NITTE EDUCATION TRUST

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

REPORT ON

COMPILER DESIGN MINI PROJECT

Course Code: 18CS702

Academic Year – 2021-2022

Course Name: **COMPILER DESIGN**

Semester: 7 Section: C

Submitted To,
Course Instructor:

Ms. Asmita Poojary
Assistant Professor-II
Department of CSE,
NMAMIT, Nitte.

Submitted By:

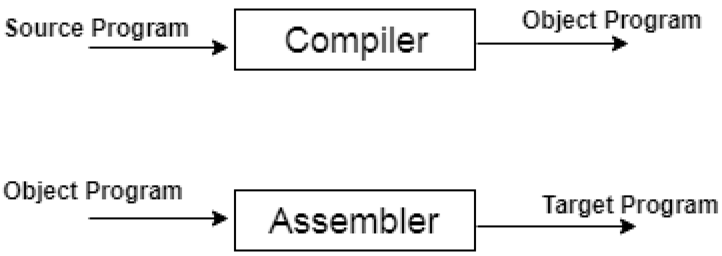
1. USN: 4NM18CS140 **NAME**: S DHRUVA
2. USN: 4NM18CS152 **NAME**: SAMIT D MANVAR

Date of submission: 20-12-2021

TABLE OF CONTENTS

INTRODUCTION

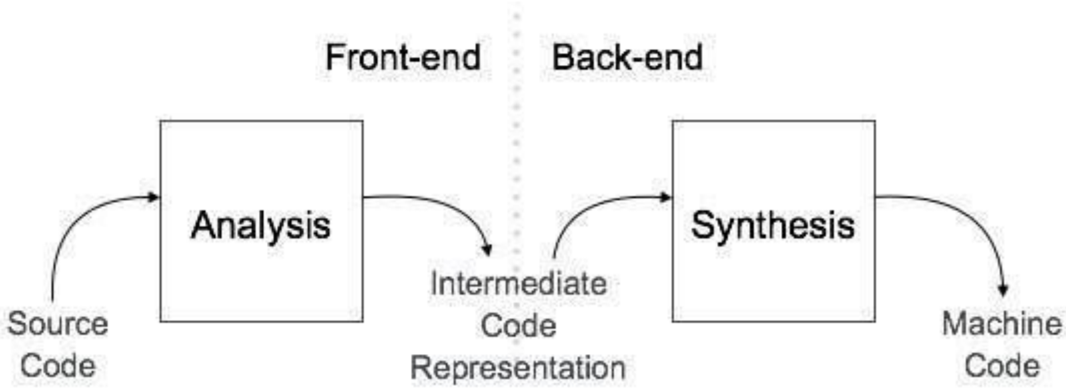
A compiler is a translator that converts the high-level language into the machine language. High-level language is written by a developer and machine language can be understood by the processor. Compiler is used to show errors to the programmer. The main purpose of compiler is to change the code written in one language without changing the meaning of the program. When you execute a program which is written in HLL programming language then it executes into two parts. In the first part, the source program compiled and translated into the object program (low level language). In the second part, object program translated into the target program through the assembler.



A compiler can broadly be divided into two phases based on the way they compile.

Analysis Phase

Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.



