COMPILER DESIGN

A Mini Project Report Submitted by

NIDHI RAI (4NM18CS103) NISHMITHA K SUVARNA (4NM18CS104)

UNDER THE GUIDANCE OF

Mrs. Anusha Anchan
Assistant Professor GD-II
Department of Computer Science and Engineering

in partial fulfillment of the requirements for the award of the Degree of

Bachelor of Engineering in
Computer Science & Engineering
from

Visveshvaraya Technological University, Belgaum

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
N.M.A.M. INSTITUTE OF TECHNOLOGY

(An Autonomous Institution under VTU, Belgaum) (AICTE approved, NBA Accredited, ISO 9001:2008 Certified) NITTE -574 110, Udupi District, KARNATAKA.

DEC 2021

N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India
(ISO 9001:2015 Certified), Accredited with 'A' Grade by NAAC
☎: 08258 - 281039 – 281263, Fax: 08258 – 281265
Department of Computer Science and Engineering

B.E. CSE Program Accredited by NBA, New Delhi from 1-7-2018 to 30-6-2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the Mini Project work entitled

COMPILER DESIGN

is a bonafide work carried out by

NIDHI RAI(4NM18CS103) NISHMITHA K SUVARNA(4NM18CS104)

in partial fulfillment of the requirements for the award of Bachelor of Engineering Degree in Computer Science and Engineering prescribed by Visvesvaraya Technological University, Belgaum during the year 2021-2022.

It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report.

The Mini project report has been approved as it satisfies the academic requirements in respect of the project work prescribed for the Bachelor of Engineering Degree.

Signature of Guide Signature of HOD

ACKNOWLEDGEMENT

We believe that our project will be complete only after we thank the people who have contributed to make this project successful.

First and foremost, our sincere thanks to our beloved principal, **Dr. Niranjan N. Chiplunkar** for giving us an opportunity to carry out our project work at our college and providing us with all the needed facilities.

We express our deep sense of gratitude and indebtedness to our guide **Mrs. Anusha Anchan**, Assistant Professor GD-II, Department of Computer Science and Engineering, for her inspiring guidance, constant encouragement, support and suggestions for improvement during the course of our project.

We sincerely thank **Dr. Jyothi Shetty**, Head of Department of Computer Science and Engineering, Nitte Mahalinga Adyantaya Memorial Institute of Technology, Nitte.

We also thank all those who have supported us throughout the entire duration of our project.

Finally, we thank the staff members of the Department of Computer Science and Engineering and all our friends for their honest opinions and suggestions throughout the course of our project.

NIDHI RAI(4NM18CS103)
NISHMITHA K SUVARNA(4NM18CS104)

TABLE OF CONTENTS

ABSTRACT

The computer is an integral tool in our lives because it turns application that solve many real-life problems. Computer programmers write programs to perform various tasks. The high-level programming languages currently used can only be understood by human beings, but not by the computer. It requires a compiler to convert this high-level language to a language that can be understood by the machine.

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called compilers.
This report contains the details of how one can develop the simple compiler for given language using Lex (Lexical Analyzer Generator) and YACC (Yet Another Compiler). Lex tool helps write programs whose control flow is directed by instances of regular expressions in the input stream.

Lex tool source is the table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. On the other hand, YACC tool receives input of the user grammar. Starting from this grammar it generates the C source code for the parser. YACC invokes Lex to scan the source code and uses the tokens returned by Lex to build a syntax tree. With the help of YACC and Lex tool one can write their own compiler.

INTRODUCTION

COMPILER:
A compiler is a software that takes a program written in a high-level language and translates it into an equivalent program in a target language. Most specifically a compiler takes a computer program and translates it into an object program. Some other tools associated with the compiler are responsible for making an object program into executable form.

Source program – It is normally written in a high-level programming language. It contains a set of rules, symbols and special words used to construct a computer program.

Target program – It is normally the equivalent program in machine code. It contains the binary representation of the instructions that the hardware of computer can perform.

Error Message – A message issued by the compiler due to detection of syntax errors in the source program.

Compilation is a large process. It is often broken into stages. Many phases of the compiler try and optimize by translating one form into a better (more efficient) form. Most of compiling is about "pattern matching" languages and tools that support pattern matching, are very useful. An efficient compiler must preserve semantics of the source program and it should create an efficient version of the target language.

PHASES OF COMPILER:

Typically, a compiler includes several functional parts. For example, a conventional compiler may include a lexical analyzer that looks at the source program and identifies successive "tokens" in the source program. A conventional compiler also includes a parser or syntactical analyzer, which takes as an input a grammar defining the language being compiled and a series of actions associated with the grammar.

