

Garnish Compiler – Parser Generator

Submitted by

**Sahil Satasiya [RA2011026010110]
Utkarsh Srivastava [RA2011026010104]**

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



SCHOOL OF COMPUTING

**COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR - 603203**

May 2023



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

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203**

BONAFIDE CERTIFICATE

Certified that **18CSC304J – COMPILER DESIGN** project report titled “Garnish Compiler” is the bonafide work of Sahil Satasiya [RA2011026010110], Utkarsh Srivastava [RA2011026010104] 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

The project is a compiler design project based on the generation of Canonical Parser. The canonical parser operates based on the concept of a context-free grammar and employs bottom-up parsing algorithms such as LR(0), SLR(1), LALR(1), or LR(1). It systematically builds a parsing table by analyzing the grammar's production rules, utilizing lookahead symbols to determine the appropriate actions for reducing or shifting. This process leads to the construction of a deterministic finite automaton (DFA) or a state machine that efficiently handles the parsing tasks. The aim of the project is to develop a table which represent the CLR Parsing Table which analyze the grammar's production. The project consists of a code which is first meant to generate the First and Follow of all the Non-Terminals, following by calculating the item sets and finally calculating the CLR Parsing Table.

TABLE OF CONTENTS

ABSTRACT	3
Chapter 1	5
1.1 Introduction	5
1.2 Problem statement	6
1.3 Objective	6
1.4 Hardware requirement	7
1.5 Software requirement	7
Chapter 2 – Anatomy of Compiler	8
2.1 Lexical Analyzer	8
2.2 Intermediate Code Generation	8
2.3 Parser Generator	9
2.3 Quadruple	10
2.4 Triple	11
Chapter 3 – Architecture and Component	12
3.1 Architecture Diagram	12
3.2 Component Diagram	13
3.2.1 Lexical Analyzer	13
3.2.2 Intermediate Code Generator	14
3.2.3 Parser Generator	14
3.2.3 Quadruple	15
3.2.4 Triple	16
Chapter 4 – Coding and Testing	17
4.1 Lexical Analysis	17
4.1.1 Frontend	17
4.1.2 Backend	18
4.2 Intermediate Code Generator	19
4.2.1 Frontend	19
4.2.2 Backend	20
4.3 Quadruple	21
4.3.1 Frontend	21
4.3.2 Backend	21
4.4 Triple	22
4.4.1 Frontend	22
4.4.2 Backend	22
Chapter 5 – Result	23
Chapter 6 – Conclusion	24

CHAPTER 1

1.1 INTRODUCTION

The development of a tool that allows developers to input their code and see the output of what the compiler would produce is an excellent tool for developers. As software development becomes increasingly complex, the ability to test code and identify potential errors before deploying it is critical. This tool provides developers with a convenient way to quickly and efficiently test their code without the need to install and configure a complete development environment.

Consider a scenario where a developer is working on a new feature for an application. The feature requires the use of a complex algorithm that the developer has not worked with before. The developer writes the code and attempts to run it, but immediately receives an error message from the compiler. Without the ability to test the code in isolation, the developer must spend significant time trying to isolate the error and determine how to fix it.

However, with the use of the tool we have developed, the developer can simply input the code into the website and select the appropriate part of the compiler to test it. In this case, the developer could select the lexical analyzer to identify and correct any syntax errors. The tool quickly identifies the error and provides a clear message to the developer on how to fix it. With the error corrected, the developer can then run the code again and see the output of what the compiler would produce. This tool saves the developer a significant amount of time and effort in the debugging process.

Additionally, this tool is also useful for teaching programming. Beginners can use the tool to learn the basics of programming without the need to install and configure a development environment. The tool provides an easy-to-use interface that allows users to write code, see the output of what the compiler would produce, and make changes as needed. The feedback provided by the tool is immediate and clear, which helps beginners quickly identify and correct errors.

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, and help them deliver high-quality code.

