Date: _____

Experiment No 1

Aim: To write an article about Compiler Design.

Introduction

Normally a program building process involves four stages and utilizes different tools such as a pre-processor, compiler, assembler, and linker. At the end there should be a single executable file. Below are the stages that happen

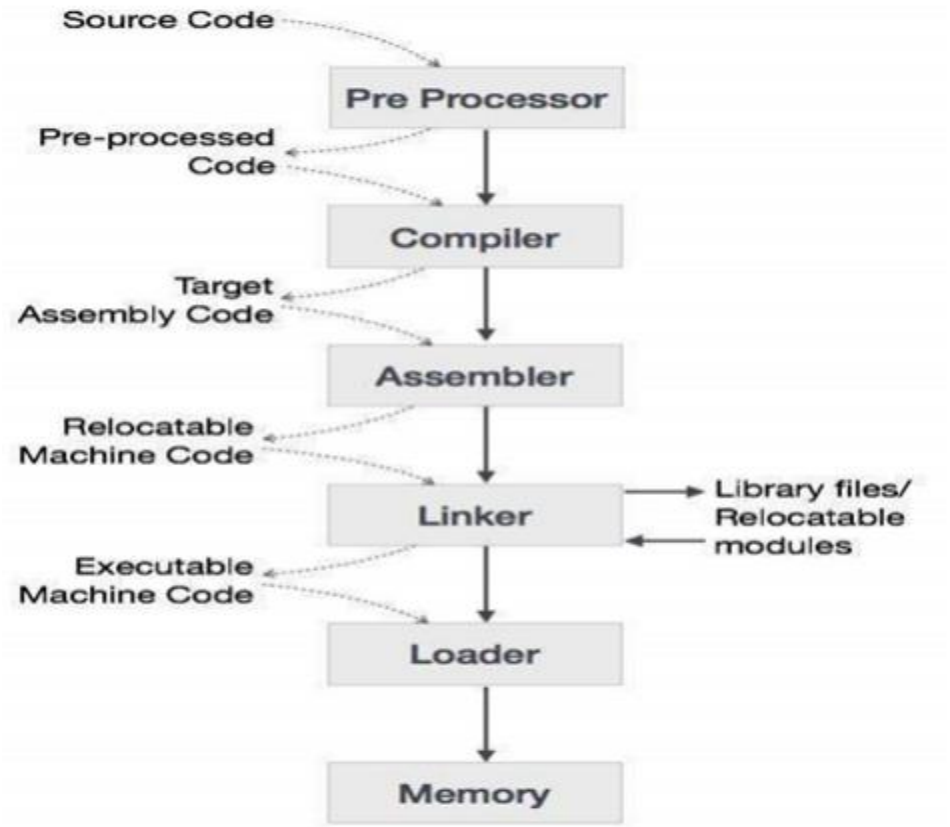


Fig 1.1- Language Processing System (courtesy- geeksfor geeks)

Pre-processing is the first pass of any C compilation. It processes include files, conditional compilation instructions and macros.

Compilation is the second pass. It takes the output of the pre-processor, and the source code, and generates assembler source code.

Assembly is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.

Linking is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

In this process, compilation phase use compiler or interpreter. But we will be discussing about compiler.

A Compiler is a special program that processes statements written in a particular programming language and turns them into machine language or ”code” that a computer’s processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an editor. The file that is created contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

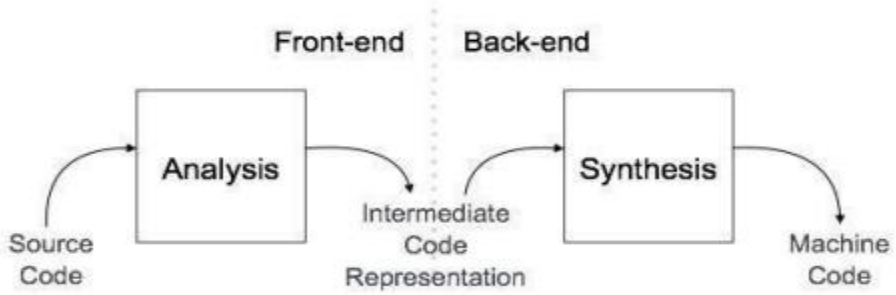


Fig 1.2- Front-end and Back-end of a compiler (courtesy- geeksfor geeks)

Why do we need compiler?

Any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in highlevel language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.

Interpreter is also a program that executes instructions written in a highlevel language. So why we prefer compiler over interpreter to convert a highlevel language into machine-level language? Let’s understand the difference between them.

Phases of Compiler

The compilation process is a sequence of various phases. First phase of compiler takes source program as input. Then 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. The last phase i.e. Code Generator generates the required code. There are the following phases of compiler:

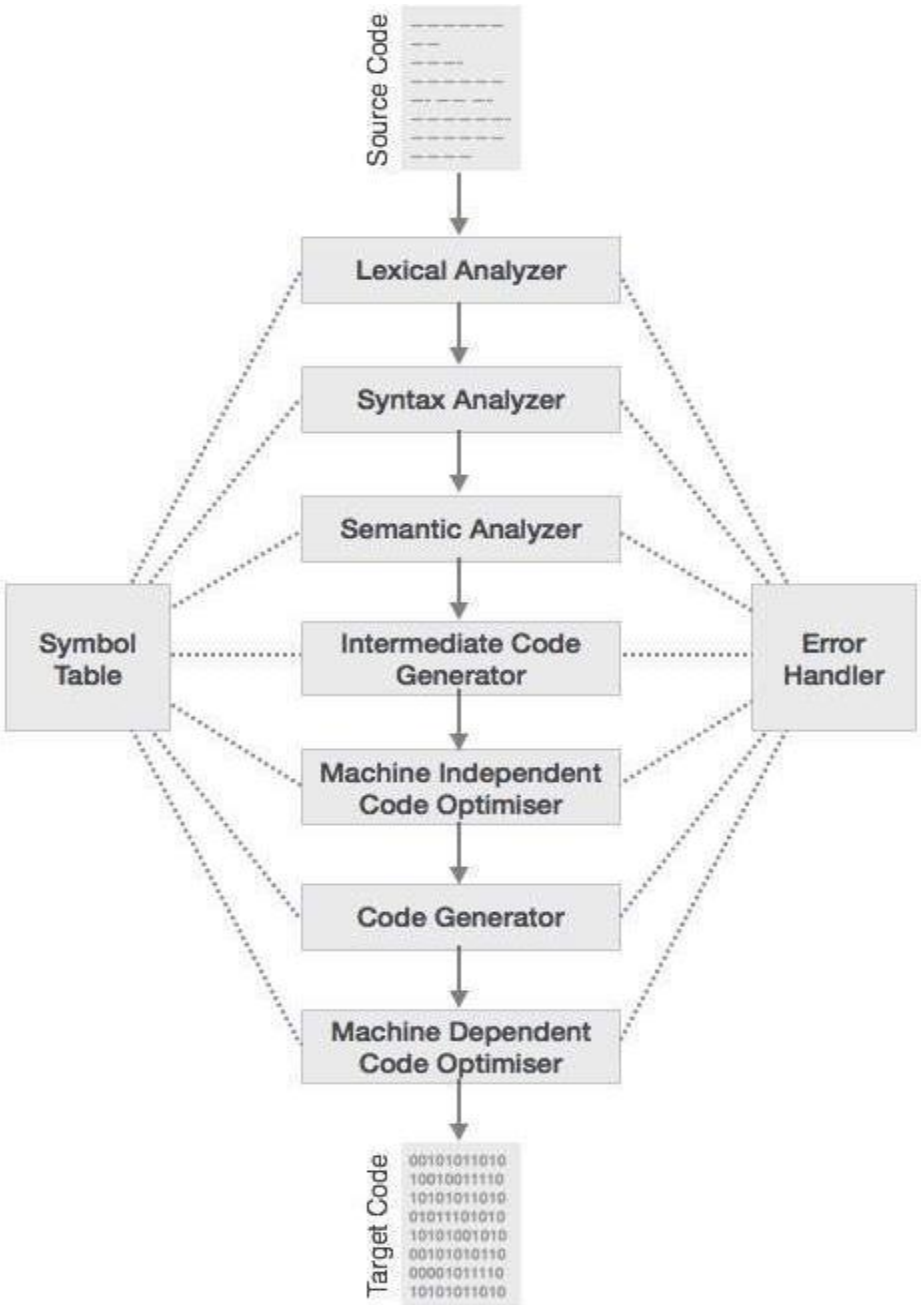


Fig 1.3 – Phases of Compiler (courtesy- geeksfor geeks)

1. Lexical Analysis

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

Tokens

Lexemes are said to be a sequence of characters alphanumericalphanumeric 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.

2. Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser. We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by pushdown automata.

CFG, on the other hand, is a superset of Regular Grammar, as depicted below:

Relation of CFG and Regular Grammar

It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

Context-Free Grammar

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.A context-free grammar has four components:

1. A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
2. A set of tokens, known as terminal symbols. Terminals are the basic symbols from which strings are formed.
3. A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.
4. One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

3. Semantic Analysis

The plain parse-tree constructed in previous phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = “value”;

It should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the

language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

1. Scope resolution
2. Type checking
3. Array-bound checking

Here are some of the semantics errors that the semantic analyzer is expected to recognize:

1. Type mismatch
2. Undeclared variable
3. Reserved identifier misuse.
4. Multiple declaration of variable in a scope.
5. Accessing an out of scope variable.
6. Actual and formal parameter mismatch.

4. Intermediate Code Generation

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.

1. If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
2. Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
3. The second part of compiler, synthesis, is changed according to the target machine.
4. It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

This procedure should create an entry in the symbol table, for variable name, having its type set to type and relative address offset in its data area.

5. Code Optimization

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed. In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

1. The output code must not, in any way, change the meaning of the program.
2. Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
3. Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

1. At the beginning, users can change/rearrange the code or use better algorithms to write the code.
2. After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
3. While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

Machine-independent Optimization : In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.

Machine-dependent Optimization: Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

6. Code Generation

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

1. It should carry the exact meaning of the source code.
2. It should be efficient in terms of CPU usage and memory management.
3. We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

Error Handling

One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler. Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Table Management OR Book-keeping

A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

Date: _____

Experiment No. 2

Aim: Implement a program for constructing a transition table and string processing for the following type of DFAs.

1. **All string ending with particular type of suffix**
2. **All string starting with particular type of prefix**
3. **All string constituting particular type of substring**

Introduction:

```
Formal specification of machine is
{ Q, ∑, q, F, δ }.

DFA consists of 5 tuples {Q, ∑, q, F, δ}.
Q : set of all states.
∑ : set of input symbols. (Symbols which machine takes as input)
q : Initial state. (Starting state of a machine )
F : set of final state.
δ : Transition Function, defined as δ : Q X ∑ --> Q.
```

In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character. One important thing to note is, there can be many possible DFAs for a pattern. A DFA with minimum number of states is generally preferred.

Input:

1. Input the alphabets.
2. Select particular type of DFA
3. Input string composed from the alphabet {a,b} to be checked for acceptance for the selected DFA.
4. Input the string for processing.

Expected Output:

If the string is ending with ‘ab’ or starting with ‘a’ or having a substring ‘ab’, then a message “String is valid” should be displayed otherwise a message “String is invalid” should be displayed.

Algorithm 2.1: Generating the transition table for selected type of DFA

```
Input:
        1.      Read the alphabets
        2.      Read the choice for the type of DFA (i.e. suffix, prefix, substring)
        3.      Read the string for that particular DFA

Output:
        1.      Displays the Transition State Table
        2.      Shows the state transitions and processing for the string


main()
        1.      START
        2.      Input alphabets
        3.      Input the type of DFA
                1. String ending with a particular suffix
                2. String starting with a particular prefix
                3. String containing a particular substring
        4.      Input the string
        5.      Initialize the transitionstates data structure (here, dictionary)
                for storing the transition table
        6.      if choice = 1
                        traverse the string and
                        if alphabet[char] = string[char]
                                set the transition to next state
                        else
                                set the transition to dead state
                else
                        create a stateinfo dictionary
        7.      in stateinfo, set the state as keys and string as values in increasing order
        8.      create values list, having all the from stateinfo
        9.      traverse the stateinfo dictionary and
                        if value[item] + alphabet is found in values
                                return index
                        else
                                remove first character
                                goto if statement again
        10.     set the transition state to the state returned above
        11.     DFA is ready
        12.     print the DFA in tabular form
        13.     input the string for processing
        14.     set currstate to initial state and print it
        15.     traverse the string
                        if currstate is deadstate
                                print(dead state..string rejected)
                                goto step 17
                        else
                                set currstate to next state in transitionstates
                                print that state
        16.     if currstate = finalstate
                        print(string accepted)
                else
                        print(string rejected)
        17.     STOP
```

Flowchart:

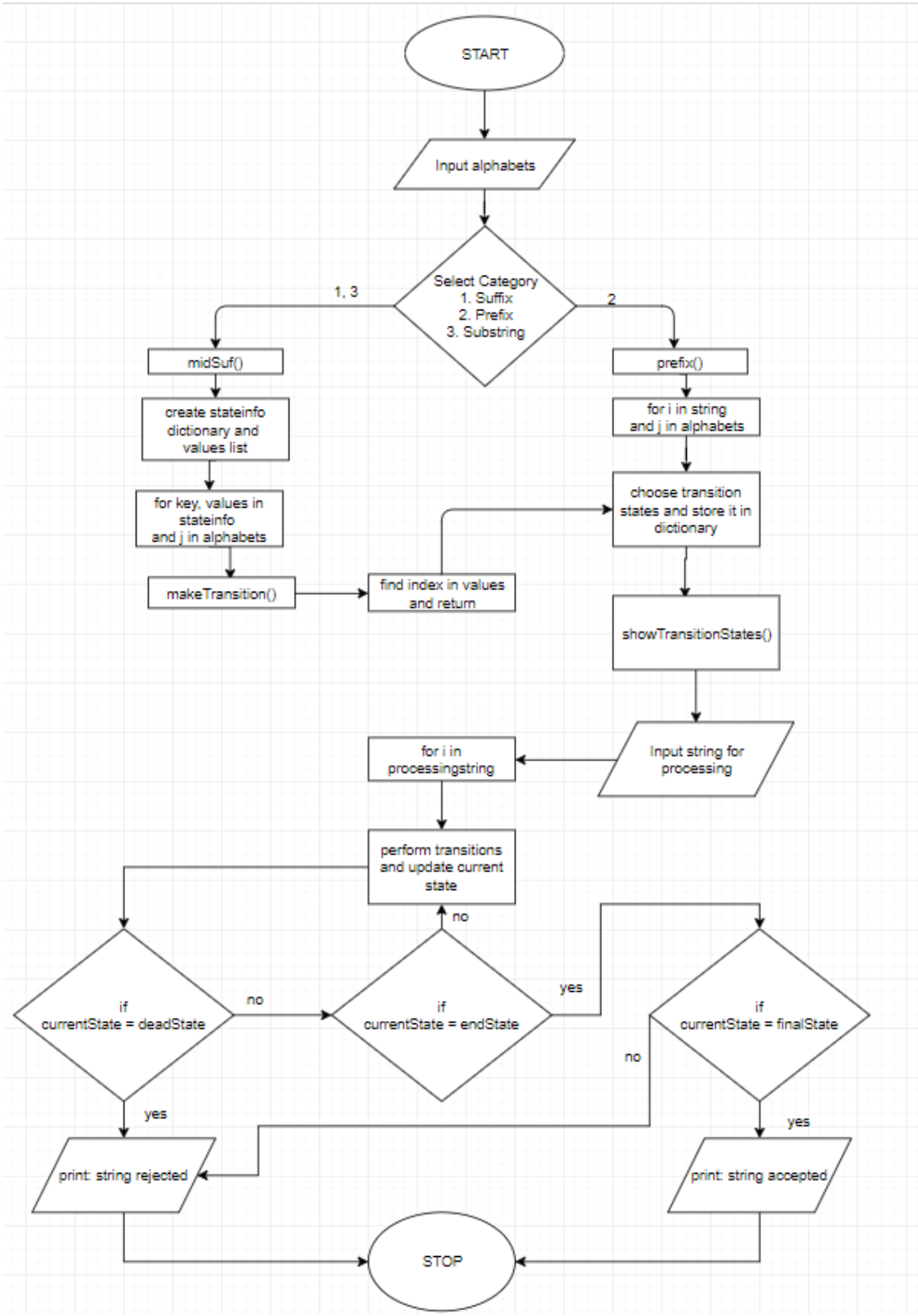


Fig 2.2 – Flowchart for algorithm 2.1

Source Code:

```
def show(transitionStates, alphabets):
    print("\nState Transition table is\n")
    print('{:>10}'.format("|"), end="")
    for j in alphabets:
        print('{:>10}'.format(j+"    "+"|"), end="")
    print()

    for i in transitionStates:
        if(str(i) == "q0"):
            print('{:>10}'.format("-> q0    |"), end="" )
            for j in transitionStates[i]:
                print('{:>10}'.format(j+"    "+"|"), end="")
            print()
        else:
            print('{:>10}'.format(str(i)+"    |"), end="" )
            for j in transitionStates[i]:
                print('{:>10}'.format(j+"    "+"|"), end="")
            print()

def makeTransition(strcmp, values, key, j, transitionStates):
    if(strcmp[:3] == 'eps'):
        strcmp = strcmp[3:]
    index = 0
    while strcmp:
        if(strcmp in values):
            index = values.index(strcmp)
            return index
        strcmp = strcmp[1:]
    return index

def midSuf(transitionStates, string, alphabets, category):
    stateinfo = {}
    count = 0
    stateinfo.update({'q' + str(count):['eps']})
    count = count + 1
    while count <= len(string):
        for i in range(count):
            stateinfo.update({'q' + str(count):[string[:i+1]]})
        count = count + 1

    values = []

    for key, val in stateinfo.items():
        values.append(val[0])

    for key, val in stateinfo.items():
        for j in alphabets:
            strcmp = val[0] + j
            index = makeTransition(strcmp, values, key, j,
transitionStates)
            transitionStates[key].append('q' + str(index))
        del transitionStates[key][0]
```

```
    if(category == '2'):
        k = len(transitionStates) - 1
        for j in range(len(alphabets)):
            transitionStates['q' + str(k)][j] = 'q' + str(k)

def prefix(transitionStates, string, alphabets):
    k = 0
    for i in range(len(string)):
        for j in alphabets:
            if(string[i] == j):
                transitionStates['q' + str(k)].append('q' +
str(k+1))
            else:
                transitionStates['q' + str(k)].append('q-1')
        k = k + 1
        del transitionStates['q' + str(i)][0]

    k = len(transitionStates) – 1

    for j in alphabets:
        transitionStates['q' + str(k)].append('q' + str(k))
    del transitionStates['q' + str(k)][0]

def processString(transitionStates):
    processingString = input("\nEnter string to be processed: ")
    for i in transitionStates:
        currstate = i
        break
    print()
    print(" -> " + currstate, end="")
    for i in processingString:
        j = alphabets.index(i)
        if(currstate == 'q-1'):
            print("\nDead State...String Rejected")
            return
        currstate = transitionStates[currstate][j]
        print(" -> " + currstate, end="")
    print()
    finalState = 'q' + str(len(transitionStates)-1)
    if(currstate == finalState):
        print("\nString accepted")
    else:
        print("\nString rejected")

alphabets = input("\nEnter all alphabets(with space): ").split()
numberOfAlphabets = int(len(alphabets))
cat = ["suffix", "prefix", "substring"]
category = input("Enter category: suffix (0) | prefix (1) |
substring (2): ")
string = input("Enter " + cat[int(category)] + ": ")
transitionStates = {}

for i in range(len(string) + 1):
```

```
    transitionStates.update({('q' + str(i)):[' ']})

if(category == '1'):
    prefix(transitionStates, string, alphabets)
else:
    midSuf(transitionStates, string, alphabets, category)

show(transitionStates, alphabets)

processString(transitionStates)
```