The syntactical analyzer builds a “parse tree” for the statements in the source program in accordance with the grammar productions and actions. For each statement in the input source program, the syntactical analyzer generates a parse tree of the source input in a recursive, “bottom-up” manner in accordance with relevant productions and actions in the grammar. Generation of the parse tree allows the syntactical analyzer to determine whether the parts of the source program comply with the grammar. If not, the syntactical analyzer generates an error

CLASSIFICATION OF COMPILER PHASES:
There are two major parts of a compiler phases: Analysis and Synthesis.
In analysis phase, an intermediate representation is created from the given source program that contains:

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer

In synthesis phase, the equivalent target program is created from this intermediate representation. This contains:

- Intermediate code Generator
- Code Optimisation
- Code Generation

1: LEXICAL ANALYZER
Lexical analyzer takes the source program as an input and produces a string of tokens or lexemes. Lexical Analyzer reads the source program character by character and returns the tokens of the source program. The process of generation and returning the tokens is called lexical analysis.

2: SYNTAX ANALYZER

A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program. In other words, a Syntax Analyzer takes output of lexical analyser (list of tokens) and produces a parse tree. A syntax analyser is also called as a parser. The parser checks if the expression made by the tokens is syntactically correct.

3. SEMANTIC ANALYSER

Semantic analyser takes the output of syntax analyser. Semantic analyser checks a source program for semantic consistency with the language definition. It also gathers type information for use in intermediate-code generation.

4. INTERMEDIATE CODE GENERATION

After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language.

5. CODE OPTIMISER

The code optimizer takes the code produced by the intermediate code generator. The code optimizer reduces the code (if the code is not already optimized) without changing the meaning of the code. The optimization of code is in terms of time and space.

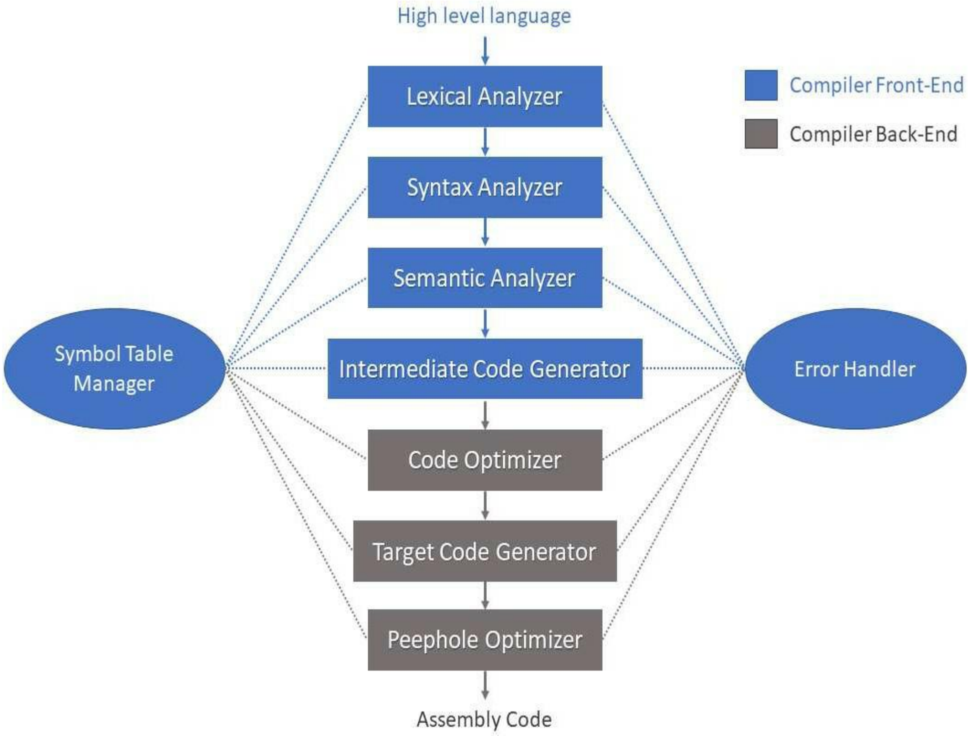
6. CODE GENERATION

This produces the target language in a specific architecture. The target program is normally is an object file containing the machine codes. Memory locations are selected for each of the variables used by the program.

SYMBOL TABLE

It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

PHASES OF COMPILER:



IMPLEMENTATION

PROBLEM STATEMENT :

```
int main()
begin
    int count=1;
    while(n>1)
        count=count+1;
        n=n/2;
    end while
return count
end
```

LEXICAL ANALYSIS :

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyser finds a token invalid, it generates an error. The lexical analyser works closely with the syntax analyser. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyser when it demands. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase.