1.3 OBJECTIVES

- Developed a website that allows developers to input their code and see the output of what the compiler would produce.
- Implemented a lexical analyzer, intermediate code generation, and quadruple triple in Python.
- Used ReactJS as frontend and NodeJS with ExpressJS as backend.
- When the server API is called, based on the part of the compiler selected from drop down, that part's shell script is executed and stored into a buffer.
- Utilized Linux file directory to store the buffer and read it for the response to the frontend REST API and display the output on the website.

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 simultaneously.
- Sufficient disk space to store the code files and other resources.

1.5 SOFTWARE REQUIREMENTS

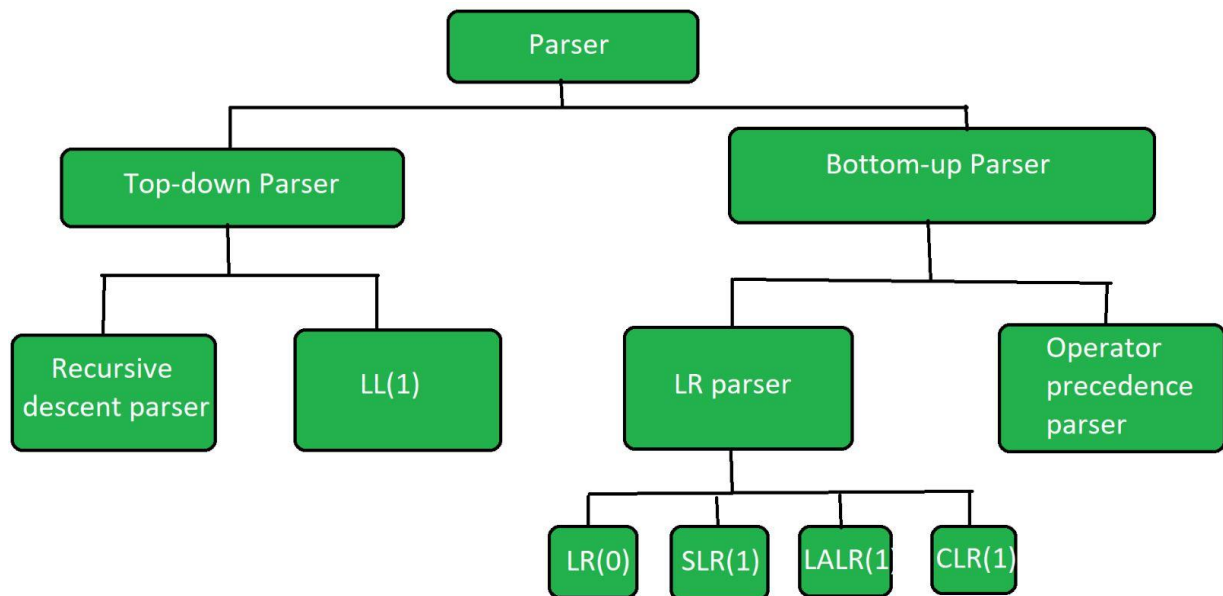
- Operating system: Linux or compatible OS.
- Python interpreter installed on the server.
- NodeJS and Express JS installed for the backend.
- A web browser to access the website.

CHAPTER 2

ANATOMY OF A COMPILER

2.1 PARSER GENERATOR

The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation(IR). The parser is also known as *Syntax Analyzer*.



Types of Parser:

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

- **Top-Down Parser:**

The top-down parser is the parser that **generates parse for the given input string** with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation. Further Top-down parser is classified into 2 types: A recursive descent parser, and Non-recursive descent parser.

- **Bottom-up Parser:**

Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.

Further Bottom-up parser is classified into two types: LR parser, and Operator precedence parser.

