ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of people who made it possible because “Success is the abstract of hard work and perseverance, but steadfast of all is encouraging guidance.” So I acknowledge all those whose guidance and encouragement served as a beacon light and crowned my efforts with success.

I would like to thank our principal **Prof. Niranjan N. Chiplunkar** firstly, for providing us with this unique opportunity to do the mini project in the 7th semester of Computer Science and Engineering.

I would like to thank my college administration for providing a conducive environment and also suitable facilities for this mini project. I would like to thank our HOD **Dr. Jyothi Shetty** for showing me the path and providing the inspiration required for taking the project to its completion. It is my great pleasure to thank my mentor **Ms. Anusha Anchan** for her continuous encouragement, guidance, and support throughout this project.

Finally, thanks to staff members of the department of CSE, my parents, and friends for their honest opinions and suggestions throughout the course of our mini-project

Nischitha Shetty (4NM20CS122)

Prathiksha Kini (4NM20CS139)

ABSTRACT

The main objective of this mini project is to implement the first two phases of a compiler, which are lexical analysis and syntax analysis, for a given piece of code. In the lexical analysis phase, the input will be source code or program. The output should be in the form of tokens. This phase removes spaces, new lines, and comment lines from the program. This phase is also known as the tokenizer.

We generate the parse table which has entries for each terminal and non-terminals identified in them. Before the generation of the parse table, we identified the FIRST and FOLLOW", of each terminal using a recursive method. The final stage is the parsing which is done by using the standard LL(1) parsing steps.

The outcome of the project is to identify the parsing actions taken by the grammar for proper and invalid source code. Along with this, we are generating the parse table for the given input.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

A compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

PHASES OF A COMPILER:

If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in the figure below.

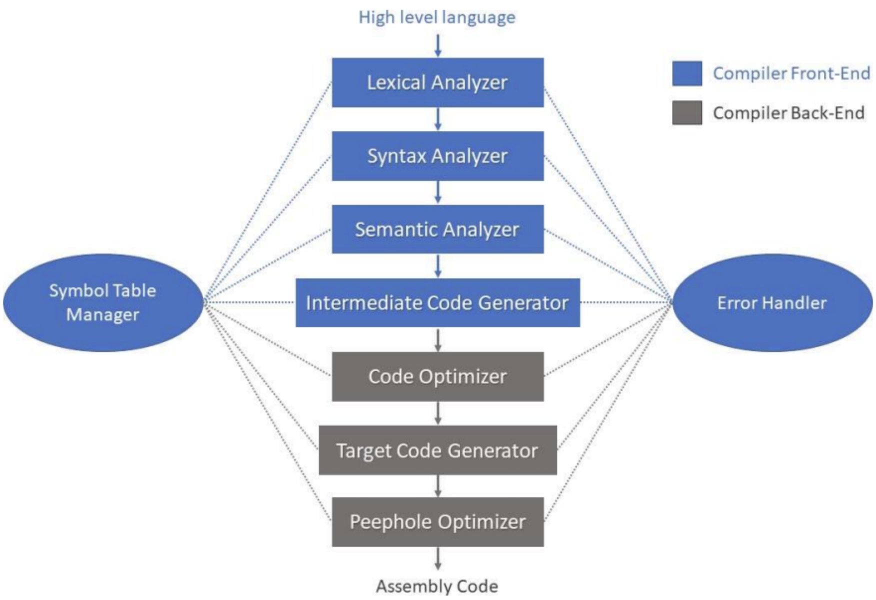

Fig 1: Phases of Compiler

ANALYSIS PART:

This part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill-formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

SYNTHESIS PART:

This part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.

Lexical Analyser

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form

(token-name, attribute-value)

Syntax Analyzer

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. Code Generation The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

Symbol Table

An essential function of a compiler is to record the variable names used in the source program and collect information about the various attributes of each name.

2. DESIGN

2.1 LEXICAL ANALYSER:

Lexical Analysis is the first phase when the compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens. Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to the next phase.