Token:

Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers
2) keywords
3) operators
4) special symbols
5) constants

Pattern:
A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme:
A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

<u>Lex Program</u> :

```
%{
#include "y.tab.h"
%}
%%
"int" {return INT;}
"float" {return FLOAT;}
"char" {return CHAR;}
"void" {return VOID;}
"main" {return MAIN;}
"begin" {return BEG;}
"end" {return END;}
"(" {return OPENP;}
")" {return CLOSEP;}
"return" {return RETURN;}
"while" {return WHILE;}
[a-zA-Z][a-zA-Z0-9]* {return ID;}
[0-9]+ {return NUM;}
[ ] {return SP;}
[;] {return SC;}
[ ]* {return SPS;}
[\t]* {return TAB;}
[\n] {return NL;}
[,] {return CM;}
">="|"<="|"<"|">"|"=="|"!=" {return RELOP;}
[+\-=*/] {return OPER;}
. {return yytext[0];}
%%
int yywrap(){
        return 1;
}
```

<u>YACC Program</u> :

```
%{
#include<stdio.h>
%}
%token MAIN OPENP CLOSEP BEG END INT VOID CHAR FLOAT ID NUM SC NL SP CM TAB SPS WHILE RELOP OPER RETURN
%%
stmt: type SP MAIN OPENP CLOSEP NL space BEG NL space stmts space RETURN SP vals NL space END {printf("\nValid string\n"); return 1;};
space: SPS space|
       TAB space|
       SP space|;
vals: ID|
      NUM;
stmts: type SP space varlist SC NL stmts|
       while_stmt NL stmts|
       expr SC NL stmts|
       ;
varlist: ID space|
         expr space|
         expr space CM|
         ID space CM;
type: INT|
      VOID|
      FLOAT|
      CHAR;
while_stmt: WHILE OPENP space vals space RELOP space vals space CLOSEP NL space stmts space END SP WHILE;
expr: expr space OPER space expr|
      vals;
%%
int yyerror(char* msg){
        printf("\nInvalid string\n");
        return 0;
}
void main(){
        printf("Enter the string:\n");
        yyparse();
}
```

<u>Output</u> :

```
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ gedit project.l
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ gedit project.y
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ lex project.l
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ yacc -d project.y
project.y: warning: 21 shift/reduce conflicts [-Wconflicts-sr]
project.y: warning: 3 reduce/reduce conflicts [-Wconflicts-rr]
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ cc y.tab.c lex.yy.c
y.tab.c: In function ‘yyparse’:
y.tab.c:1298:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
 1298 |       yychar = yylex ();
      |                ^~~~~
y.tab.c:1431:7: warning: implicit declaration of function ‘yyerror’; did you mean ‘yyerrok’? [-Wimplicit-function-declaration]
 1431 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ ./a.out
Enter the string:
int main()
begin
int count=1;
while(n>1)
count=count+1;
n=n/2;
end while
return count
end

Valid string
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$ ./a.out
Enter the string:
void main()
end

Invalid string
nidhirai@nidhirai-VirtualBox:~/Desktop/cd project$
```

<u>SYNTAX ANALYSIS</u> :

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in the figure below, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing

PARSER:

Parser is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer. The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