Synthesis Phase

Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

Compiler Phases

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

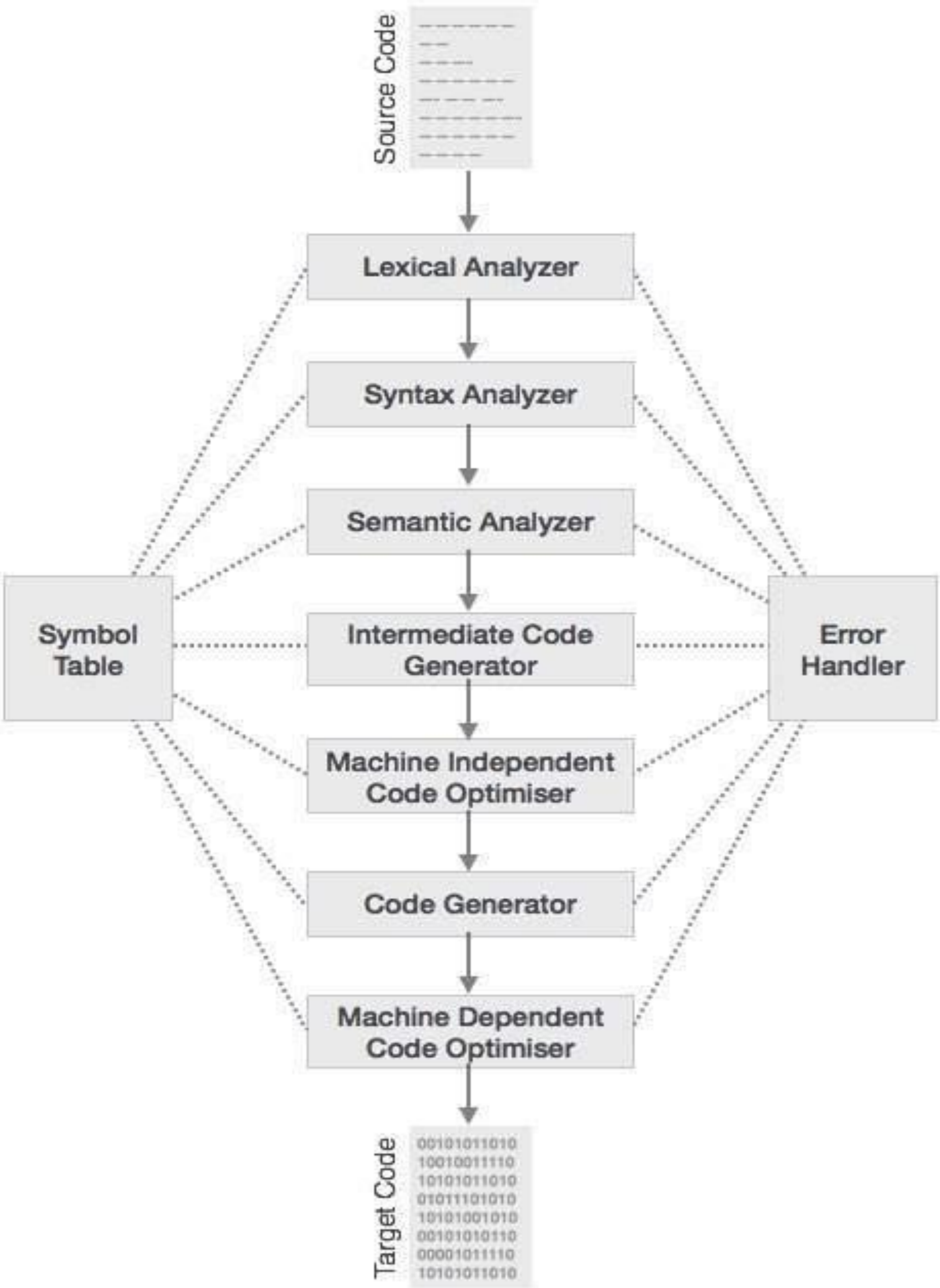


Fig 1: Phases of Compiler

Lexical Analysis:

Lexical analyser phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyser represents these lexemes in the form of tokens.

Syntax Analysis

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

Semantic Analysis

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyser keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

Intermediate Code Generation

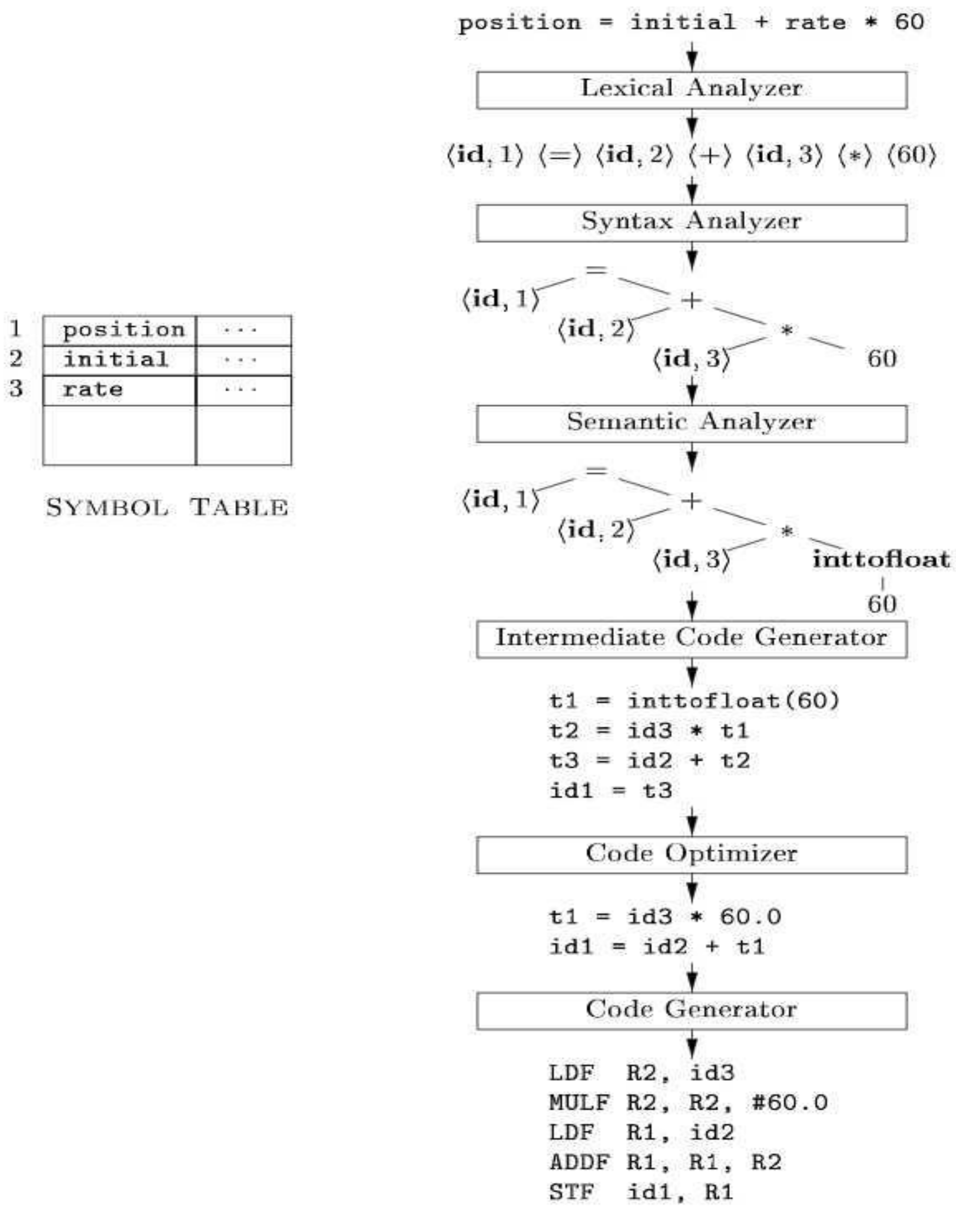
In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