The primary functions of this phase are:

● Identify the lexical units in a source code

● Classify lexical units into classes like constants, and reserved words, and enter them in different tables. It will Ignore comments in the source program

● Identify a token which is not a part of the language

TOKENS: The token is a sequence of characters that represents a unit of information in the source program

Problem Statement:

```
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
```

Fig 2: Input String to be parsed

2.2 SYNTAX ANALYZER

Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code written by the programmer is correct or not. Syntax analysis is based on the rules of the specific programming language by constructing the parse tree with the help of tokens. It also determines the structure of the source language and the grammar or syntax of the language. Here, is a list of tasks performed in this phase:

- Obtain tokens from the lexical analyzer
- Checks if the expression is syntactically correct or not
- Report all syntax errors
- Construct a hierarchical structure which is known as a parse tree

LL(1) Parser: LL(1) parsing is a top-down parsing technique. Thus, to use this technique we must eliminate the ambiguity of the grammar. After removing the ambiguity, we must eliminate left recursion and left factoring .

LL(1) grammar for the above problem statement is:

```
Rules:
S -> [['T', 'M', 'B', 'A', 'D']]
T -> [['int']]
M -> [['main()']]
B -> [['begin']]
D -> [['End']]
A -> [['E', 'F', 'G', 'W', 'X']]
E -> [['T', 'L[10]', ';']]
F -> [['T', 'maxval', '=', 'L[0]', ';']]
G -> [['for', 'C', 'do']]
C -> [['i', '=', '1', 'to', 'n', '-', '1']]
W -> [['if', 'L[i]', '>', 'maxval', 'P', 'Q', 'R']]
P -> [['maxval', '=', 'L[i]', ';']]
Q -> [['endif']]
R -> [['endfor']]
X -> [['return', '(', 'maxval', ')']]
```

Fig 3: Rules of LL(1) grammar

Algorithm to eliminate left recursion:

If a grammar is of the form A->A@|B

Then A->BA'

A'->@A'| E

Algorithm to eliminate left factoring:

If a grammar is of the form A->@X|@Y|…..|k|l

Then A->@A'|k|l

A'->X|Y|…

Construction of LL(1) Parsing Table Algorithm:

Input: Grammar G

Output: Parsing table M

Method: For each production A->@ of the grammar, do the following, 1. For each terminal a in FIRST(@), add A-> @ to M[A, a]

2. If e is in FIRST(@) then for each then for each terminal b in FOLLOW(A) add A->@ to M[A, b]. If e is in FIRST(@) and $ is in FOLLOW(A) add A->@ to M[A,$] as well

If after performing the above, there is no production at all in M[A,a], then set M[A, a] to error

Calculating First and Follow:

```
Calculated firsts:
first(S) => {'int'}
first(T) => {'int'}
first(M) => {'main()'}
first(B) => {'begin'}
first(D) => {'End'}
first(A) => {'int'}
first(E) => {'int'}
first(F) => {'int'}
first(G) => {'for'}
first(C) => {'i'}
first(W) => {'if'}
first(P) => {'maxval'}
first(Q) => {'endif'}
first(R) => {'endfor'}
first(X) => {'return'}

Calculated follows:
follow(S) => {'$'}
follow(T) => {'maxval', 'main()', 'L[10]'}
follow(M) => {'begin'}
follow(B) => {'int'}
follow(D) => {'$'}
follow(A) => {'End'}
follow(E) => {'int'}
follow(F) => {'for'}
follow(G) => {'if'}
follow(C) => {'do'}
follow(W) => {'return'}
follow(P) => {'endif'}
follow(Q) => {'endfor'}
follow(R) => {'return'}
follow(X) => {'End'}
```

Fig 4: First and Follow Sets

3. IMPLEMENTATION

```python
import re
import warnings
from tabulate import tabulate
import copy

warnings.filterwarnings("ignore")
```