CODE.py

```
import pandas as pd
import numpy as np
from lexer import get_tokens

EPSILON = "a"

def get_productions(X):
    # This function will return all the productions X->A of the grammar
    productions = []
    for prod in grammar:
        lhs, rhs = prod.split('->')
        # Check if the production has X on LHS
        if lhs == X:
            # Introduce a dot
            rhs = '.'+rhs
            productions.append('->'.join([lhs, rhs]))
    return productions
```

```
def closure(I):
    # This function calculates the closure of the set of items I
    for production, a in I:
        # This means that the dot is at the end and can be ignored
        if production.endswith("."):
            continue
        lhs, rhs = production.split('->')
        alpha, B_beta = rhs.split('.')
        B = B_beta[0]
        beta = B_beta[1:]
        beta_a = beta + a
        first_beta_a = first(beta_a)
        for b in first_beta_a:
            B_productions = get_productions(B)
            for gamma in B_productions:
                new_item = (gamma, b)
                if (new_item not in I):
                    I.append(new_item)
    return I


def get_symbols(grammar):
    # Check the grammar and get the set of terminals and non_terminals
    terminals = set()
    non_terminals = set()
    for production in grammar:
        lhs, rhs = production.split('->')
        # Set of non terminals only
        non_terminals.add(lhs)
        for x in rhs:
            # Add add symbols to terminals
            terminals.add(x)
    # Remove the non terminals
    terminals = terminals.difference(non_terminals)
    terminals.add('$')
    return terminals, non_terminals


def first(symbols):
    # Find the first of the symbol 'X' w.r.t the grammar
    final_set = []
    for X in symbols:
        first_set = []  # Will contain the first(X)
        if isTerminal(X):
            final_set.extend(X)
            return final_set
        else:
            for production in grammar:
                # For each production in the grammar
                lhs, rhs = production.split('->')
```

```
            if lhs == X:
                # Check if the LHS is 'X'
                for i in range(len(rhs)):
                    # To find the first of the RHS
                    y = rhs[i]
                    # Check one symbol at a time
                    if y == X:
                        # Ignore if it's the same symbol as X
                        # This avoids infinite recursion
                        continue
                    first_y = first(y)
                    first_set.extend(first_y)
                    # Check next symbol only if first(current) contains EPSILON
                    if EPSILON in first_y:
                        first_y.remove(EPSILON)
                        continue
                    else:
                        # No EPSILON. Move to next production
                        break
                else:
                    # All symbols contain EPSILON. Add EPSILON to first(X)
                    # Check to see if some previous production has added epsilon already
                    if EPSILON not in first_set:
                        first_set.extend(EPSILON)

                    # Move onto next production
        final_set.extend(first_set)
        if EPSILON in first_set:
            continue
        else:
            break
    return final_set


def isTerminal(symbol):
    # This function will return if the symbol is a terminal or not
    return symbol in terminals


def shift_dot(production):
    # This function shifts the dot to the right
    lhs, rhs = production.split('->')
    x, y = rhs.split(".")
    if(len(y) == 0):
        print("Dot at the end!")
        return
    elif len(y) == 1:
        y = y[0]+"."
    else:
        y = y[0]+"."+y[1:]
```

```
    rhs = "".join([x, y])
    return "->".join([lhs, rhs])


def goto(I, X):
    # Function to calculate GOTO
    J = []
    for production, look_ahead in I:
        lhs, rhs = production.split('->')
        # Find the productions with .X
        if "."+X in rhs and not rhs[-1] == '.':
            # Check if the production ends with a dot, else shift dot
            new_prod = shift_dot(production)
            J.append((new_prod, look_ahead))
    return closure(J)


def set_of_items(display=False):
    # Function to construct the set of items
    num_states = 1
    states = ['I0']
    items = {'I0':  closure([('P->.S', '$')])}
    for I in states:
        for X in pending_shifts(items[I]):
            goto_I_X = goto(items[I], X)
            if len(goto_I_X) > 0 and goto_I_X not in items.values():
                new_state = "I"+str(num_states)
                states.append(new_state)
                items[new_state] = goto_I_X
                num_states += 1
    if display:
        for i in items:
            print("State", i, ":")
            for x in items[i]:
                print(x)
            print()

    return items


def pending_shifts(I):
    # This function will check which symbols are to be shifted in I
    symbols = []  # Will contain the symbols in order of evaluation
    for production, _ in I:
        lhs, rhs = production.split('->')
        if rhs.endswith('.'):
            # dot is at the end of production. Hence, ignore it
            continue
        # beta is the first symbol after the dot
        beta = rhs.split('.')[1][0]
```

```
        if beta not in symbols:
            symbols.append(beta)
    return symbols


def done_shifts(I):
    done = []
    for production, look_ahead in I:
        if production.endswith('.') and production != 'P->S.':
            done.append((production[:-1], look_ahead))
    return done


def get_state(C, I):
    # This function returns the State name, given a set of items.
    key_list = list(C.keys())
    val_list = list(C.values())
    i = val_list.index(I)
    return key_list[i]


def CLR_construction(num_states):
    # Function that returns the CLR Parsing Table function ACTION and GOTO
    C = set_of_items()  # Construct collection of sets of LR(1) items

    # Initialize two tables for ACTION and GOTO respectively
    ACTION = pd.DataFrame(columns=terminals, index=range(num_states))
    GOTO = pd.DataFrame(columns=non_terminals, index=range(num_states))

    for Ii in C.values():
        # For each state in the collection
        i = int(get_state(C, Ii)[1:])
        pending = pending_shifts(Ii)
        for a in pending:
            # For each symbol 'a' after the dots
            Ij = goto(Ii, a)
            j = int(get_state(C, Ij)[1:])
            if isTerminal(a):
                # Construct the ACTION function
                ACTION.at[i, a] = "Shift "+str(j)
            else:
                # Construct the GOTO function
                GOTO.at[i, a] = j

        # For each production with dot at the end
        for production, look_ahead in done_shifts(Ii):
            # Set GOTO[I, a] to "Reduce"
            ACTION.at[i, look_ahead] = "Reduce " + str(grammar.index(production)+1)

        # If start production is in Ii
```

```
        if ('P->S.', '$') in Ii:
            ACTION.at[i, '$'] = "Accept"

    # Remove the default NaN values to make it clean
    ACTION.replace(np.nan, '', regex=True, inplace=True)
    GOTO.replace(np.nan, '', regex=True, inplace=True)

    return ACTION, GOTO


def parse_string(string, ACTION, GOTO):
    # This function parses the input string and returns the talble
    row = 0
    # Parse table column names:
    cols = ['Stack', 'Input', 'Output']
    if not string.endswith('$'):
        # Append $ if not already appended
        string = string+'$'
    ip = 0  # Initialize input pointer
    # Create an initial (empty) parsing table:
    PARSE = pd.DataFrame(columns=cols)
    # Initialize input stack:
    input = list(string)
    # Initialize grammar stack:
    stack = ['$', '0']
    while True:
        S = int(stack[-1])  # Stack top
        a = input[ip]  # Current input symbol
        action = ACTION.at[S, a]
        # New row to be added to the table:
        new_row = ["".join(stack), "".join(input[ip:]), action]
        if 'S' in action:
            # If it is a shift operation:
            S1 = action.split()[1]
            stack.append(a)
            stack.append(S1)
            ip += 1
        elif "R" in action:
            # If it's a reduce operation:
            i = int(action.split()[1])-1
            A, beta = grammar[i].split('->')
            for _ in range(2*len(beta)):
                # Remove 2 * rhs of the production
                stack.pop()
            S1 = int(stack[-1])
            stack.append(A)
            stack.append(str(GOTO.at[S1, A]))
            # Replace the number with the production for clarity:
            new_row[-1] = "Reduce "+grammar[i]
        elif action == "Accept":
```

```
            # Parsing is complete. Return the table
            PARSE.loc[row] = new_row
            return PARSE
        else:
            # Some conflict occurred.
            print("Invalid input!!!")
            return PARSE
        # All good. Append the new row and move on to the next.
        PARSE.loc[row] = new_row
        row += 1


def get_grammar(filename):
    grammar = []
    F = open(filename, "r")
    for production in F:
        grammar.append(production[:-1])
    return grammar


if __name__ == "__main__":
    grammar = get_grammar("grammar")
    terminals, non_terminals = get_symbols(grammar)
    symbols = terminals.union(non_terminals)

    start = [('P->.S', '$')]
    I0 = closure(start)
    goto(I0, '*')
    C = set_of_items(display=True)
    ACTION, GOTO = CLR_construction(num_states=len(C))
    print(ACTION)
    print(GOTO)
    # Demonstrating helper functions:
    string = None
    try:
        string = "".join(get_tokens("code"))
    except:
        pass
    if string!=None:
        print(string)
    try:
        PARSE_TABLE = parse_string(string, ACTION, GOTO)
        print(PARSE_TABLE)
    except:
        print('Invalid input:(')
```