CHAPTER 3

ARCHITECTURE AND COMPONENTS

3.1 ARCHITECTURE DIAGRAM

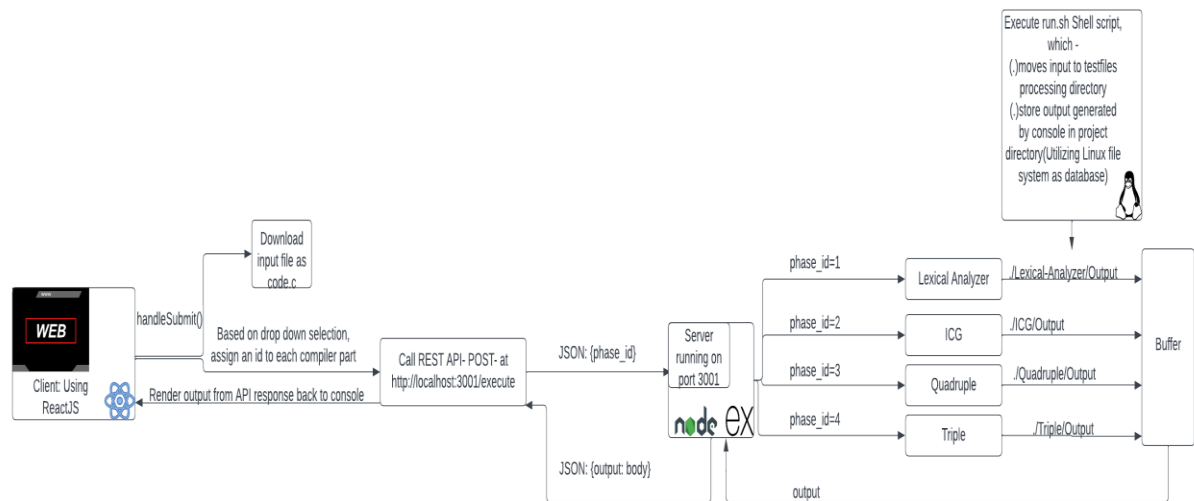
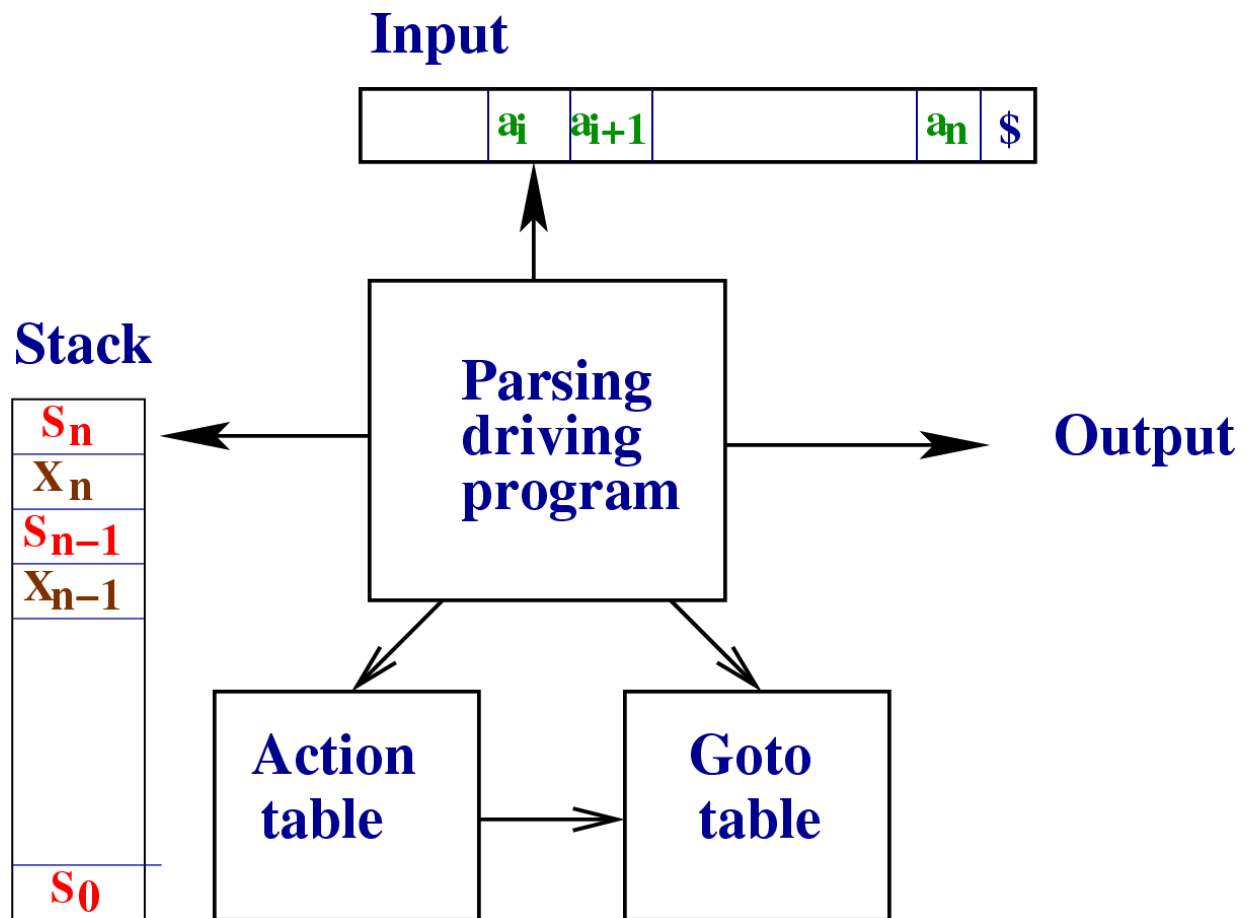


