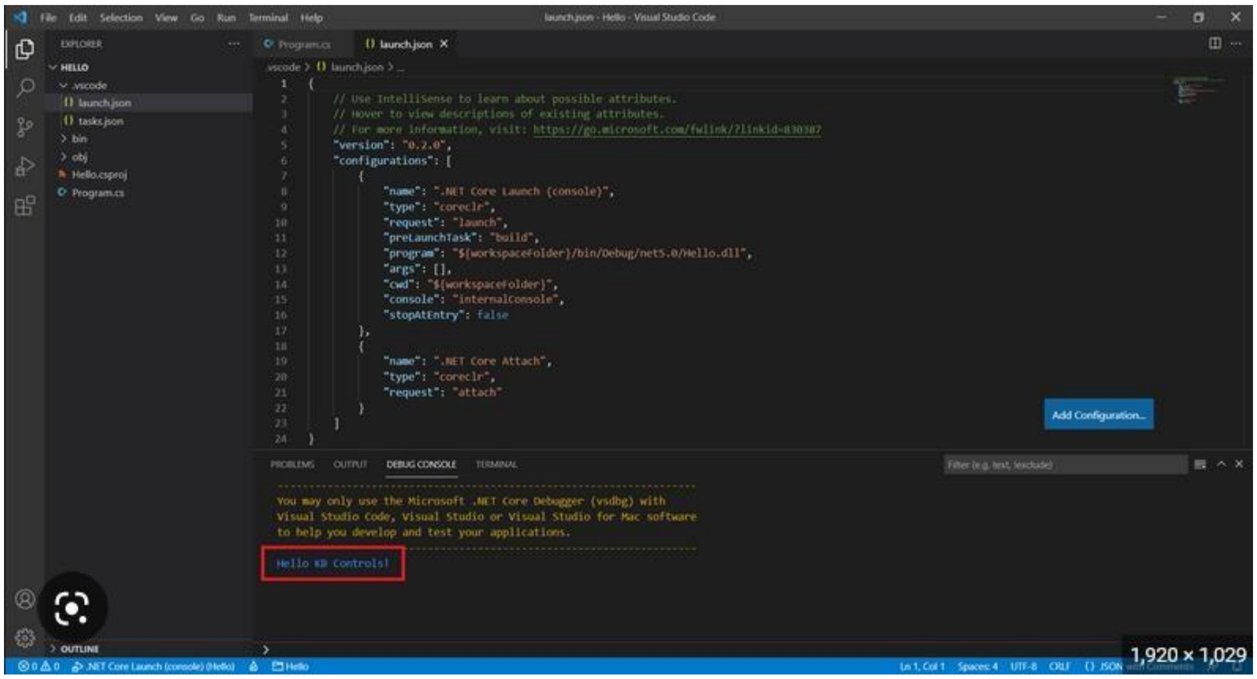
```
F' : { $ }
F : { $, + }
T : { + }
```

```
GOTO: ( I0 , F ) = I1
GOTO: ( I0 , i ) = I2
GOTO: ( I2 , d ) = I3
```

```
STATE I0:
 F' -> . F
 F -> . id

STATE I1:
 F' -> F .

STATE I2:
 F -> i . d

STATE I3:
 F -> id .
```