Fig 5: Standard Library Imports

```python
def removeLeftRecursion(rulesDiction):
    store = {}
    for lhs in rulesDiction:
        # alphaRules will store the rules with left recursion.
        # betaRules will store the rules without left recursion.
        # allrhs is the list of right-hand sides for the current non-terminal lhs
        alphaRules = []
        betaRules = []
        allrhs = rulesDiction[lhs]

        # Separate into 2 groups those with left recursion and those without
        for subrhs in allrhs:
            if subrhs[0] == lhs:
                alphaRules.append(subrhs[1:])
            else:
                betaRules.append(subrhs)

        ''' If there are rules with left recursion (alphaRules is not empty), it creates a new non-terminal symbol (lhs_)
        to replace the left-recursive rules. The loop ensures that the new symbol doesn't already exist in the original
        grammar or in the temporary storage (store).'''

        if len(alphaRules) != 0:
            lhs_ = lhs + "'"
            while lhs_ in rulesDiction.keys() or lhs_ in store.keys():
                lhs_ += "'"

            # For each rule in betaRules, it appends the new non-terminal lhs_
            # to the end of the rule and updates the original non-terminal's rules with the modified betaRules.

            for b in range(0, len(betaRules)):
                betaRules[b].append(lhs_)
            rulesDiction[lhs] = betaRules

            for a in range(0, len(alphaRules)):
                alphaRules[a].append(lhs_)
            alphaRules.append(['#'])
            store[lhs_] = alphaRules
    for left in store:
        # Result of left recursion will be stored in temp storage store
        rulesDiction[left] = store[left]
    return rulesDiction
```

Fig 6: Left Recursion Removal

Comment Code

```python
def LeftFactoring(rulesDiction):
    # This dictionary will store the left-factored grammar rules.
    newDict = {}
    for lhs in rulesDiction:
        # Group right-hand sides (rhs) based on the first terminal/non-terminal in each:
        allrhs = rulesDiction[lhs]
        temp = dict()
        for subrhs in allrhs:
            if subrhs[0] not in list(temp.keys()):
                temp[subrhs[0]] = [subrhs]
            else:
                temp[subrhs[0]].append(subrhs)
        # Process each group:
        new_rule = []
        tempo_dict = {}
        for term_key in temp:
            allStartingWithTermKey = temp[term_key]
            # If a group has more than one rule, perform left factoring:
            if len(allStartingWithTermKey) > 1:
                lhs_ = lhs + "'"
                while lhs_ in rulesDiction.keys() or lhs_ in tempo_dict.keys():
                    lhs_ += "'"
                new_rule.append([term_key, lhs_])
                ex_rules = []
                for g in temp[term_key]:
                    ex_rules.append(g[1:])
                tempo_dict[lhs_] = ex_rules
            # If a group has only one rule, keep it unchanged:
            else:
                new_rule.append(allStartingWithTermKey[0])
        # Update the newDict with the left-factored rules:
        newDict[lhs] = new_rule
        for key in tempo_dict:
            newDict[key] = tempo_dict[key]
    return newDict
```

Fig 7: Left Factoring Removal

```python
def first(rule):
    global rules, nonterm_userdef, term_userdef, diction, firsts
    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return rule[0]
        elif rule[0] == '#': # For epsilon
            return '#'

    # If the first symbol is a non-terminal, recursively calculate the FIRST set for the corresponding
    # right-hand side rules in the grammar.
    if len(rule) != 0:
            if rule[0] in list(diction.keys()):
                fres = []
                rhs_rules = diction[rule[0]]
                for itr in rhs_rules:
                    indivRes = first(itr)
                    if type(indivRes) is list:
                        for i in indivRes:
                            fres.append(i)
                    else:
                        fres.append(indivRes)

                '''
                If the FIRST set of the non-terminal contains epsilon ('#'), remove it from the set.
                If the remaining symbols in the rule can derive epsilon, add epsilon back to the set.
                '''

                if '#' not in fres:
                        return fres
                else:
                        newList = []
                        fres.remove('#')
                        if len(rule) > 1:
                            ansNew = first(rule[1:])
                            if ansNew != None:
                                if type(ansNew) is list:
                                    newList = fres + ansNew
                                else:
                                    newList = fres + [ansNew]
                            else:
                                newList = fres
                        fres.append('#')
                        return newList
```