FIG 3.1 Architecture Diagram of the Project

The project consists of several components that work together to achieve its goal. The Front-end UI is responsible for presenting the user interface to the end-user, and it is built using ReactJS, which is a popular JavaScript library for building user interfaces. The REST API serves as the communication interface between the front-end UI and the back-end logic. It is implemented using a RESTful architecture that enables communication between different components using HTTP protocols. On the other hand, the back-end logic consists of several components, including the Lexical Analyzer and the Intermediate Code Generator (ICG). The Lexical Analyzer is responsible for analysing the input source code and generating a stream of tokens, while the ICG is responsible for generating an intermediate code representation of the input source code. The Quadruple and Triple data structures are used to represent the intermediate code generated by the ICG, and they are also implemented as a part of the back-end logic. Overall, the project architecture involves the front-end UI, the REST API, and the back-end logic components, including the Lexical Analyzer, ICG, and data structures. Each component has a specific role and works together to achieve the project's objective.

3.2 COMPONENTS DIAGRAMS

3.2.1 CANONICAL PARSING (CLR)



CLR Parser (Canonical LR Parser):

A CLR parser is a type of bottom-up parser used in computer science for parsing programming languages. CLR stands for "Canonical LR", and the term refers to a set of parsing algorithms that can handle a large class of context-free grammars. A CLR parser operates by building a parse tree from the bottom up, starting with the leaves and working its way up to the root.

The process of canonical parsing involves constructing a parsing table that guides the parser in making decisions about reducing or shifting grammar symbols. This table is built by analyzing the production rules of the grammar and utilizing lookahead symbols to determine the appropriate actions. The lookahead symbols help in resolving conflicts that may arise when multiple parsing actions are possible at a given point.