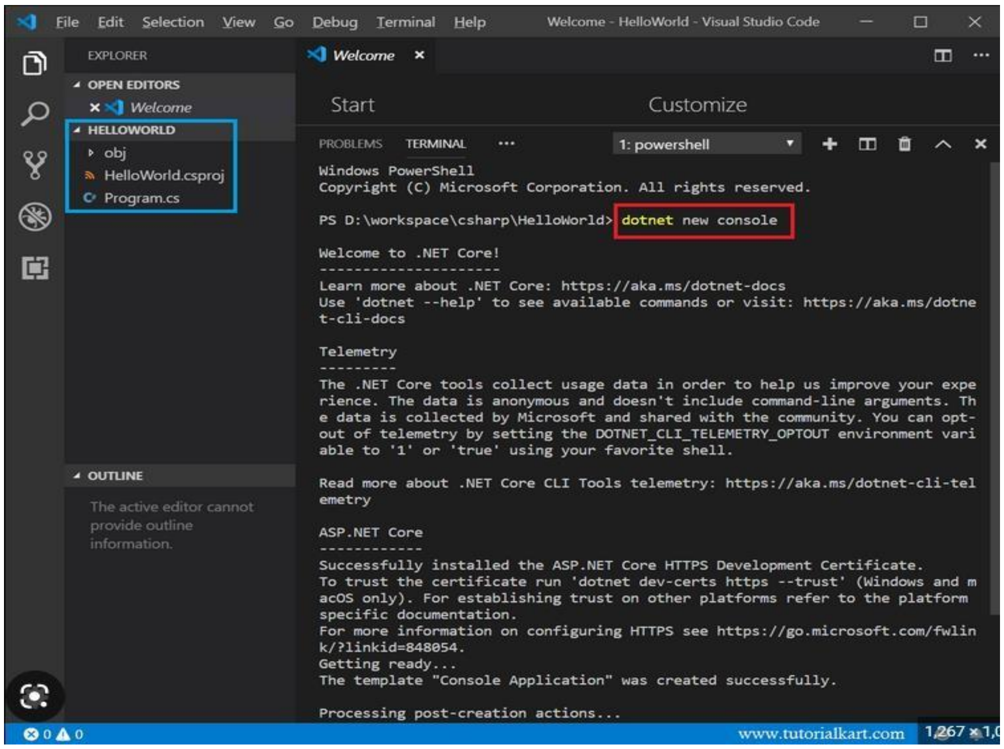
	+	$	F'	F	T
I0				1	
I1		Accept			
I2					
I3	R1	R1			

4.2.2 BACKEND



```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (console)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program": "${workspaceFolder}/bin/Debug/net5.0/Hello.dll",
            "args": [],
            "cwd": "${workspaceFolder}",
            "console": "internalConsole",
            "stopAtEntry": false
        },
        {
            "name": ".NET Core Attach",
            "type": "coreclr",
            "request": "attach"
        }
    ]
}
```

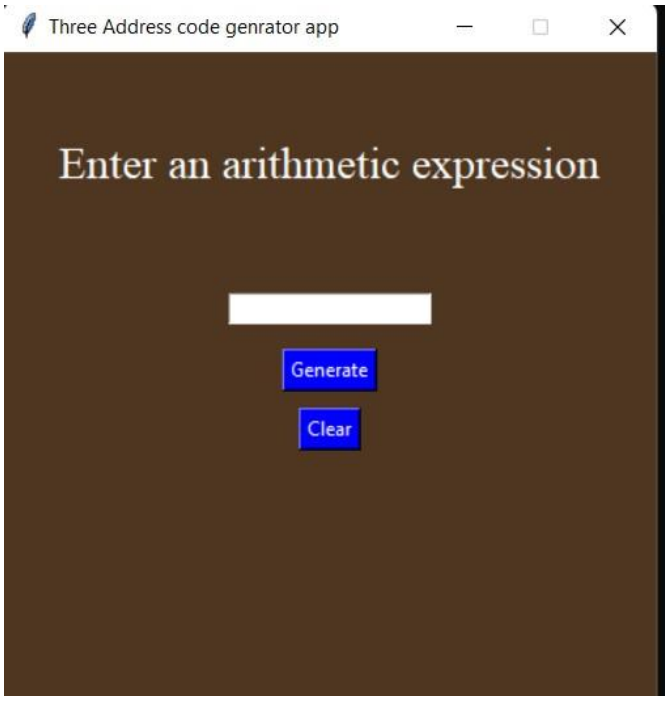
```
--------------------------------------------------------------------
You may only use the Microsoft .NET Core Debugger (vsdbg) with
Visual Studio Code, Visual Studio or Visual Studio for Mac software
to help you develop and test your applications.
--------------------------------------------------------------------
Hello KB Controls!
```