Code Optimization

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

Code Generation

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.



PROBLEM STATEMENT

To design a compiler (Lexical and Parser Phase) for the given hypothetical language
Hypothetical language:

```
int main ()
begin
 int n, i, sum = 0;
   for (i=1; i <= n; ++i)
   begin
     expr= expr+expr;
   end
 End
```

OBJECTIVES

- ❖ To demonstrate the first phase of the compiler – lexical analysis for the given hypothetical language.
- ❖ To demonstrate the working of Parser phase – Syntax analysis.
- ❖ To demonstrate the working of CLR Parser and parse the given string.

METHODOLOGY

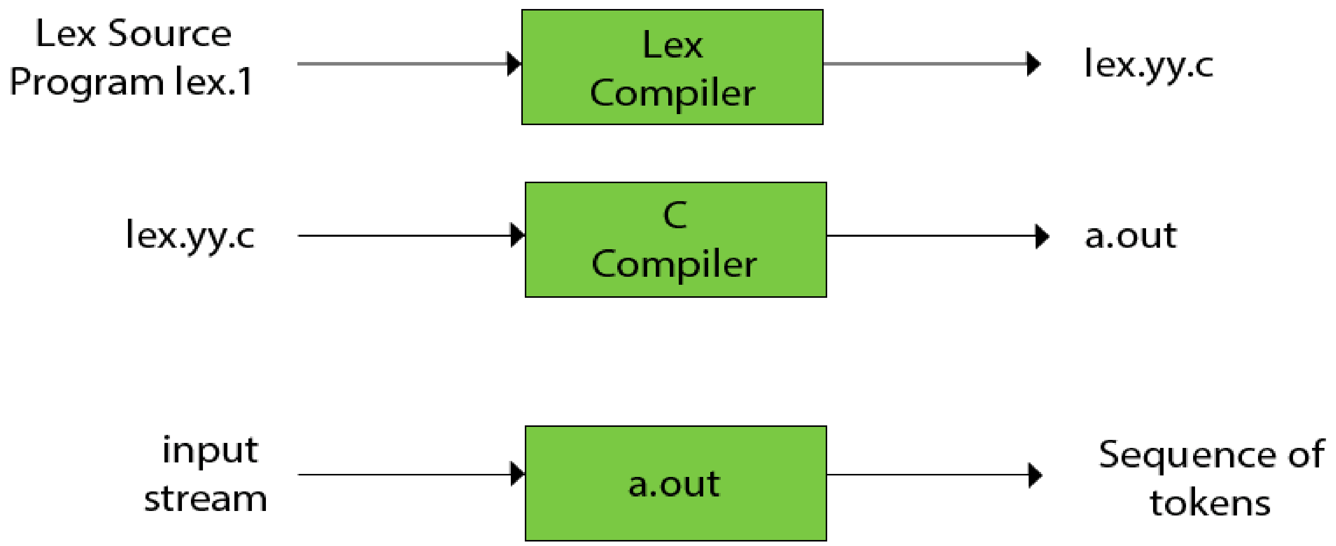
Lexical Analysis: Lexical analysis is the first phase of a compiler. It takes modified source code from language pre-processors that are written in the form of sentences. The lexical analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

In our project we can generate tokens for the given pseudocode with help of lex. A lex is a tool for automatically generating a lexer or scanner given a lex specification.