Fig 8: Function for creating FIRST

```python
def follow(nt):
    global start_symbol, rules, nonterm_userdef, term_userdef, diction, firsts, follows

    # The solset set will store the symbols in the FOLLOW set for the given non-terminal.
    solset = set()

    # Handling the Start Symbol:
    if nt == start_symbol:
        solset.add('$')

    # Iterating Over Non-terminals and Production Rules:
    for curNT in diction:
        rhs = diction[curNT]
        for subrule in rhs:
            # Finding the Occurrences of the Target Non-terminal in a Rule:
            if nt in subrule:
                while nt in subrule:
                    index_nt = subrule.index(nt)
                    subrule = subrule[index_nt + 1:]
                    # Handling Symbols Following the Target Non-terminal:
                    if len(subrule) != 0:
                     res = first(subrule)
                     # Handling Epsilon Transitions in FIRST set
                     if '#' in res:
                        newList = []
                        res.remove('#')
                        ansNew = follow(curNT)
                        if ansNew != None:
                            if type(ansNew) is list:
                                newList = res + ansNew
                            else:
                                newList = res + [ansNew]
                        else:
                            newList = res
                        res = newList

                    else:
                     if nt != curNT:
                        res = follow(curNT)

                    # Adding Symbols to solset
                    if res is not None:
                            if type(res) is list:
                                for g in res:
                                    solset.add(g)
                            else:
                                solset.add(res)

    return list(solset)
```

Fig 9: Function for creating FOLLOW

```python
def computeAllFirsts():
    global rules, nonterm_userdef, term_userdef, diction, firsts
    for rule in rules:
        k = rule.split("->")
        k[0] = k[0].strip()
        k[1] = k[1].strip()
        rhs = k[1]
        multirhs = rhs.split('|')
        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()
        diction[k[0]] = multirhs
    # Open a file named rules.txt in write mode
    with open('rules.txt', 'w') as file:
        file.write("Rules:\n")
        for y in diction:
            file.write(f"{y} -> {diction[y]}\n")

    # Remove left recursion
    diction = removeLeftRecursion(diction)
    # Remove left factoring
    diction = LeftFactoring(diction)
    for y in list(diction.keys()):
        t = set()
        for sub in diction.get(y):
            res = first(sub)
            if res != None:
                if type(res) is list:
                    for u in res:
                        t.add(u)
                else:
                    t.add(res)
        firsts[y] = t

    print("============================================================================================")
    print("\nCalculated firsts: ")
    key_list = list(firsts.keys())
    index = 0
    for gg in firsts:
     print(f"first({key_list[index]}) "f"=> {firsts.get(gg)}")
     index += 1
```

Fig 10: Function to compute and store all FIRSTS

```python
def computeAllFollows():
    global start_symbol, rules, nonterm_userdef, term_userdef, diction, firsts, follows
    for NT in diction:
        solset = set()
        sol = follow(NT)
        if sol is not None:
            for g in sol:
                solset.add(g)
        follows[NT] = solset
    print("\nCalculated follows: ")
    key_list = list(follows.keys())
    index = 0
    for gg in follows:
     print(f"follow({key_list[index]})"f" => {follows[gg]}")
     index += 1
```