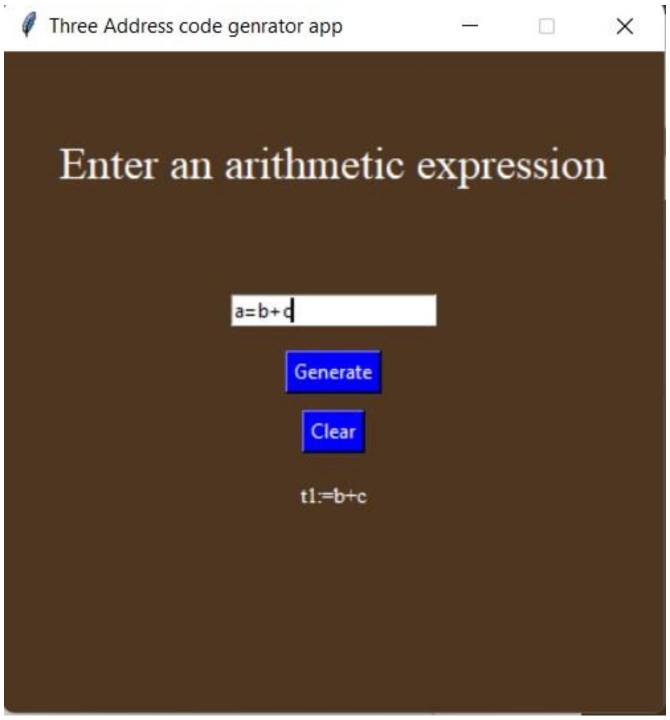


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS D:\workspace\csharp\HelloWorld> dotnet new console

Welcome to .NET Core!
---------------------
Learn more about .NET Core: https://aka.ms/dotnet-docs
Use 'dotnet --help' to see available commands or visit: https://aka.ms/dotne
t-cli-docs

Telemetry
---------
The .NET Core tools collect usage data in order to help us improve your expe
rience. The data is anonymous and doesn't include command-line arguments. Th
e data is collected by Microsoft and shared with the community. You can opt-
out of telemetry by setting the DOTNET_CLI_TELEMETRY_OPTOUT environment vari
able to '1' or 'true' using your favorite shell.

Read more about .NET Core CLI Tools telemetry: https://aka.ms/dotnet-cli-tel
emetry

ASP.NET Core
------------
Successfully installed the ASP.NET Core HTTPS Development Certificate.
To trust the certificate run 'dotnet dev-certs https --trust' (Windows and m
acOS only). For establishing trust on other platforms refer to the platform
specific documentation.
For more information on configuring HTTPS see https://go.microsoft.com/fwlin
k/?linkid=848054.
Getting ready...
The template "Console Application" was created successfully.

Processing post-creation actions...
```

4.3 INTERMEDIATE CODE GENERATION

4.3.1 FRONTEND

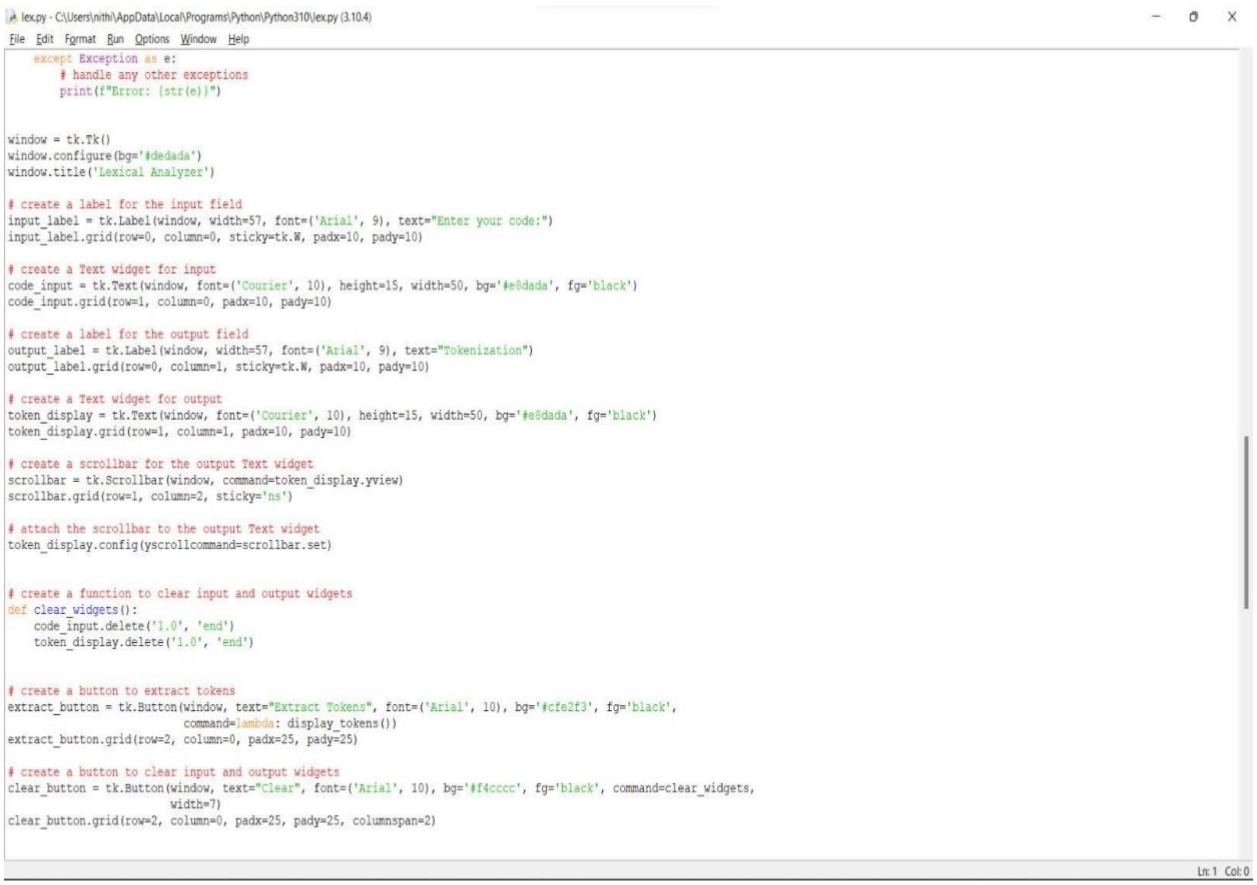

Three Address code genrator app
Enter an arithmetic expression
Generate
Clear


Three Address code genrator app
Enter an arithmetic expression
Generate
Clear