With help of lex program we are able to generate tokens. The rules defined to identify a valid token is described as pattern.

Lex program is divided into 3 sections

- Global C and Lex declarations section
- Lex rules section
- C code section.

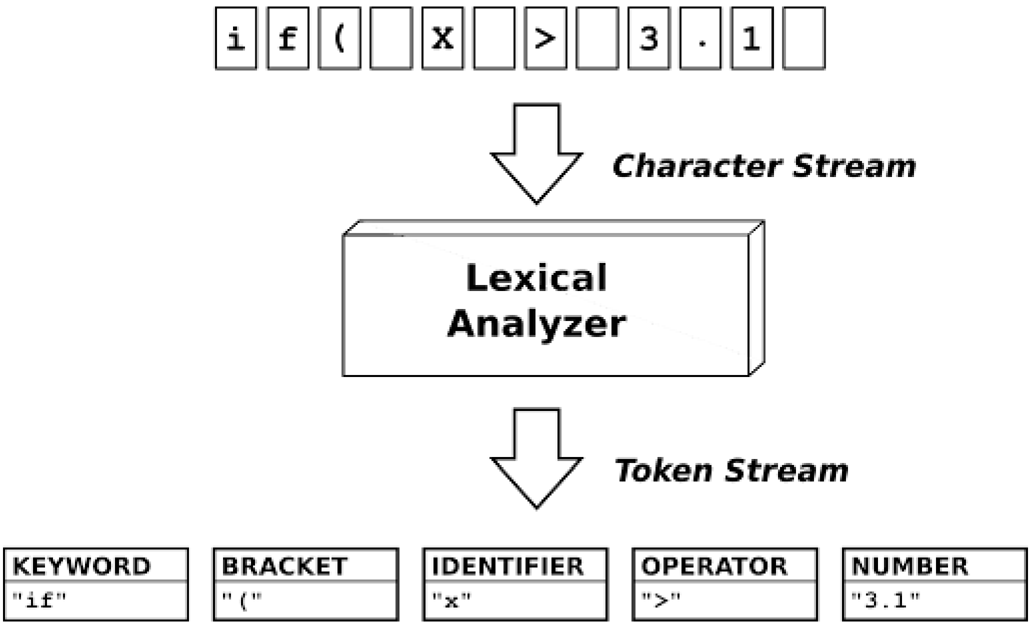
These sections are delimited by %%.

```
... Definition section ...

%%

... Lex rules ...

%%

... C subroutines ...
```

Global C and Lex declarations section: -This is the place to define macros, declare C variables and import header files written in C.

Lex Rules: - Lex rules are of the form **pattern {action}.** E.g., digit [0-9], letter [a-Za-z] etc.

C subroutines: -The C code section contains C statements and functions.



Parser Phase: - Parser phase is implemented using python program. Here we are going to implement CLR Parser for parsing the string.

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the greater number of states as compare to the SLR (1) parsing. LR (1) item is a collection of LR (0) items and a look ahead symbol. The look ahead is used to determine that where we place the final item. The look ahead always adds $ symbol for the argument production

LR (1) item = LR (0) item + look ahead

IMPLEMENTATION

Implementation of Lexical Analyser: - To generate tokens from given input (pseudocode) we use lex. We have defined patterns and corresponding actions associated with it. For example, when a pattern named keyword is matched corresponding action – printing the keyword takes place. In this way we generate tokens. For each token – keywords, identifiers, special symbols, constants, operators we have defined different lex rules (regular definition). Some important lex functions we need to remember.

- **yylex ()**: - The function that starts the analysis. It is automatically generated by Lex. (This is the entry point to LEX)
- **yywrap ():** - This function is called by LEX when end of file (or input) is encountered. If this function returns 1, the parsing stops.
- **yyin** of the type FILE*. This points to the current file being parsed by the lexer. (Input file).

```
%{
#include<stdlib.h>
%}
digit [0-9]
constants {digit}+
text [A-Za-z]
keywords "for"|"while"|"do"|"int"|"float"|"double"|"long"|"void"|"main"|"begin"|"end"|"End"
special_char ";"|"("|")"|"()"|","
identifier {text}({digit}|{text}|"_")*
operators "+"|"-"|"/"|"%"|"*"|"&&"|"||"|"!"|"!="|"<"|">"|"<="|">="|"=="|"++"|"--"|"="
%%
[\n]+ ;

{keywords}  {printf("%s\t==> keyword            |\n",yytext);}

{identifier}  {printf("%s\t==> identifier          |\n",yytext);}

{operators}  {printf("%s\t==> operators           |\n",yytext);}

{special_char}  {printf("%s\t==> special symbols     |\n",yytext);}

{constants} {printf("%s\t==> constant            |\n",yytext);}

. ;
%%
int yywrap()
{
        return 1;
}

int main(int argc,char* argv[])
{

printf("---------------------------------\n");
printf("STRING\t      TOKEN              | \n");
printf("---------------------------------\n");

        yyin=fopen(argv[1],"r");
        yylex();
        fclose(yyin);
printf("---------------------------------\n");
}
```