Fig 11: Function to compute and store all FOLLOWS

```python
def createParseTable():
    global diction, firsts, follows, term_userdef
    print("\n")
    print("===========================================================================================")
    print("Firsts and Follow Result table")
    # Printing FIRST and FOLLOW Sets
    mx_len_first = 0
    mx_len_fol = 0
    for u in diction:
        k1 = len(str(firsts[u]))
        k2 = len(str(follows[u]))
        if k1 > mx_len_first:
            mx_len_first = k1
        if k2 > mx_len_fol:
            mx_len_fol = k2
    print(tabulate([["Non-T", "FIRST", "FOLLOW"]] + [[u, str(firsts[u]),
                    str(follows[u])] for u in diction], headers='firstrow', tablefmt='fancy_grid'))

    print("\n")
    print("===========================================================================================")

    ntlist = list(diction.keys())
    terminals = copy.deepcopy(term_userdef)
    terminals.remove('(')
    terminals.remove(')')
    terminals.remove('+')
    terminals.remove('-')
    terminals.remove('=')
    terminals.remove('1')
    terminals.append('$')
    mat = []
    for x in diction:
        row = []
        for y in terminals:
            row.append('')
        mat.append(row)
    grammar_is_LL = True
```

Fig 12: Function to create parse table, Part-1

```python
# Filling in the Parsing Table:
for lhs in diction:
    rhs = diction[lhs]
    for y in rhs:
        res = first(y)
        if '#' in res:
            if type(res) == str:
                firstFollow = []
                fol_op = follows[lhs]
                if fol_op is str:
                    firstFollow.append(fol_op)
                else:
                    for u in fol_op:
                        firstFollow.append(u)
                res = firstFollow
            else:
                res.remove('#')
                res = list(res) + list(follows[lhs])
        ttemp = []
        if type(res) is str:
            ttemp.append(res)
            res = copy.deepcopy(ttemp)
        for c in res:
            xnt = ntlist.index(lhs)
            yt = terminals.index(c)
            if mat[xnt][yt] == '':
                mat[xnt][yt] = f"{lhs}->{' '.join(y)}"
            else:
                if f"{lhs}->{y}" in mat[xnt][yt]:
                    continue
                else:
                    grammar_is_LL = False
                    mat[xnt][yt] = mat[xnt][yt] \
                        + f",{lhs}->{' '.join(y)}"

with open('parsingtable.txt', 'w', encoding='utf-8') as file:
    file.write("Generated parsing table:\n")
    headers = [""] + terminals
    rows = [[ntlist[j]] + y for j, y in enumerate(mat)]
    file.write(tabulate(rows, headers, tablefmt='fancy_grid'))

return (mat, grammar_is_LL, terminals)
```

Fig 13: Function to create parse table, Part-2

```python
def validateStringUsingStackBuffer(parsing_table, grammarll1, table_term_list, input_string, term_userdef, start_symbol):
    with open('parsing.txt', 'w') as file:
        # file.write(f"Validate String:\n{input_string}\n")
        print(f"Validate String:\n\n{input_string}\n")
        print("Look into the parsing.txt file for the parsing steps")
        print("\n")
        if grammarll1 == False:
            file.write(f"Input String = \"{input_string}\"\nGrammar is not LL(1)\n")
        stack = [start_symbol, '$']
        buffer = []
        input_string = input_string.split()
        input_string.reverse()
        buffer = ['$'] + input_string

        # Writing Header for Parsing Steps:
        file.write("{:>70} {:>10} {:>20}\n".format("Input\t\t\t\t\t\t\t\t\t\t\t\t\t\t", "Stack\t\t", "Action"))
        while True:
            # Checking for Valid End Condition:
            if stack == ['$'] and buffer == ['$']:
                file.write("{:>100} | {:>25} | {:>30}\n".format(' '.join(buffer), ' '.join(stack), "Valid"))
                return "Valid String!"

            # Parsing Non-terminals:
            elif stack[0] not in term_userdef:
                x = list(diction.keys()).index(stack[0])
                y = table_term_list.index(buffer[-1])
                if parsing_table[x][y] != '':
                    entry = parsing_table[x][y]
                    file.write("{:>100} | {:>25} | {:>30}\n".format(' '.join(buffer),
                                                                    ' '.join(stack), f"T[{stack[0]}][{buffer[-1]}] = {entry}"))
                    lhs_rhs = entry.split("->")
                    lhs_rhs[1] = lhs_rhs[1].replace('#', '').strip()
                    entryrhs = lhs_rhs[1].split()
                    stack = entryrhs + stack[1:]
                else:
                    file.write(f"Invalid String! No rule at Table[{stack[0]}][{buffer[-1]}]\n")
                    return "Invalid String! No rule at Table[{stack[0]}][{buffer[-1]}]"

            # Matching Terminals:
            else:
                if stack[0] == buffer[-1]:
                    file.write("{:>100} | {:>25} | {:>30}\n".format(' '.join(buffer), ' '.join(stack), f"Matched1:{stack[0]}"))
                    buffer = buffer[:-1]
                    stack = stack[1:]
                else:
                    file.write("Invalid String! Unmatched terminal symbols\n")
                    return "Invalid String! Unmatched terminal symbols"
```