```
a=b+c
```

```
t1:=b+c
```

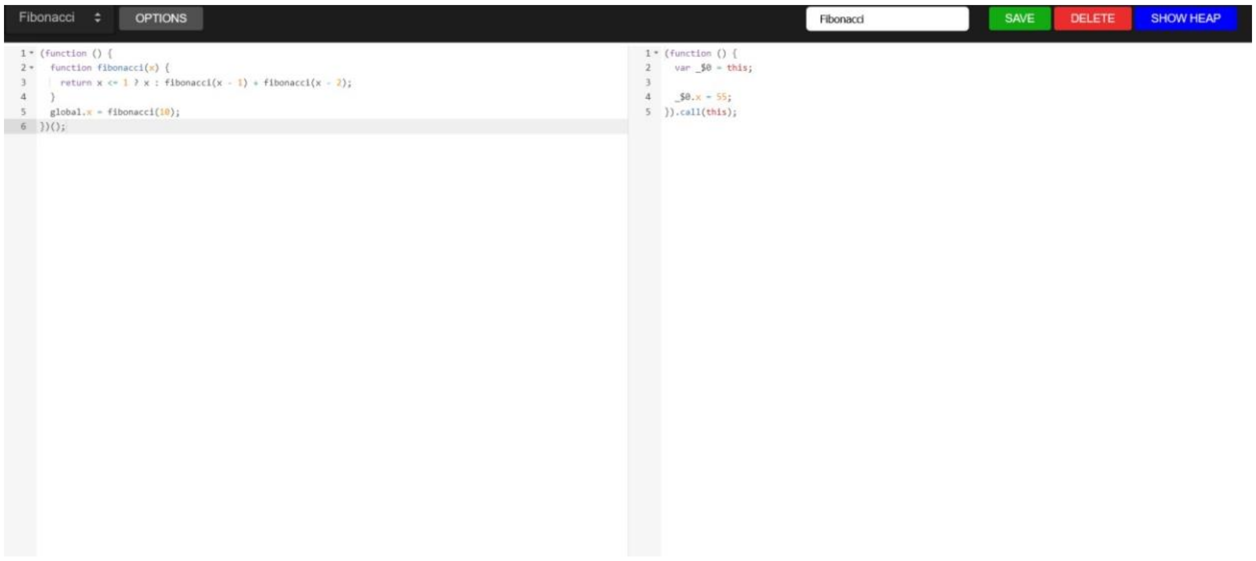
4.3.2 BACKEND


lex.py - C:\Users\nithi\AppData\Local\Programs\Python\Python310\lex.py (3.10.4)
File Edit Format Run Options Window Help
Ln: 1 Col: 0

```
    except Exception as e:
        # handle any other exceptions
        print(f"Error: {str(e)}")


window = tk.Tk()
window.configure(bg='#dedada')
window.title('Lexical Analyzer')