Fig 3: Generating tokens using lex

- First, we have declared pre-processor directives (header files) in global c declaration section.
- Next, we have defined the regular definition for corresponding tokens. For example, a keyword includes for, int, void, main, while, break etc. Similarly, it is defined for other tokens as well.
- We have defined pattern {action} for each token. When a string is identifier is found it prints the corresponding string as an identifier.
- We declare yywrap () and in the main () function we have c subroutines. Here we declare yyin to take input from file and also declare yylex () function. The input is read from a file name input.txt with help of yyin. We are opening the file in read mode

Steps to run the lex program to obtain tokens

1. Create a lex file with .l as extension. Here our lex file is token.l.
2. The input file which is the input to lex is input.txt. The input file contains the pseudocode for which we want to generate tokens
3. To run the lex program open terminal and type **lex token.l** and press Enter key.
4. Next, type **cc lex.yy.c** and press the Enter Key.
5. Finally type **./a.out input.txt** and press the enter key. The output is shown below.

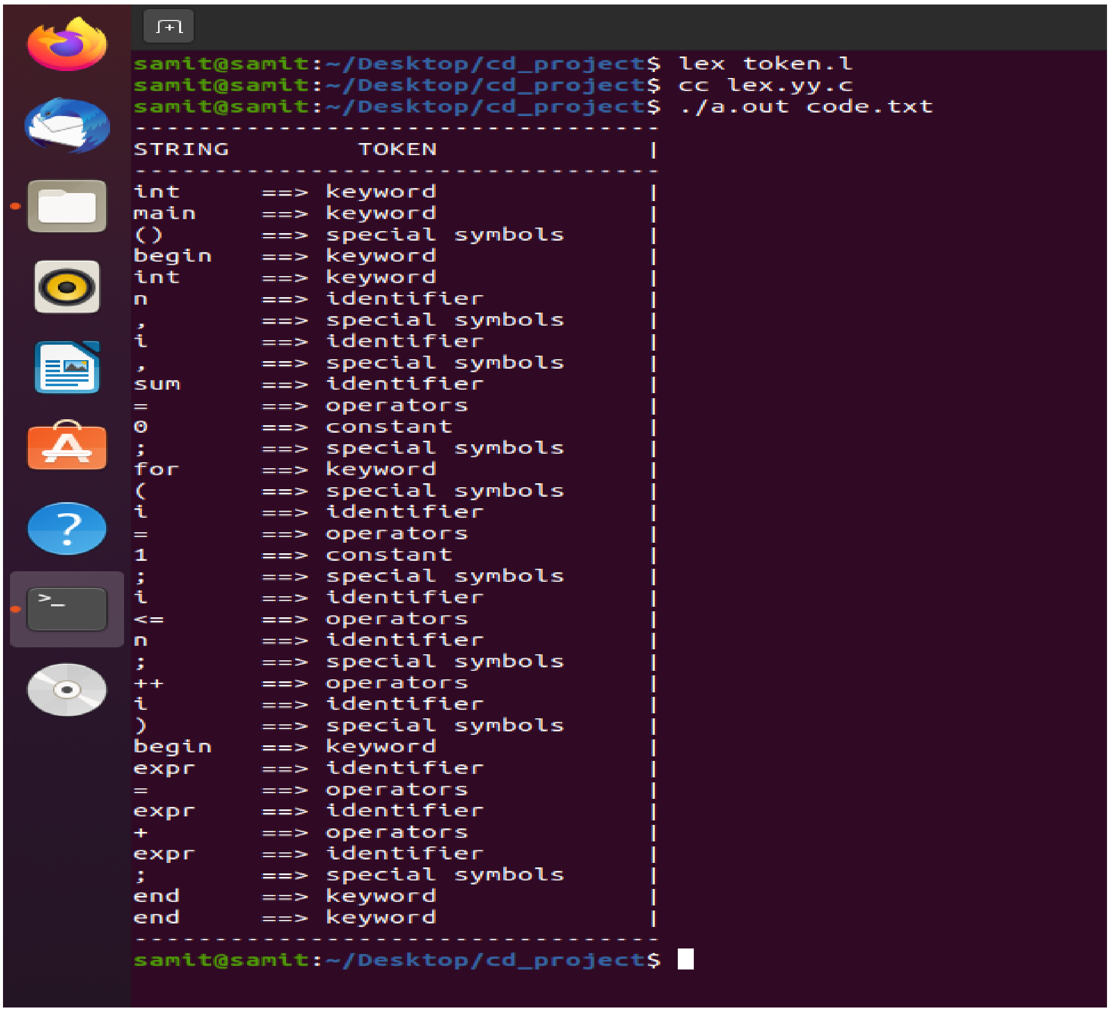


Fig 4 : Output of Lexical Analyser (Generated Tokens)