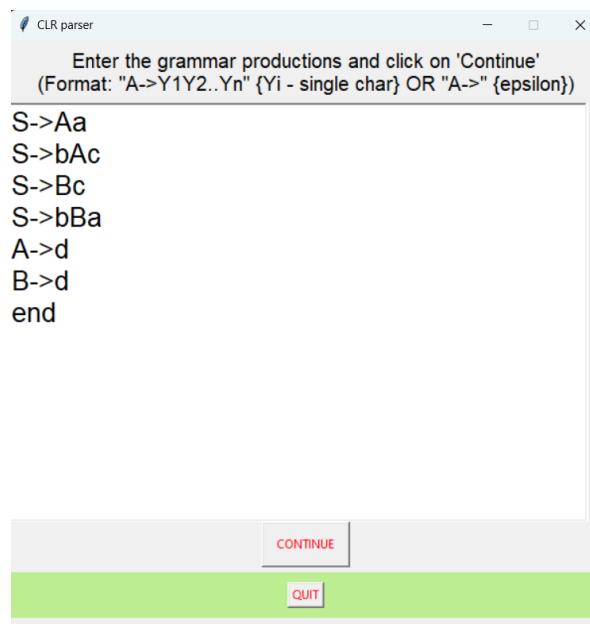
CHAPTER 4

CODING AND TESTING

4.1 CANONICAL PARSING

4.1.1 FRONTEND

INPUT (GRAMMER)



CLR parser

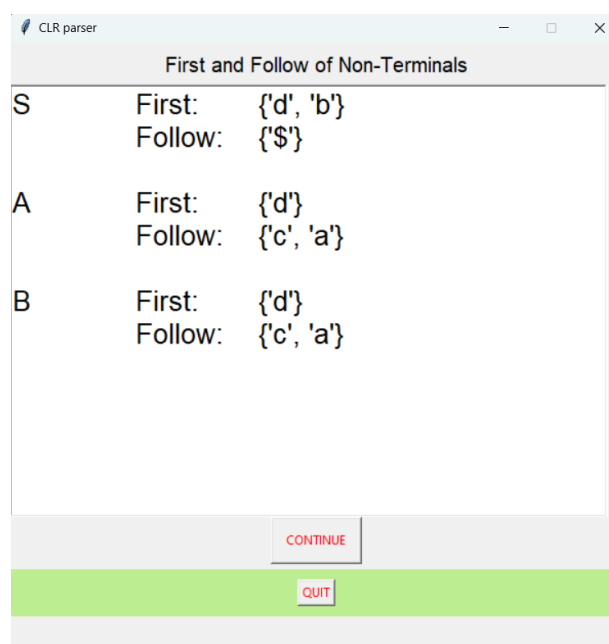
Enter the grammar productions and click on 'Continue'
(Format: "A->Y1Y2..Yn" {Yi - single char} OR "A->" {epsilon})

S->Aa
S->bAc
S->Bc
S->bBa
A->d
B->d
end

CONTINUE

QUIT

FIRST AND FOLLOW CALCULATION



CLR parser

First and Follow of Non-Terminals

S	First: { 'd', 'b' }
	Follow: { '\$' }
A	First: { 'd' }
	Follow: { 'c', 'a' }
B	First: { 'd' }
	Follow: { 'c', 'a' }

CONTINUE

QUIT

ITEM SETS CALCULATION

CLR parser

Canonical LR(1) Items

10:

S->bA.c, \$

19:

S->bB.a, \$

110:

A->d., c
B->d., a

111:

S->bAc., \$

112:

CONTINUE

QUIT

PASRSING TABLE

CLR parser

CLR(1) Table

	a	b	c	d	\$	S	A
B							
0		s4		s5		1	2
3							
1					accept		
2	s6						
3			s7				
4				s10			8
9							
5	r5		r6				
6					r1		
7					r3		
8			s11				