Fig 14: Function to validate the string

```
sample_input_string = None
inps = ''
with open('input.txt', 'r+') as f:
    for line in f.readlines():
        inps += line
sample_input_string = inps
arr1 = inps.split()[4] # L[10]
id = inps.split()[7] # maxval
arr2 = inps.split()[9] # L[0]
arr3 = inps.split()[21] # L[i]

# Prints all the tokens
print("The tokens are:\n")
print(inps.split())
with open('tokens.txt','w') as f:
    for token in inps.split():
        f.write(token+"\n")

# Checking if the identifier maxval is following the format of a valid identifier
x = re.search("^[a-zA-Z][a-zA-Z0-9_]*", id)
if not x:
    print("Invalid identifier")
    exit(1)

# Checking if L[10] is a valid array name or not
y = re.search("^[a-zA-Z][a-zA-Z0-9_]*[[0-9]+]", arr1)
if not y:
    print("Invalid array name")
    exit(1)

# Checking if L[0] is a valid array name or not
z = re.search("^[a-zA-Z][a-zA-Z0-9_]*[[0-9]+]", arr2)
if not z:
    print("Invalid initialization")
    exit(1)

# Checking if L[i] is a valid array name or not
zz = re.search("^[a-zA-Z][a-zA-Z0-9_]*[[a-z]+]", arr3)
if not zz:
    print("Invalid index")
    exit(1)
```

Fig 15: Lexical Analyser

```python
# Rules for LL(1) grammar
rules = [
    "S -> T M B A D",
    "T -> int",
    "M -> main()",
    "B -> begin",
    "D -> End",
    "A -> E F G W X",
    "E -> T " + arr1 + " ;",
    "F -> T " + id + " = " + arr2 + " ;",
    "G -> for C do",
    "C -> i = 1 to n - 1",
    "W -> if " + arr3 + " > " + id + " P Q R",
    "P -> " + id + " = " + arr3 + " ;",
    "Q -> endif",
    "R -> endfor",
    "X -> return ( " + id + " )"
]

# List of non terminals in our grammar
nonterm_userdef = ['S', 'T', 'M', 'B', 'D', 'A', 'E', 'F', 'K', 'Z', 'G', 'W', 'P', 'Q', 'R', 'X', 'C']
# List of terminals in our grammar
term_userdef = [id, arr1, arr2, arr3, 'n', 'int', 'main()', 'End', 'for', 'if', 'begin', 'do', 'i', 'to',
                '(', ')', '+', '-', 'endif', 'endfor', 'return', '>', '=', '1', ',', ';']
diction = {}
firsts = {}
follows = {}
computeAllFirsts()
start_symbol = list(diction.keys())[0]
computeAllFollows()
(parsing_table, result, tabTerm) = createParseTable()
if sample_input_string != None:
  validity = validateStringUsingStackBuffer(parsing_table, result,
  tabTerm, sample_input_string,term_userdef, start_symbol)
  print(validity)
else:
    print("No input String detected")
```