Implementation of Parser Phase: - Parser phase is implemented by constructing a CLR Parser with help of python. CLR Algorithm is implemented using python 3.

For constructing a CLR Parser we need to follow certain steps

- For the given input string write a context free grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Construct a CLR (1) parsing table

Context Free Grammar: - Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

G describes the grammar
T describes a finite set of terminal symbols.
V describes a finite set of non-terminal symbols
P describes a set of production rules
S is the start symbol.

For our project we have defined production rules or context free grammar for the given pseudocode. The grammar is:

S->im()A
A->biv,v,vod;B
B->f(vod;vov;oov)C
C->beoeoe;yz

Where S, A, B, C are non-terminals.
Terminal symbols are i, m, b, v, o, d, f, b, e, y, z where

- i stands for int
- m stands for main
- b stands for begin
- f stands for for
- v stands for identifier
- o stands for operators
- e stands for expr
- y stands for end
- Z stands for End

Add augmented production in given grammar: - Before we start generating parse table, we need to add augmented production for example S’ -> S etc. For our pseudocode we have added S’ -> S .

S’ -> S //Augmented grammar

S -> im()A

A ->biv,v,vod;B

B ->f(vod;vov;oov)C

C ->beoeoe;yz

Then we create CLR Parser code with help of Python 3 and finally create parsing table and parse the string from CLR Parser code written in python.

In CLR algorithm implemented, we have already defined the file name to be taken as input for production rules and for parsing string. The file grammarrules contains the production rules and the pseudocode contains the string/code to be parsed.

To run the clr parser: -In the command prompt type clrparsing.py. This file has the clr algorithm. Press Enter and the Parser table is generated. Also, the string is parsed and the result is displayed.

Code Snippets for implementing CLR Parser Algorithm

*C:\Users\SAMIT D MANVAR\Desktop\cd\clrparsing.py - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

clrparsing.py

```python
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
```

```python


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
```

clrparsing.py

```python
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
```

```python
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)

```

```python
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

```

*C:\Users\SAMIT D MANVAR\Desktop\cd\clrparsing.py - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

clrparsing.py

```python
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]

```

*C:\Users\SAMIT D MANVAR\Desktop\cd\clrparsing.py - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

clrparsing.py

```python
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

```

*C:\Users\SAMIT D MANVAR\Desktop\cd\clrparsing.py - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

clrparsing.py

```python
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()
```

```python
            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
```

*C:\Users\SAMIT D MANVAR\Desktop\cd\clrparsing.py - Notepad++

```python

if __name__ == "__main__":
    grammar = get_grammar("grammarrules")
    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("wrong"))
    except:
        pass
    if string!=None:
        print("the string to be parsed is :- ", string)
        print("b stands for begin")
        print("f stands for for")
        print("i stands for int")
        print("m stands for main")
        print("y stands for end")
        print("z stands for End")
        print("b stands for begin")
        print("v stands for identifier")
        print("o stands for operators")
        print("d stands for digit")
        print("e stands for expr")
    try:
        PARSE_TABLE = parse_string(string, ACTION, GOTO)
        print(PARSE_TABLE)
    except:
        print('Invalid input!!')

```

RESULT

The CLR Parsing algorithm applied to the given pseudocode or Hypothetical language produces 41 states – I0 TO I40. The string is successfully parsed as shown below.