# create a label for the input field
input_label = tk.Label(window, width=57, font=('Arial', 9), text="Enter your code:")
input_label.grid(row=0, column=0, sticky=tk.W, padx=10, pady=10)

# create a Text widget for input
code_input = tk.Text(window, font=('Courier', 10), height=15, width=50, bg='#e8dada', fg='black')
code_input.grid(row=1, column=0, padx=10, pady=10)

# create a label for the output field
output_label = tk.Label(window, width=57, font=('Arial', 9), text="Tokenization")
output_label.grid(row=0, column=1, sticky=tk.W, padx=10, pady=10)

# create a Text widget for output
token_display = tk.Text(window, font=('Courier', 10), height=15, width=50, bg='#e8dada', fg='black')
token_display.grid(row=1, column=1, padx=10, pady=10)

# create a scrollbar for the output Text widget
scrollbar = tk.Scrollbar(window, command=token_display.yview)
scrollbar.grid(row=1, column=2, sticky='ns')

# attach the scrollbar to the output Text widget
token_display.config(yscrollcommand=scrollbar.set)


# create a function to clear input and output widgets
def clear_widgets():
    code_input.delete('1.0', 'end')
    token_display.delete('1.0', 'end')


# create a button to extract tokens
extract_button = tk.Button(window, text="Extract Tokens", font=('Arial', 10), bg='#cfe2f3', fg='black',
                           command=lambda: display_tokens())
extract_button.grid(row=2, column=0, padx=25, pady=25)

# create a button to clear input and output widgets
clear_button = tk.Button(window, text="Clear", font=('Arial', 10), bg='#f4cccc', fg='black', command=clear_widgets,
                         width=7)
clear_button.grid(row=2, column=0, padx=25, pady=25, columnspan=2)
```

4.4 CODE OPTIMIZATION

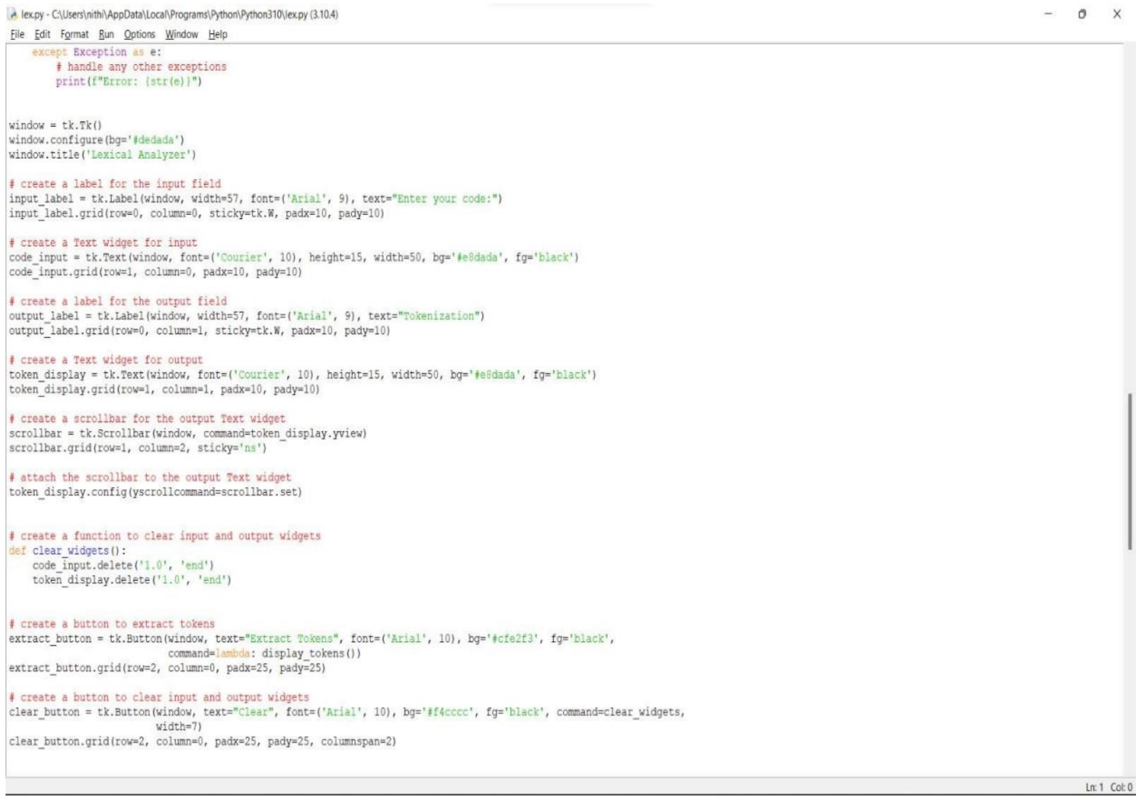
4.4.1 FRONTEND