Fig 16 : Driver code

4. RESULT

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
S	{'int'}	{'$'}
T	{'int'}	{'maxval', 'main()', 'L[10]'}
M	{'main()'}	{'begin'}
B	{'begin'}	{'int'}
D	{'End'}	{'$'}
A	{'int'}	{'End'}
E	{'int'}	{'int'}
F	{'int'}	{'for'}
G	{'for'}	{'if'}
C	{'i'}	{'do'}
W	{'if'}	{'return'}
P	{'maxval'}	{'endif'}
Q	{'endif'}	{'endfor'}
R	{'endfor'}	{'return'}
X	{'return'}	{'End'}

Fig 17: FIRST and FOLLOW result table

Generated parsing table:

	maxval	L[10]	L[0]	L[i]	n	int	main()	End	for	if	begin	do	i	to	endif	endfor	return	>	,	;	$
S						S->T M B A D															
T						T->int															
M							M->main()														
B											B->begin										
D								D->End													
A						A->E F G W X															
E						E->T L[10] ;															
F						F->T maxval = L[0] ;															
G									G->for C do												
C													C->i = 1 to n - 1								
W										W->if L[i] > maxval P Q R											
P	P->maxval = L[i] ;																				
Q															Q->endif						
R																R->endfor					
X																	X->return (maxval)				

Fig 18: Parsing Table

Fig 19: String parsing

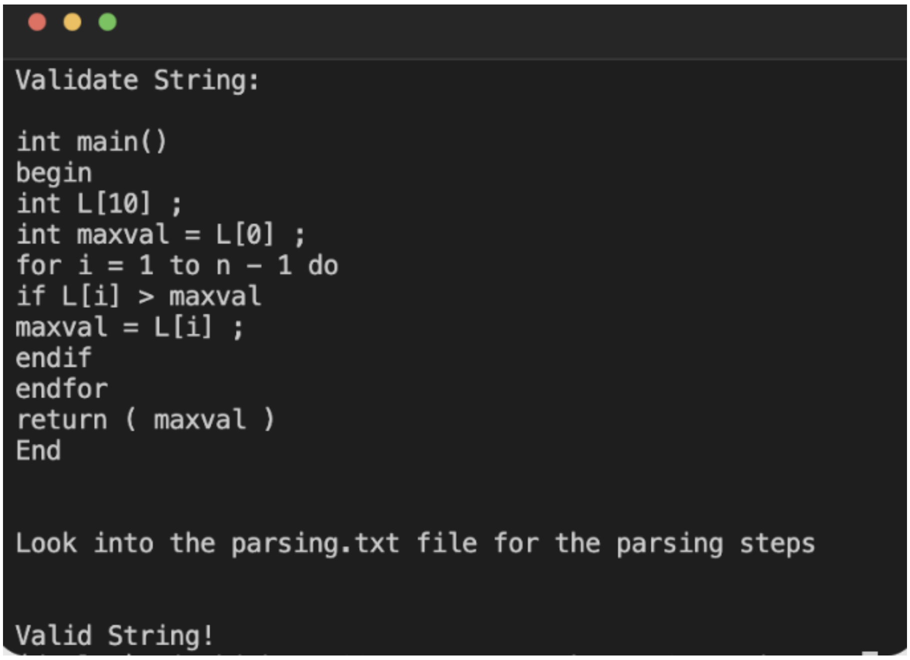

Fig 20:Terminal Output

5. CONCLUSION

We successfully implemented the front end which is the lexical analysis and syntax analysis part of the compiler for the given hypothetical problem statement to find the greatest element in an array using a top-down parser that is LL(1)

The language used in this project is Python. The lexical analyzer implemented successfully generates the token for the given problem statement. The syntax analysis checks for ambiguity, left recursion, and left factoring of the grammar. The parser generates the first and following sets and also generates a parsing table. Finally using the generated parsing table string is parsed.

6. REFERENCES

1. https://docs.python.org/3/
2. https://pypi.org/project/tabulate/
3. https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/
4. https://www.youtube.com/watch?v=9C1vEG1udtY