Output:

1. **Ending with 'ab'.**

```
Enter all alphabets(with space): a b
Enter category: suffix (0) | prefix (1) | substring (2): 0
Enter suffix: ab

State Transition table is

         |    a    |    b    |
-> q0    |   q1    |   q0    |
   q1    |   q1    |   q2    |
   q2    |   q1    |   q0    |

Enter string to be processed: ababaab

 -> q0 -> q1 -> q2 -> q1 -> q2 -> q1 -> q1 -> q2

String accepted
```

Fig 2.3 Output for String Ending with suffix ab

2. **With Substring 'ab'.**

```
Enter all alphabets(with space): a b
Enter category: suffix (0) | prefix (1) | substring (2): 2
Enter substring: ab

State Transition table is

         |    a    |    b    |
-> q0    |   q1    |   q0    |
   q1    |   q1    |   q2    |
   q2    |   q2    |   q2    |

Enter string to be processed: aaabbb

 -> q0 -> q1 -> q1 -> q1 -> q2 -> q2 -> q2

String accepted
```

Fig 2.4 Output for String containing substring ab

3. **Starting with ‘a’.**

```
Enter all alphabets(with space): a b
Enter category: suffix (0) | prefix (1) | substring (2): 1
Enter prefix: a

State Transition table is

         |    a    |    b    |
-> q0    |   q1    |  q-1    |
   q1    |   q1    |   q1    |

Enter string to be processed: abba

 -> q0 -> q1 -> q1 -> q1 -> q1

String accepted
```

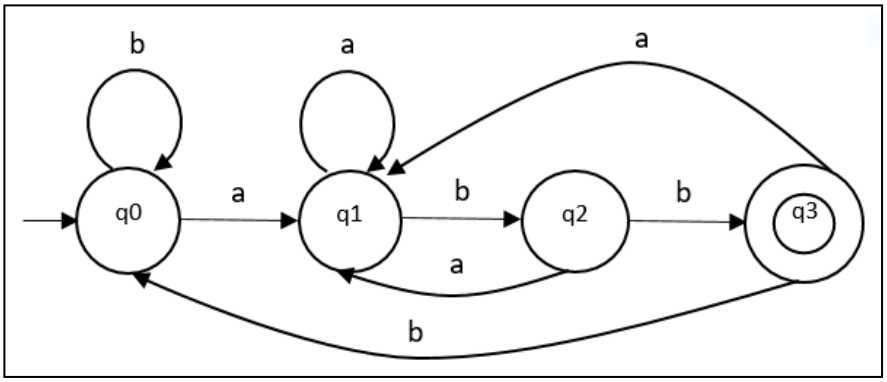
Fig 2.5 Output for String starting with prefix ab

Frequently Asked Questions

1. **What is a string?**

   A string is a finite sequence of symbols selected from some alphabet.

2. **Design a DFA over alphabet {a,b} accepting strings beginning with “ab” .**



3. **What are regular expressions?**

These are mathematical expressions used to describe regular languages. In compiler design regular expressions are used for description of tokens.

4. **Write the regular expression for strings containing three consecutive a's over alphabet {a, b}.**

   (a+b)* aaa (a+b)*

Date: _____

Experiment No. 3

Aim - Write a python program to count number of single line and multi-line comments in the user input or from a file containing program.

Introduction:

In Python, there are two ways to annotate your code.

The first is to include comments that detail or indicate what a section of code – or snippet – does.

The second makes use of multi-line comments or paragraphs that serve as documentation for others reading your code.

Think of the first type as a comment for yourself, and the second as a comment for others. There is not right or wrong way to add a comment, however. You can do whatever feels comfortable.

Single-line comments are created simply by beginning a line with the hash (#) character, and they are automatically terminated by the end of line.

Comments that span multiple lines – used to explain things in more detail – are created by adding a delimiter (''') on each end of the comment.

```
#This would be a comment in Python

''' This would be a multiline comment
in Python that spans several lines and
describes your code, your day, or anything you want it to'''
```

Fig 3.1 – Comments in Python

Input:

1. Input choice (Read from file or give user input)
2. Input the file name (if code is to be read from file) or the user is expected to enter a program.

Expected Output:

1. No. of lines found
2. No. of single line comments found.
3. No. of multi-line comments found.

Algorithm 3.2: Finding the single and multiple line comments in python program

```
INPUT
        1.      Input choice
        2.      Input the filename or the enter the program

OUTPUT
        1.      No. of lines found
        2.      No. of single line comments
        3.      No. of multiple line comments

main()
        1.      START
        2.      Set noOfLines, single and multiple variables count to 0
        3.      Enter choice and enter the filename or the program
        4.      Traverse the program line by line
        5.      Find index of #, ''', "
        6.      If index(") > index(#) or index('''')
                        If index(#) < index('''')
                                increment single
                        Else increment multiple
        7.      Display single and multiple
        8. STOP
```

Flowchart:

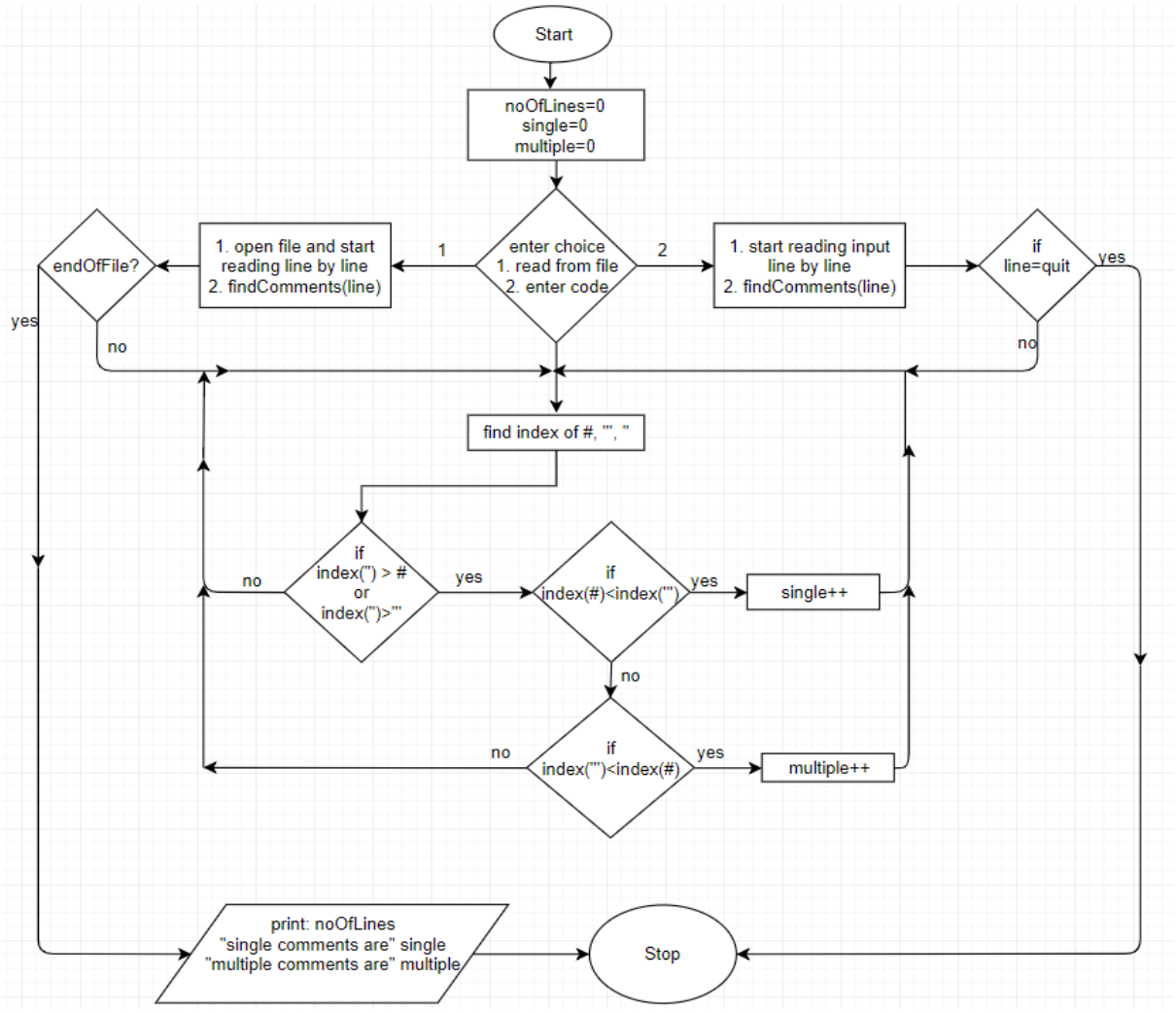


Fig 3.3 Flowchart for the algorithm 3.2

Source Code:

```
def findComments(data):
    global single
    global multiple
    global comment
    if(comment == 'm'):
        mfind = data.find(multipleCom)
        if(mfind != -1):
            comment = '-1'
            multiple = multiple + 1
    else:
        sfind = data.find(singleCom)
        mfind = data.find(multipleCom)
        stringactive = data.find(stringopen)

        if((sfind<stringactive and stringactive^sfind != 0 and
sfind!=-1) or (mfind<stringactive and stringactive^mfind != 0 and
mfind!=-1) or stringactive == -1):
            if(sfind!= -1 and mfind == -1):
                single = single + 1
            elif(sfind==-1 and mfind!=-1):
                comment = 'm'
                multiple = multiple + 1
                mfind2 = data.rfind(multipleCom)
                if((mfind<mfind2)):
                    comment = '-1'
                    multiple = multiple + 1
            elif(sfind!=-1 and mfind !=-1):
                if((sfind<mfind) and (sfind!=-1)):
                    comment = 's'
                    single = single + 1
                elif((mfind<sfind) and (mfind!=-1)):
                    comment = 'm'
                    mfind2 = data.rfind(multipleCom)
                    multiple = multiple + 1
                    if((mfind!=mfind2) and (mfind2!=-1)):
                        comment = '-1'
                        multiple = multiple + 1
```

```
single = 0
multiple = 0
multipleCom = "'''"
singleCom = "#"
stringopen = '"'
comment = '-1'
noOfLines=0

print("enter choice \n 1. read from file \n 2. give input")
choice = input()

if(choice == '1'):
    filename = input("\nEnter filename: ")
    with open(filename, 'r') as file:
        for data in file.readlines():
            noOfLines=noOfLines+1
            findComments(data)

elif(choice == '2'):
    data = " "
    print("\nEnter quit when done\n")
    while(data!="quit"):
        noOfLines=noOfLines+1
        data = input()
        findComments(data)

print("\nNo. of lines: ", noOfLines)
print("single-line comments: ", single, "\nmulti-line comments: ",
int(multiple/2))
```

Input File:

```
#asdasds
'''asdasdas'''
'''asdasds
'''
'''sdasds#asdfaf'''
#fafa'''asas""'''

print("#saasfd")
```

Output:

1. **Reading from program file**

```
E:\F\College\sem_6\CS 604 Compiler Design\Lab\2019-02-05 comm>python comments.py
enter choice
 1. read from file
 2. give input
1

Enter filename: code.py

No. of lines:  9
single-line comments:  2
multi-line comments:  3

E:\F\College\sem_6\CS 604 Compiler Design\Lab\2019-02-05 comm>
```

Fig 3.4 Output for finding single amd multi-line comments from file

2. **User giving input**

```
E:\F\College\sem_6\CS 604 Compiler Design\Lab\2019-02-05 comm>python comments.py
enter choice
 1. read from file
 2. give input
2

Enter quit when done

#asdasd
'''sdasds'''
'''asdae
'''
'''edeed#asdsca'''
#sacsaas'''adas""sadS'''

PRINT("#HELLO")

quit

No. of lines:  10
single-line comments:  2
multi-line comments:  3

E:\F\College\sem_6\CS 604 Compiler Design\Lab\2019-02-05 comm>
```

Fig 3.4 Output for finding single amd multi-line comments from user input

Frequently Asked Questions:

1. **List the different types of comment styles in C language.**

   There are two types of comment styles used in C:

- **Single line comment** - Single Line Comment is used to comment out just Single Line in the Code. It is used to provide one liner description of line. It begins with //.
- **Multi-line comment** – It is used to cover comments over multiple lines. Multi-line comment starts with /* and ends with */

2. Give an example of Single line comments and multi-line comments.

```
/*-------------------------------------------------------
        Title : Program Name
        Author: Author Name
 -------------------------------------------------------*/
#include<stdio.h>
int main()
{
        int num1=1,num2=2;  // Declare Variables
        int sum = 0;        // Declare Sum
        sum = num1 + num2;  // Compute Sum
        printf("Sum : %d",sum);
        return(0);
}
```

3. How compiler treats comment lines.

Since the compiler treats a comment as a single white-space character, you cannot include comments within tokens. The compiler ignores the characters in the comment.