```
(function () {
  function fibonacci(x) {
    return x <= 1 ? x : fibonacci(x - 1) + fibonacci(x - 2);
  }
  global.x = fibonacci(10);
})();
```

```
(function () {
  var _$0 = this;

  _$0.x = 55;
}).call(this);
```

4.4.2 BACKEND


lex.py - C:\Users\nithi\AppData\Local\Programs\Python\Python310\lex.py (3.10.4)

```
    except Exception as e:
        # handle any other exceptions
        print(f"Error: {str(e)}")


window = tk.Tk()
window.configure(bg='#dedada')
window.title('Lexical Analyzer')

# create a label for the input field
input_label = tk.Label(window, width=57, font=('Arial', 9), text="Enter your code:")
input_label.grid(row=0, column=0, sticky=tk.W, padx=10, pady=10)

# create a Text widget for input
code_input = tk.Text(window, font=('Courier', 10), height=15, width=50, bg='#e8dada', fg='black')
code_input.grid(row=1, column=0, padx=10, pady=10)

# create a label for the output field
output_label = tk.Label(window, width=57, font=('Arial', 9), text="Tokenization")
output_label.grid(row=0, column=1, sticky=tk.W, padx=10, pady=10)

# create a Text widget for output
token_display = tk.Text(window, font=('Courier', 10), height=15, width=50, bg='#e8dada', fg='black')
token_display.grid(row=1, column=1, padx=10, pady=10)

# create a scrollbar for the output Text widget
scrollbar = tk.Scrollbar(window, command=token_display.yview)
scrollbar.grid(row=1, column=2, sticky='ns')

# attach the scrollbar to the output Text widget
token_display.config(yscrollcommand=scrollbar.set)


# create a function to clear input and output widgets
def clear_widgets():
    code_input.delete('1.0', 'end')
    token_display.delete('1.0', 'end')


# create a button to extract tokens
extract_button = tk.Button(window, text="Extract Tokens", font=('Arial', 10), bg='#cfe2f3', fg='black',
                           command=lambda: display_tokens())
extract_button.grid(row=2, column=0, padx=25, pady=25)

# create a button to clear input and output widgets
clear_button = tk.Button(window, text="Clear", font=('Arial', 10), bg='#f4cccc', fg='black', command=clear_widgets,
                         width=7)
clear_button.grid(row=2, column=0, padx=25, pady=25, columnspan=2)
```

CHAPTER 5 RESULT

We have successfully implemented the different phases of a compiler, including the lexical analyzer, intermediate code generation, quadruples, and triples, in our project. By utilizing modern desktop application technologies like C# and the .NET framework, we have made the compiler more accessible to a wider user base. It effectively translates source code into executable code, making it a valuable tool for developers and programmers. While we encountered challenges during the development process, such as ensuring the accuracy of intermediate code generation, we overcame them through thorough planning and testing. Overall, our project showcases our proficiency in compiler development and the practical application of concepts learned in our coursework. We are confident that our compiler will be a valuable asset to the programming community, and we eagerly anticipate its future use and impact.

CHAPTER 6 CONCLUSION

In conclusion, the creation of a compiler that incorporates the different stages of the compilation process has been both challenging and rewarding. Building the lexical analyzer, intermediate code generation, and optimization components has deepened our understanding of compilers and emphasized the significance of each phase in compilation. By integrating modern web technologies, we have enhanced the userfriendliness and accessibility of the compiler, making it beneficial to a wider range of users. We anticipate that our compiler will serve as a valuable tool for developers and programmers, and we eagerly anticipate its future applications.

Throughout the development process, we have gained invaluable insights into software engineering and project management. We have recognized the importance of meticulous planning, thorough testing, and comprehensive documentation in ensuring project success. Moreover, we have discovered the significance of effective communication and collaboration within a team, as they contribute to more efficient and successful outcomes.

Overall, our project has provided us with a valuable learning experience, allowing us to apply classroom knowledge to real-world applications. We take pride in our accomplishments and eagerly anticipate the utilization of our newfound knowledge in future projects and endeavors.