grammar.txt

```
S->tm()bP
P->tvoh;Q
Q->w(voh)R
R->vovoh;T
T->vovoh;U
U->ewX
X->rve
```

invalidcode.txt

```
int main()
begin
int L[10];
int maxval=L[0];
for i=1 to n-1 do
if L[i]>maxval
maxval=L[i];
endif
endfor
return(maxval)
End
```

lexer.py

```
SYMBOLS = ['(',
           ')',
           ';',
           ',',
           ':',
           '\'']

KEYWORDS = {'t': ['int', 'char'],
            'm': ['main'],
            'w': ['while'],
            'b': ['begin'],
            'e': ['end'],
            'o': ['+', '-', '=','==','<=','>=','/','>'],
            'k': ['\n'],
            'h': ['1','2'],
```

```
            'r': ['return']}

OPERATORS = ['+', '-', '=','==','<=','>=']

line_count = 0


def getIndex(word):
    keys = list(KEYWORDS.keys())
    values = list(KEYWORDS.values())
    for value in values:
        if word in value:
            i = values.index(value)
            return keys[i]


def get_tokens(filename):
    tokens = []
    F = open(filename, "r")
    for line in F:
        for word in line.split():
            # Check if it's an isolated keyword
            token = getIndex(word)
            if token in KEYWORDS:
                tokens.append(token)
            else:
                # Check if it's a keyword followed by a symbol
                buffer = []
                for character in word:
                    if character.isalnum():
                        # Serves a string builder
                        buffer.append(character)
                        current_word = "".join(buffer)
                        token = getIndex(current_word)
                        if token in KEYWORDS:
                            # A fully formed keyword has been detected
                            tokens.append(token)
                            buffer = []
                    elif character in SYMBOLS or character in OPERATORS:
                        if len(buffer) != 0:
                            tokens.append('v')
                            buffer = []
                        # If it's a special operator
                        if character in SYMBOLS:
```

```
                tokens.append(character)
            else:
                tokens.append("o")
        if len(buffer) != 0:
            tokens.append('v')
    return tokens
```