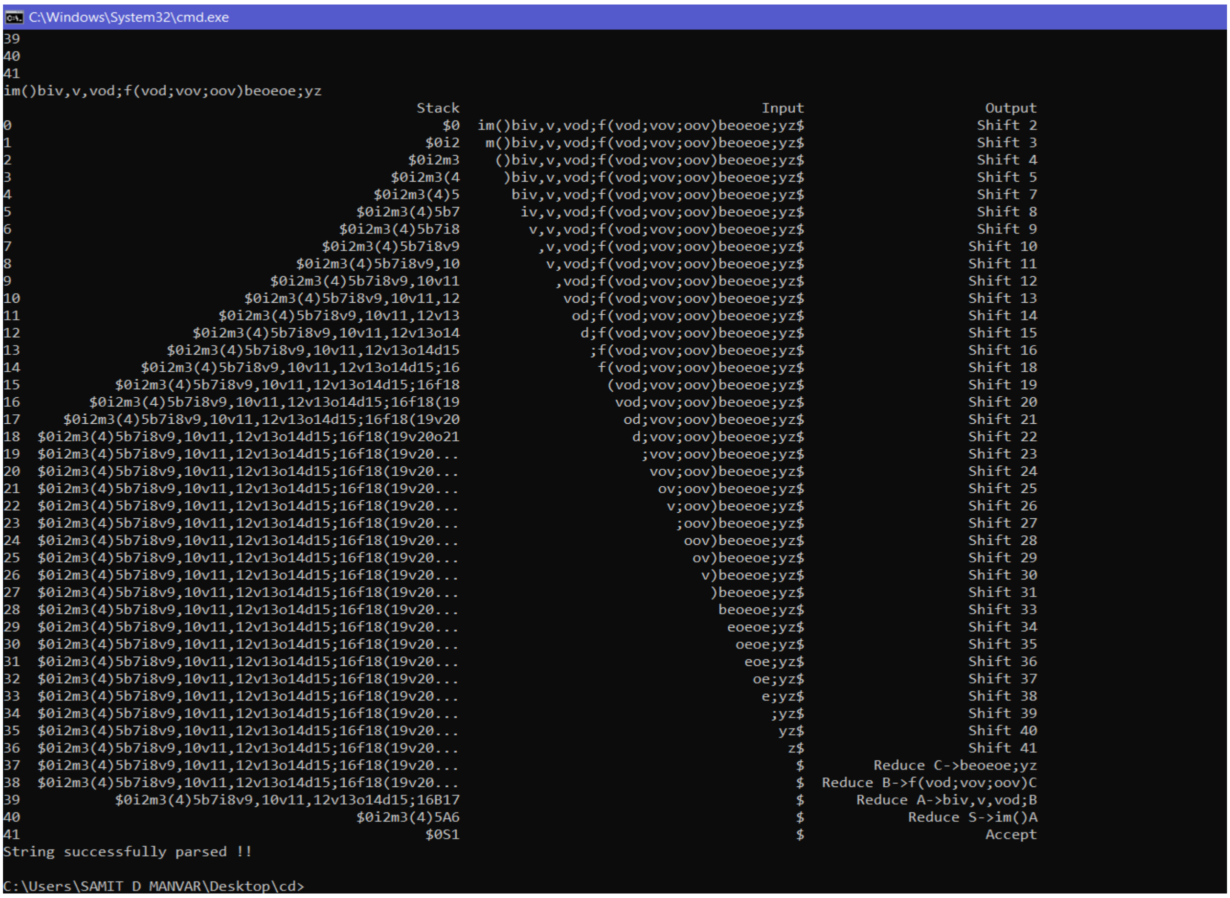
C:\Windows\System32\cmd.exe

	b	(	i	z	;	f	m	v	$	d	,	o	e	y	)
0			Shift 2												
1									Accept						
2							Shift 3								
3		Shift 4													
4															Shift 5
5	Shift 7														
6									Reduce 1						
7			Shift 8												
8								Shift 9							
9											Shift 10				
10								Shift 11							
11											Shift 12				
12								Shift 13							
13												Shift 14			
14										Shift 15					
15					Shift 16										
16						Shift 18									
17									Reduce 2						
18		Shift 19													
19								Shift 20							
20												Shift 21			
21										Shift 22					
22					Shift 23										
23								Shift 24							
24												Shift 25			
25								Shift 26							
26					Shift 27										
27												Shift 28			
28												Shift 29			
29								Shift 30							
30															Shift 31
31	Shift 33														
32									Reduce 3						
33													Shift 34		
34												Shift 35			
35													Shift 36		
36												Shift 37			
37													Shift 38		
38					Shift 39										
39														Shift 40	
40				Shift 41											
41									Reduce 4						