QUIT

4.1.2 BACKEND

```
1
2 from tkinter import *
3 from collections import deque, OrderedDict
4 from pprint import pprint
5 import firstfollow
6 from firstfollow import production_list, nt_list as ntl, t_list as tl
7 nt_list, t_list=[], []
8 j=None
9
10 class Application(Frame):
11
12     def __init__(self, master=None):
13         Frame.__init__(self, master)
14         self.master=master
15         master.title("CLR parser")
16         master.geometry("600x600")
17         master.resizable(0, 0)
18         self.pack()
19         self.createWidgets(master)
20
21     def center(self, toplevel):
22         toplevel.update_idletasks()
23         w = toplevel.winfo_screenwidth()
24         h = toplevel.winfo_screenheight()
25         size = tuple(int(_) for _ in toplevel.geometry().split('+')[0].split('x'))
26         x = w/2 - size[0]/2
27         y = h/2 - size[1]/2
28         toplevel.geometry("%dx%d+%d+%d" % (size + (x, y)))
29
30     def createWidgets(self, master):
31         self.center(master)
32         self.mframe=Frame(master)
33         self.mframe.pack(padx=0, pady=0, ipadx=0, ipady=0)
34         frame=Frame(self.mframe)
35         frame.pack(side=TOP)
36         frame2=Frame(self.mframe)
37         frame2.pack()
38
39         bottomframe=Frame(self.mframe, bd=10, bg="#BCED91")
40         bottomframe.pack(side=BOTTOM, fill=BOTH, pady=5)
41
42         self.head=Label(frame, text='''Enter the grammar productions and click on 'Continue'
43 (Format: "A->V1V2.\n\n" (V1 - single char) OR "A->" {epsilon})''', font='Helvetica -20', fg="black")
44         self.head.pack(padx=5, pady=5)
45         self.make_tb(frame)
46
47         self.cont=Button(frame2, fg="red", text="CONTINUE", command=self.start)
48         self.cont.pack(ipadx=10, ipady=10, expand=1, side=BOTTOM)
49
50         Button(bottomframe, text="QUIT", fg="red", command=master.destroy).pack(fill=Y, expand=1, side=RIGHT)
51
52     def start(self):
53         pl=self.text.get("1.0", END).split("\n")+['']
54         pprint(pl)
55
56         self.head.config(text="First and Follow of Non-Terminals")
57         self.text.delete("1.0", END)
58         self.master.geometry("600x600")
```

```
62
63     global nt_list, t_list
64
65     firstfollow.production_list=firstfollow.main(pl)
66
67     for nt in ntl:
68         firstfollow.compute_first(nt)
69         firstfollow.compute_follow(nt)
70         self.text.insert(END, nt)
71         self.text.insert(END, "\tFirst:\t{}\n\n".format(firstfollow.get_first(nt)))
72         self.text.insert(END, "\tFollow:\t{}\n\n".format(firstfollow.get_follow(nt)))
73         #self.text.config(state=DISABLED)
74
75     augment_grammar()
76     nt_list=list(ntl.keys())
77     t_list=list(tl.keys()) + ['$']
78
79     #self.text.insert(END, "{}\n\n".format(nt_list))
80     #self.text.insert(END, "{}\n\n".format(t_list))
81     self.text.see(END)
82     self.text.config(state=DISABLED)
83
84     def more(self):
85         self.text.config(state=NORMAL)
86         global j
87         j=calc_states()
88         global nt_list, t_list
89
90         self.head.config(text="Canonical LR(1) Items")
91         self.text.delete("1.0", END)
92         self.cont.config(command=self.more2)
93         ctr=0
94
95         for s in j:
96             self.text.insert(END, "\n{i}:\n\n".format(ctr))
97             for i in s:
98                 self.text.insert(END, "\t{i}\n\n".format(i))
99             ctr+=1
100         self.text.see(END)
101         self.text.config(state=DISABLED)
102
103     def more2(self):
104         self.text.config(state=NORMAL)
105         global j
106         self.head.config(text="CLR(1) Table")
107         self.text.delete("1.0", END)
108         self.cont.destroy()
109
110         table=make_table(j)
111
112         sr, rr=0, 0
113
114         self.text.config(font='-size 12', height=20)
115         self.text.insert(END, "\t{}\t{}\n\n".format('\t'.join(t_list), '\t'.join(nt_list)))
116         for i, j in table.items():
```

```

class State:
    _id=0
    def __init__(self, closure):
        self.closure=closure
        self.no=State._id
        State._id+=1

class Item(str):
    def __new__(cls, item, lookahead=list()):
        self=str.__new__(cls, item)
        self.lookahead=lookahead
        return self

    def __str__(self):
        return super(Item, self).__str__()+"", "+'|'.join(self.lookahead)

def closure(items):

    def exists(newitem, items):

        for i in items:
            if i==newitem and sorted(set(i.lookahead))==sorted(set(newitem.lookahead)):
                return True
        return False

    global production_list

    while True:
        flag=0
        for i in items:

            if i.index('.')==len(i)-1: continue

            Y=i.split('->')[1].split('.')[1][0]

            if i.index('.')+1<len(i)-1:
                lastr=list(firstfollow.compute_first(i[i.index('.')+2])-set(chr(1013)))

            else:
                lastr=i.lookahead

            for prod in firstfollow.production_list:
                head, body=prod.split('->')

                if head!=Y: continue

                newitem=Item(Y+'->'+body, lastr)

                if not exists(newitem, items):
                    items.append(newitem)
                    flag=1
            if flag==0: break