RESULTS

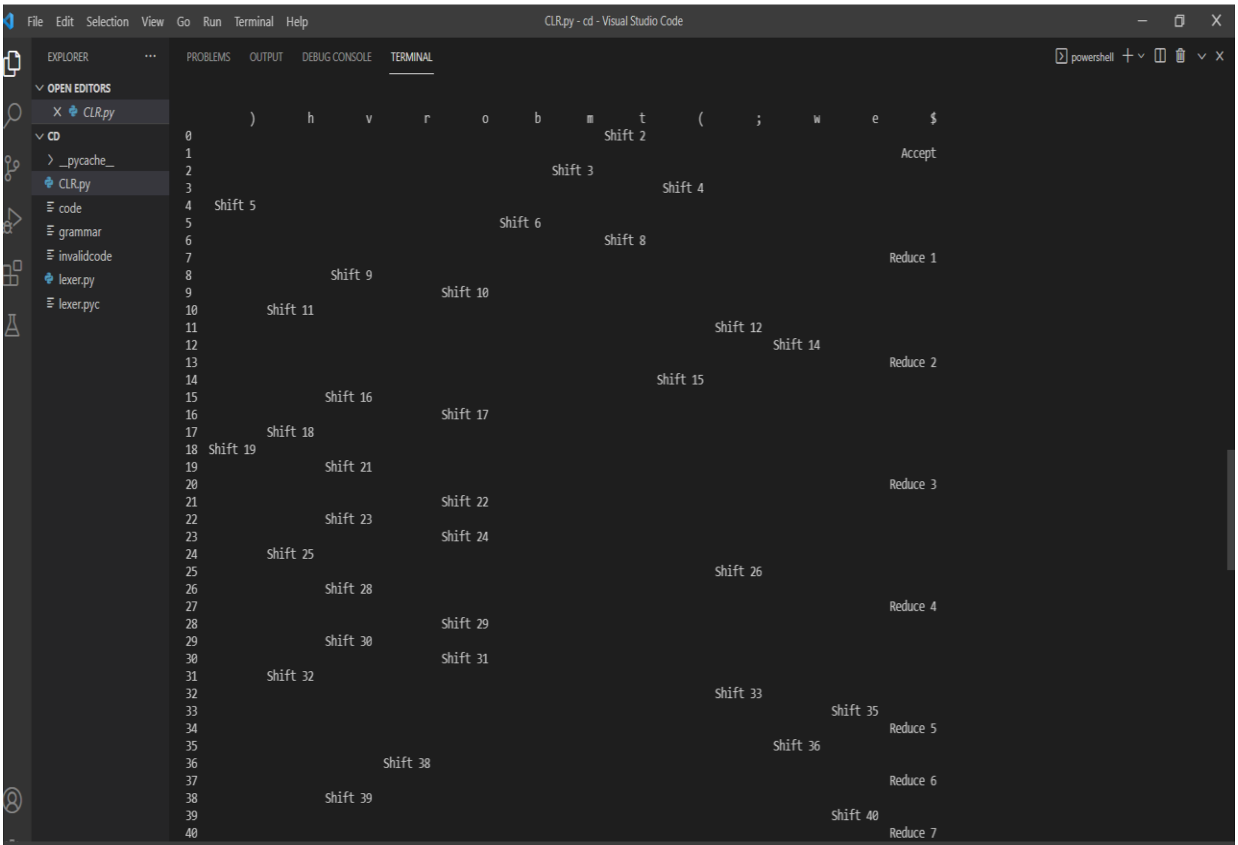
Closure Sets:

```
State I0 :
('P->.S', '$')
('S->.tm()bP', '$')

State I1 :
('P->S.', '$')

State I2 :
('S->t.m()bP', '$')

State I3 :
('S->tm.()bP', '$')

State I4 :
('S->tm(.)bP', '$')

State I5 :
('S->tm().bP', '$')

State I6 :
('S->tm()b.P', '$')
('P->.tvoh;Q', '$')

State I7 :
('S->tm()bP.', '$')

State I8 :
('P->t.voh;Q', '$')

State I9 :
('P->tv.oh;Q', '$')

State I10 :
('P->tvo.h;Q', '$')

State I11 :
('P->tvoh.;Q', '$')
```