	A	B	C	S
0				1
1				
2				
3				
4				

C:\Windows\System32\cmd.exe

```
C:\Users\SAMIT D MANVAR\Desktop\cd>clrparsing.py
State I0 :
('P->.S', '$')
('S->.im()A', '$')

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

State I2 :
('S->i.m()A', '$')

State I3 :
('S->im.()A', '$')

State I4 :
('S->im(.)A', '$')

State I5 :
('S->im().A', '$')
('A->.biv,v,vod;B', '$')

State I6 :
('S->im()A.', '$')

State I7 :
('A->b.iv,v,vod;B', '$')

State I8 :
('A->bi.v,v,vod;B', '$')

State I9 :
('A->biv.,v,vod;B', '$')

State I10 :
('A->biv,.v,vod;B', '$')

State I11 :
('A->biv,v.,vod;B', '$')

State I12 :
('A->biv,v,.vod;B', '$')

State I13 :
('A->biv,v,v.od;B', '$')

State I14 :
('A->biv,v,vo.d;B', '$')

State I15 :
```



```
C:\Windows\System32\cmd.exe
39
40
41
im()biv,v,vod;f(vod;vov;oov)beoeoe;yz
                                                      Stack                                          Input              Output
0                                                        $0  im()biv,v,vod;f(vod;vov;oov)beoeoe;yz$             Shift 2
1                                                      $0i2   m()biv,v,vod;f(vod;vov;oov)beoeoe;yz$             Shift 3
2                                                    $0i2m3    ()biv,v,vod;f(vod;vov;oov)beoeoe;yz$             Shift 4
3                                                  $0i2m3(4     )biv,v,vod;f(vod;vov;oov)beoeoe;yz$             Shift 5
4                                                $0i2m3(4)5      biv,v,vod;f(vod;vov;oov)beoeoe;yz$             Shift 7
5                                              $0i2m3(4)5b7       iv,v,vod;f(vod;vov;oov)beoeoe;yz$             Shift 8
6                                            $0i2m3(4)5b7i8        v,v,vod;f(vod;vov;oov)beoeoe;yz$             Shift 9
7                                          $0i2m3(4)5b7i8v9         ,v,vod;f(vod;vov;oov)beoeoe;yz$            Shift 10
8                                       $0i2m3(4)5b7i8v9,10          v,vod;f(vod;vov;oov)beoeoe;yz$            Shift 11
9                                    $0i2m3(4)5b7i8v9,10v11           ,vod;f(vod;vov;oov)beoeoe;yz$            Shift 12
10                                $0i2m3(4)5b7i8v9,10v11,12            vod;f(vod;vov;oov)beoeoe;yz$            Shift 13
11                             $0i2m3(4)5b7i8v9,10v11,12v13             od;f(vod;vov;oov)beoeoe;yz$            Shift 14
12                          $0i2m3(4)5b7i8v9,10v11,12v13o14              d;f(vod;vov;oov)beoeoe;yz$            Shift 15
13                       $0i2m3(4)5b7i8v9,10v11,12v13o14d15               ;f(vod;vov;oov)beoeoe;yz$            Shift 16
14                    $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16                f(vod;vov;oov)beoeoe;yz$            Shift 18
15                 $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18                 (vod;vov;oov)beoeoe;yz$            Shift 19
16              $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19                  vod;vov;oov)beoeoe;yz$            Shift 20
17           $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20                   od;vov;oov)beoeoe;yz$            Shift 21
18   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20o21                    d;vov;oov)beoeoe;yz$            Shift 22
19   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                    ;vov;oov)beoeoe;yz$            Shift 23
20   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                     vov;oov)beoeoe;yz$            Shift 24
21   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                      ov;oov)beoeoe;yz$            Shift 25
22   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                       v;oov)beoeoe;yz$            Shift 26
23   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                        ;oov)beoeoe;yz$            Shift 27
24   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                         oov)beoeoe;yz$            Shift 28
25   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                          ov)beoeoe;yz$            Shift 29
26   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                           v)beoeoe;yz$            Shift 30
27   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                            )beoeoe;yz$            Shift 31
28   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                             beoeoe;yz$            Shift 33
29   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                              eoeoe;yz$            Shift 34
30   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                               oeoe;yz$            Shift 35
31   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                                eoe;yz$            Shift 36
32   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                                 oe;yz$            Shift 37
33   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                                  e;yz$            Shift 38
34   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                                   ;yz$            Shift 39
35   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                                    yz$            Shift 40
36   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                                     z$            Shift 41
37   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                                     $   Reduce C->beoeoe;yz
38   $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16f18(19v20...                                     $ Reduce B->f(vod;vov;oov)C
39                 $0i2m3(4)5b7i8v9,10v11,12v13o14d15;16B17                                 $    Reduce A->biv,v,vod;B
40                                              $0i2m3(4)5A6                                 $        Reduce S->im()A
41                                                      $0S1                                 $                 Accept
String successfully parsed !!

C:\Users\SAMIT D MANVAR\Desktop\cd>
```

CONCLUSIONS AND FUTURE WORK

Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

In our mini project we have successfully implemented first two phases of compiler – Lexical analyser and Parser Phase (Syntax analyser). We have successfully generated tokens in lexical phase and successfully implemented CLR Parser to generate parsing table and parse the string using python. Future works can include implementing other phases of compiler such as sematic analysis, intermediate code generator, code optimization, code generation.

REFERENCES

[1] https://www.javatpoint.com/compiler-tutorial

[2] https://www.tutorialspoint.com/compiler_design/index.htm

[3] https://www.geeksforgeeks.org/compiler-design-tutorials/