Date: _____

PRACTICAL NO 4

Aim: To write a python program to construct a lexical analyzer for a hypothetical language.

Introduction:

Lexical Analysis is the first phase of compiler also known as scanner. It converts the Highlevel input program into a sequence of Tokens.

Lexical Analysis can be implemented with the Deterministic finite Automata.

The output is a sequence of tokens that is sent to the parser for syntax analysis

Input:

1. Input choice (Read from file or give user input)
2. Input the file name (if code is to be read from file) or the user is expected to enter a program.

Expected Output:

1. Display number of lines in the code.
2. Display all the tokens (keywords, operators, delimeters) found in program

Algorithm 4.1: To construct a lexical analyzer for python language

```
INPUT
        1.      Input choice
        2.      Input the filename or the enter the program

OUTPUT
        1.      No. of lines found
        2.      No. of single line comments
        3.      No. of multiple line comments

main()
        1.      START
        2.      Define the tokens of the language (here python) that is to be scanned by lexical analyzer
                        a.      Identifiers, literals, operators, delimeters are subset of tokens.
        3.      Initialize all the buffer_lists to store tokens and set noOfLines to 0
        4.      Enter the program which is to be scanned by lexical analyzer
        5.      Traverse the program character by character
                        Identify keywords, operators, delimeters and store them in their respective buffer_lists
        6.      Increment noOfLines counter
        7.      Display all the tokens found
        8.      STOP
```

Flowchart:

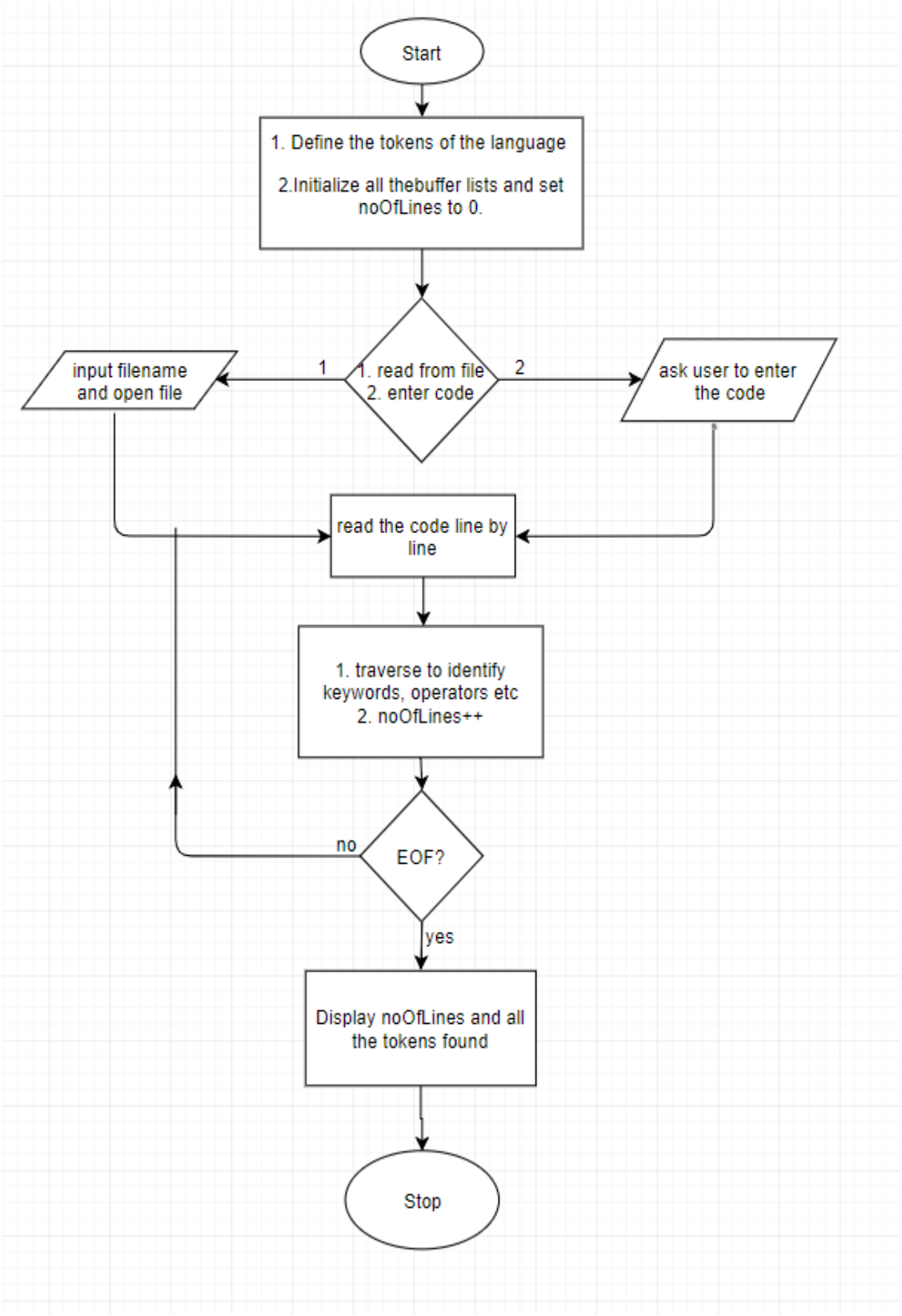


Fig 4.2 Flowchart for algorithm 4.1

Source Code:

```
def lexicalParser(line):
    global key_buffer
    global key_print
    global iden_buffer
    global iden_print
    global op_buffer
    global op_print
    global buffer
    global deli_buffer

    for c in line:
        if c in operators:
            op_buffer = op_buffer + str(c)
        elif c in delimeters and c not in deli_buffer:
            deli_buffer = deli_buffer + str(c)
        elif(c.isalnum()):
            buffer = buffer + str(c)
        elif((c==' ' or c=='\n') and buffer):
            if(buffer in keywords):
                key_buffer.append(str(buffer))
            else:
                iden_buffer.append(str(buffer))
            buffer = ""

    for ch in op_buffer:
        if ch not in op_print:
            op_print = op_print + ch
    for i in key_buffer:
        if i not in key_print:
            key_print.append(i)
    for i in iden_buffer:
        if i not in iden_print:
            iden_print.append(i)

import keyword
keywords = keyword.kwlist
```

```
delimeters = ["(", ")", "[", "]", "{","}", ",", ":", ".", "`", "=",
";", "+=", "-=", "*=", "/=", "%=", "**=", "&=", "|=", "^=", ">>=",
"<<="]

key_buffer = []
key_print = []
iden_buffer = []
iden_print = []
operators = "+-*/%="
j,k,z=0,0,0
op_buffer = ""
op_print = ""
buffer = ""
deli_buffer = ""
noOfLines = 0

print("enter choice \n 1. read from file \n 2. give input")
choice = input()
if choice=="1":
    with open("program.txt") as file:
        for line in file.readlines():
            noOfLines=noOfLines+1
            lexicalParser(line)

elif choice=="2":
    line = " "
    print("\nEnter quit when done\n")

    while(line!="quit"):
        noOfLines=noOfLines+1
        line = str(input())
        lexicalParser(line)

print("\nnumber of lines are: ", noOfLines)
print("\noperators are: ", ", ".join(op_print))
print("\nkeywords are: ","  ".join(key_print))
print("\nidentifiers are: ","  ".join(iden_print))
print("\ndelimeters are: ", ", ".join(deli_buffer))
```

Input File:

```
a = 1
if ( a == 1 ):
        print (" hello world ")
elif: (a == 2):
        print("hey")
else:
        return 0
b1 = 10
```

Output:

1. **Reading from file**

```
E:\F\College\sem_6\CS 604 Compiler Design\Lab\2019-03-19 la gr>python LA.py
enter choice
 1. read from file
 2. give input
1

number of lines are:  8

operators are:  =

keywords are:  if, elif, else, return

identifiers are:  a, 1, print, hello, world, 2, printhey, 0, b1, 10

delimeters are:  (, ), :

E:\F\College\sem_6\CS 604 Compiler Design\Lab\2019-03-19 la gr>
```

Fig 4.3 Output for lexical analyzer scanning program file

2. **User Input**

```
E:\F\College\sem_6\CS 604 Compiler Design\Lab\2019-03-19 la gr>python LA.py
enter choice
 1. read from file
 2. give input
2

Enter quit when done

print (" hello ")
if ( a == 1 ):
a = a + 2
quit

number of lines are:  4

operators are:  =, +

keywords are:  if

identifiers are:  print, hello, a, 1

delimeters are:  (, ), :
```

Fig 4.4 Output for lexical analyzer scanning user input code

Frequently asked Questions:

1. **What is the main function of a Lexical Analyzer.**

   Lexical Analyzer takes a source program as an input. It scans the entire source code as a stream of characters and convert it into sequence of tokens i.e. strings with an identified "meaning".

2. **What is a token?**

   A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages. Identifiers, keywords, constants, operators and punctuation symbols are typical tokens.

3. **How tokens are described and recognized during the lexical analysis phase of a compiler?**

   Token description is done with the help of Regular Expressions and token recognition is done with the help of Finite Automata (DFA or NFA).

Date: _____

PRACTICAL NO 5

AIM : To check whether a string is accepted by a CFG or not.

Introduction:

A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings. It consists of the following components:

- a set of terminal symbols
- a set of nonterminal symbols
- a set of productions (rules for replacing (or rewriting) nonterminal symbols)
- a start symbol

To generate a string of terminal symbols from a CFG, we:

- Begin with a string consisting of the start symbol;
- Apply one of the productions with the start symbol on the lefthand size, replacing the start symbol with the righthand side of the production;
- Repeat this process until all nonterminals have been replaced by terminal symbols.

Input:

1. Input set of productions rules.
2. Input the string.

Expected Output:

The program should output the CFG and print

1. String is acceptable by the parser
   a. Parser Tree (showing the production rules used)
2. String is not acceptable by the parser.

Algorithm 5.1: To check whether a string is accepted by CFG or not

```
INPUT
        1.      Input set of production rules
        2.      Input the string

OUTPUT
        1.      If string is accepted, display "String Accepted"
                Parser Tree (showing the production rules)
        2.      Else display "String Rejected"
```

```
main()
        1.      START
        2.      Initialize all the data structure to store all the production rules for a grammar.
        3.      Input the grammar rules.
        4.      Store the production rules in data structure
        5.      Print the grammar.
        6.      Input the string.
        7.      Initialize the parserlist (i.e. parse tree) with S, S being the start symbol.
        8.      If top element in parserlist matches the next input symbol
                        Increment m count
                        Push that symbol in parserlist
        9.      Else if top element is non-terminal,
                        Pop that element from parserlist
                        Look for its production rules and replace it.
        10.     Else
                        break
        11.     Repeat steps 7 and 8 until string is traversed
        12.     If all the elements in parserlist are same as the given string
                        Print “String accepted”
                        Display the parserlist
        13.     Else
                        Print "String Rejected"
        14.     STOP
```

Flowchart:

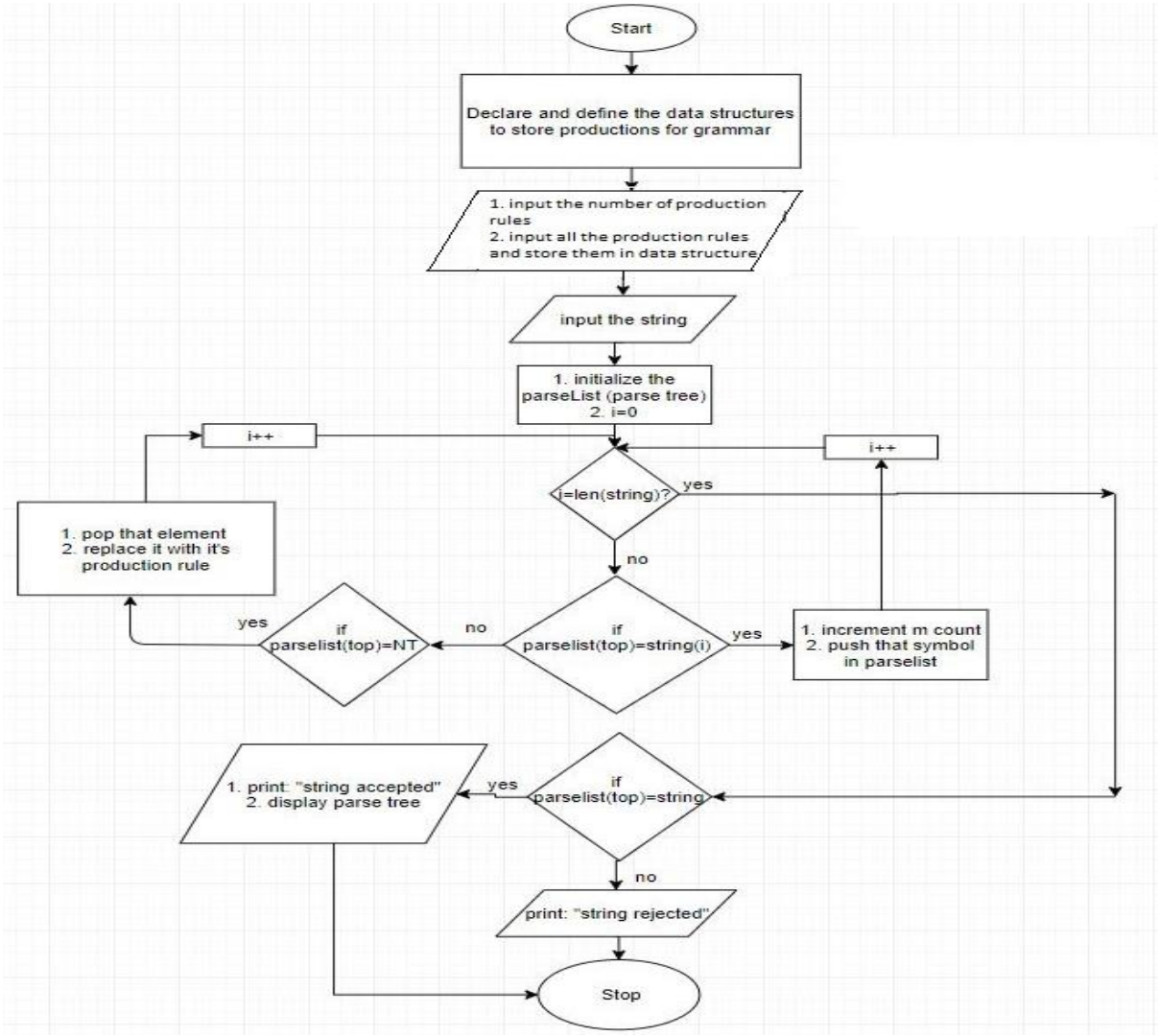


Fig 5.2 Flowchart for algorithm 5.1

Source Code:

```
def findter():
    global temp
    global n
    for k in range(n):
        if temp[i]==prod[k][lhs][0]:
            for t in range(int(nlist[k])):
                templist = list(temp)
                temp2list = []
                temp2list = templist[i+1:]
                templist[i:] = ""
                rhslist = []
                rhslist = list(prod[k][rhs][t])
                templist[i:] = rhslist[:]
                for ii in temp2list:
                    templist.append(ii)
                temp = "".join(templist)
                if string[i] == temp[i]:
                    return;
                elif string[i]!=temp[i] and temp[i].isupper():
                    break;
            break
    if temp[i].isupper():
        if temp not in outputlist:
            parserlist.append(temp)
        findter()
string=""
temp=""
lhs, rhs= 0, 1
n,z,x=0,0,0
i = 0
prod = []
nlist = []
outputlist = []
no = int(input("Enter number of production rules: "))
print("\nEnter production rules: \n")
```

```
for on in range(no) :
    line = input()
    listtemp = line.split()
    listrhs = []
    listrhs.append(listtemp[rhs])
    listt = []
    listt.append(listtemp[lhs])
    listt.append(listrhs)
    if n>0 and listt[lhs] == prod[n-1][lhs]:
        prod[n-1][rhs].append(listt[rhs][0])
        nlist[n-1] = str(int(nlist[n-1]) + 1)
    else:
        prod.append(listt)
        nlist.append(str(1))
        n=n+1
print("The grammar is: ")
for j in prod:
    print(j[0], " -> ", " | ".join(j[1]))
while(1):
    string = input("\nEnter any string (0 for exit): ")
    if(string == "0"):
        exit(1)
    for j in range(int(nlist[0])):
        parserlist = []
        parserlist.append("S")
        temp = prod[0][rhs][j]
        m=0
        for i in range(len(string)):
            if i<len(temp) and string[i] == temp[i]:
                m=m+1
                if temp not in outputlist:
                    parserlist.append(temp)
            elif i<len(temp) and string[i]!=temp[i] and
temp[i].isupper():
                findter()
                if string[i]==temp[i]:
```

```
                        m=m+1
                        if temp not in outputlist:
                            parserlist.append(temp)
                elif i<len(temp) and string[i]!=temp[i] and
    (ord(temp[i])<65 or ord(temp[i])>90):
                    break
            if m==len(string) and len(string)==len(temp):
                print("\nString Accepted\n")
                print("We used LMD Top-Down approach\n")
                print('{:>10}'.format("S =>") +
    '{:>5}'.format(parserlist[0]))
                for rules in range(len(parserlist)-1):
                    print('{:>10}'.format(" =>") +
    '{:>5}'.format(parserlist[rules+1]))
                break
        if j == (int(nlist[0])-1):
            print("String not Accepted")
```

Output:

```
Enter number of production rules: 4

Enter production rules:

S aBaA
S AB
A Bc
B c
The grammar is:
S  ->  aBaA | AB
A  ->  Bc
B  ->  c

Enter any string (0 for exit): acacc

String Accepted

We used LMD Top-Down approach

      S =>    S
        => aBaA
        => acaA
        => acaA
        =>acaBc
        =>acacc
        =>acacc

Enter any string (0 for exit): abca
String not Accepted

Enter any string (0 for exit): 0
```

Fig 5.3 Output for checking whether string is accepted by CFG or not

Frequently asked Questions:

1. **What is a CFG?**

   A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.

2. **What are the components of CFG?**

   A CFG consists of the following components:

   - a set of terminal symbols
   - a set of nonterminal symbols
   - a set of productions
   - a start symbol

3. **Give one example of a CFG.**

   $S \rightarrow AB$

   $S \rightarrow ASB$

   $A \rightarrow a$

   $B \rightarrow b$