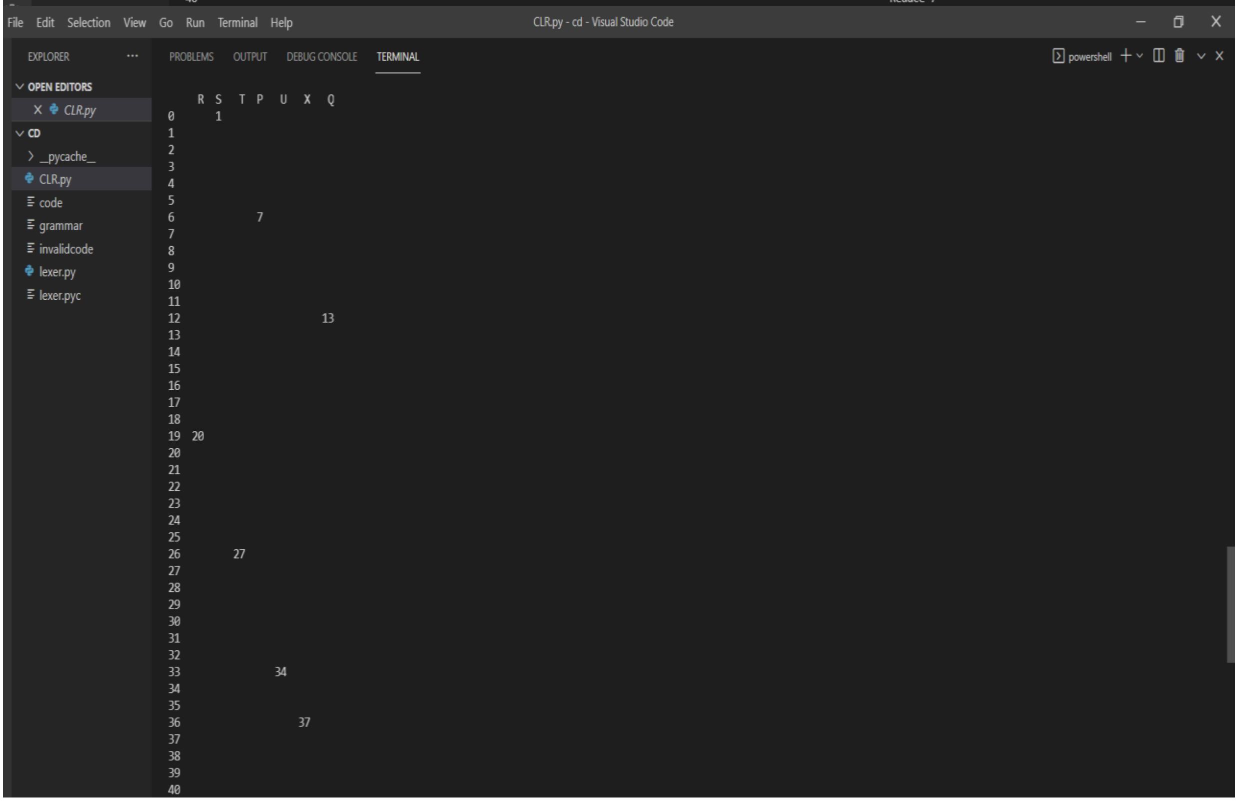
```
State I12 :
('P->tvoh;.Q', '$')
('Q->.w(voh)R', '$')

State I13 :
('P->tvoh;Q.', '$')

State I14 :
('Q->w.(voh)R', '$')

State I15 :
('Q->w(.voh)R', '$')

State I16 :
('Q->w(v.oh)R', '$')

State I17 :
('Q->w(vo.h)R', '$')

State I18 :
('Q->w(voh.)R', '$')

State I19 :
('Q->w(voh).R', '$')
('R->.vovoh;T', '$')

State I20 :
('Q->w(voh)R.', '$')

State I21 :
('R->v.ovoh;T', '$')

State I22 :
('R->vo.voh;T', '$')

State I23 :
('R->vov.oh;T', '$')
```

```
State I24 :
('R->vovo.h;T', '$')

State I25 :
('R->vovoh.;T', '$')

State I26 :
('R->vovoh;.T', '$')
('T->.vovoh;U', '$')

State I27 :
('R->vovoh;T.', '$')

State I28 :
('T->v.ovoh;U', '$')

State I29 :
('T->vo.voh;U', '$')

State I30 :
('T->vov.oh;U', '$')

State I31 :
('T->vovo.h;U', '$')

State I32 :
('T->vovoh.;U', '$')

State I33 :
('T->vovoh;.U', '$')
('U->.ewX', '$')

State I34 :
('T->vovoh;U.', '$')

State I35 :
('U->e.wX', '$')

State I36 :
('U->ew.X', '$')
('X->.rve', '$')

State I37 :
('U->ewX.', '$')

State I38 :
('X->r.ve', '$')

State I39 :
('X->rv.e', '$')

State I40 :
('X->rve.', '$')
```

Parsing Table:

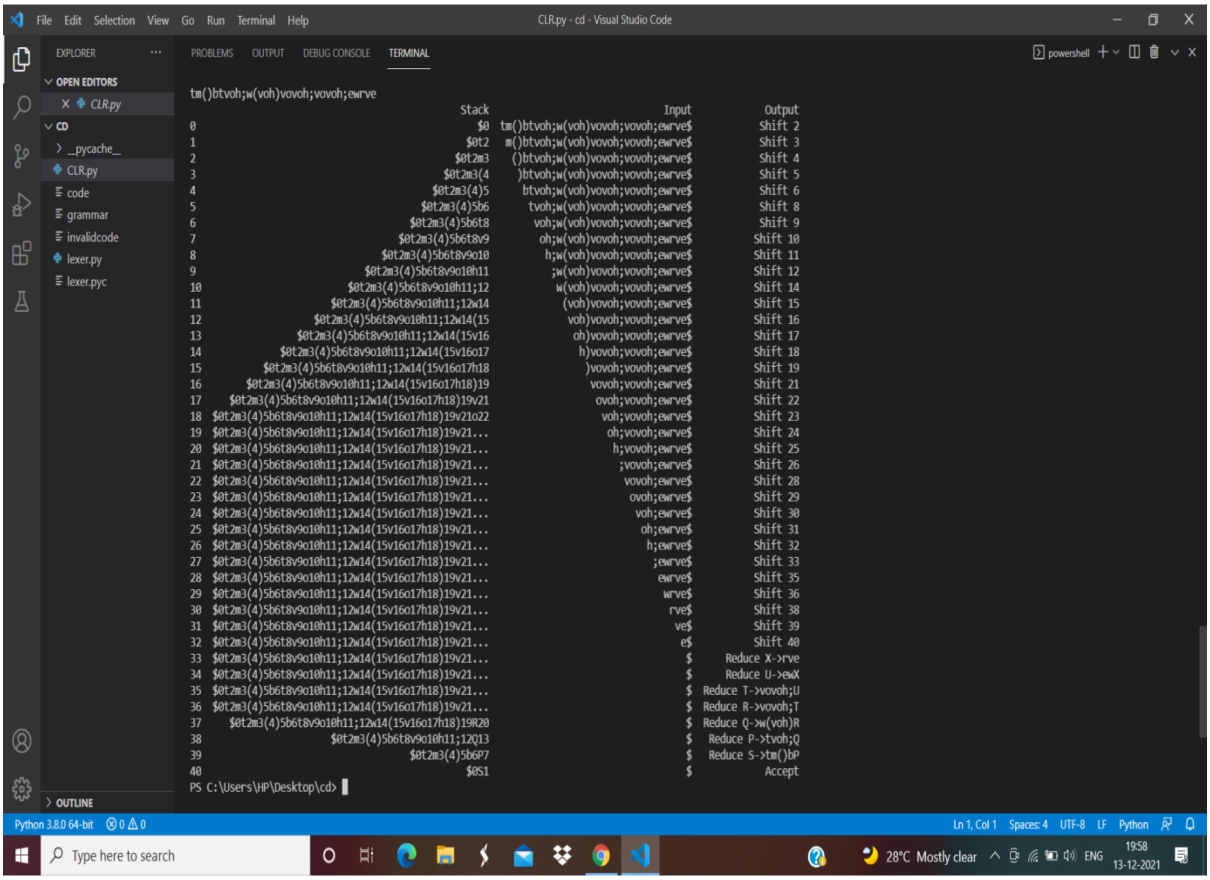


	)	h	v	r	o	b	m	t	(	;	w	e	$
0								Shift 2					
1													Accept
2							Shift 3						
3									Shift 4				
4	Shift 5												
5						Shift 6							
6								Shift 8					
7													Reduce 1
8			Shift 9										
9					Shift 10								
10		Shift 11											
11										Shift 12			
12											Shift 14		
13													Reduce 2
14									Shift 15				
15			Shift 16										
16					Shift 17								
17		Shift 18											
18	Shift 19												
19			Shift 21										
20													Reduce 3
21					Shift 22								
22			Shift 23										
23					Shift 24								
24		Shift 25											
25										Shift 26			
26			Shift 28										
27													Reduce 4
28					Shift 29								
29			Shift 30										
30					Shift 31								
31		Shift 32											
32										Shift 33			
33												Shift 35	
34													Reduce 5
35											Shift 36		
36				Shift 38									
37													Reduce 6
38			Shift 39										
39												Shift 40	
40													Reduce 7



	R	S	T	P	U	X	Q
0		1					
1							
2							
3							
4							
5							
6				7			
7							
8							
9							
10							
11							
12							13
13							
14							
15							
16							
17							
18							
19	20						
20							
21							
22							
23							
24							
25							
26			27				
27							
28							
29							
30							
31							
32							
33					34		
34							
35							
36						37	
37							
38							
39							
40							

Parsing String:



Conclusion

The project mainly aimed at implementing Lexical and Syntax Analyser which are the first two stages in Compiler Analysis Phase. We were successful in doing the same. Similarly, we can also develop remaining stages of the compiler such that our final outcome would be an assembly level language, which can be easily understood by the computer. The two phases of the compiler are separated to increase simplicity, improve the efficiency and increase the portability.

The compiler technology is applied in various computer fields such as HLL implementation, program translation, and computer architecture. In the future, we may experience complex compiler technologies that will be integrated with various computer applications.