```

```

for i in range(len(states)):
    states[i]=State(states[i])

for s in states:
    SLR_Table[s.no]=OrderedDict()

    for item in s.closure:
        head, body=item.split('->')
        if body=='.':
            for term in item.lookahead:
                if term not in SLR_Table[s.no].keys():
                    SLR_Table[s.no][term]='r'+str(getprodno(item))
                else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}
            continue

        nextsym=body.split('.')[1]
        if nextsym=='.':
            if getprodno(item)==0:
                SLR_Table[s.no]['$']='accept'
            else:
                for term in item.lookahead:
                    if term not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][term]='r'+str(getprodno(item))
                    else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}
                continue

        nextsym=nextsym[0]
        t=goto(s.closure, nextsym)
        if t != []:
            if nextsym in t_list:
                if nextsym not in SLR_Table[s.no].keys():
                    SLR_Table[s.no][nextsym]='s'+str(getstateno(t))
                else: SLR_Table[s.no][nextsym] |= {'s'+str(getstateno(t))}
            else: SLR_Table[s.no][nextsym] = str(getstateno(t))

    return SLR_Table

def augment_grammar():

    for i in range(ord('Z'), ord('A')-1, -1):
        if chr(i) not in nt_list:
            start_prod=firstfollow.production_list[0]
            firstfollow.production_list.insert(0, chr(i)+'->'+start_prod.split('->')[0])
            return

def main():
    root=tk()
    app=Application(master=root)
    app.mainloop()

    return

if __name__=="__main__":
    main()

```

CHAPTER 5

RESULT

The implementation of the phases of a compiler, including the lexical analyser, intermediate code generation, quadruples, and triples, has been successful in our project. The integration of modern web technologies, such as React JS and Node JS, has made the compiler more accessible to a wider range of users. Our compiler can effectively translate source code into executable code, making it a useful tool for developers and programmers. Through the development process, we encountered various challenges, such as ensuring the accuracy of the intermediate code generation process, but we were able to overcome these challenges through careful planning and testing. Overall, our project demonstrates our proficiency in compiler development and our ability to apply the concepts and tools learned in class to real-world applications. We believe that our compiler has the potential to be a valuable resource for the programming community and we are excited to see how it will be used in the future.

CHAPTER 6

CONCLUSION

In conclusion, the development of a compiler that implements the various phases of the compilation process has been a challenging and rewarding experience. Through the implementation of the lexical analyser, intermediate code generation, quadruples, and triples, we have gained a deeper understanding of the inner workings of compilers and the importance of each phase in the compilation process. The integration of modern web technologies has made the compiler more user-friendly and accessible to a wider range of users. We believe that our compiler has the potential to be a valuable resource for developers and programmers, and we look forward to seeing how it will be used in the future.

The development process has also taught us valuable lessons about software engineering and project management. We learned the importance of careful planning, testing, and documentation in ensuring the success of a project. Additionally, we discovered the importance of communication and collaboration in a team setting, and how effective teamwork can lead to more efficient and effective outcomes.

Overall, our project has been a valuable learning experience that has allowed us to apply the concepts and tools learned in class to a real-world application. We are proud of what we have accomplished and look forward to applying our newfound knowledge to future projects and endeavours.