Practical File

On

Compiler Design (CS 654)

File Submitted in Partial Fulfilment of

the Requirements for the Award of

Bachelor in Engineering

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted By

Vivsvaan Sharma

(Roll No: CO16362)

Under the supervision of

Dr. Gulshan Goyal

CHANDIGARH COLLEGE OF ENGINEERING AND TECHNOLOGY

(DEGREE WING)

Government Institute under Chandigarh (UT) Administration, Affiliated to Punjab University, Chandigarh

Sector-26 Chandigarh PIN-160019

Jan-June, 2019

ACKNOWLEDGEMENT

It is a great pleasure to present this practical file of Compiler Design. I have taken efforts in this Lab. However, it would not have been possible without the kind support and help of our teacher Dr. Gulshan Goyal. I would like to extend my sincere thanks to him.

I am highly indebted to Chandigarh College of Engineering & Technology (Degree Wing) for their guidance and constant supervision as well as for providing necessary information regarding the practical & also for their support in completing the practical file.

They taught me all the basic concepts required for the practical and guided me through each step of building the programs whenever I was stuck.

I would like to express my special gratitude and thanks to institution (C.C.E.T.) persons for giving me such attention and time.

I would also like to thank the University for including this Compiler Design Lab as a part of our curriculum.

INDEX

Date: 05-02-2019

Practical No 1

Aim: Introduction to Compiler Design.

1.1 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

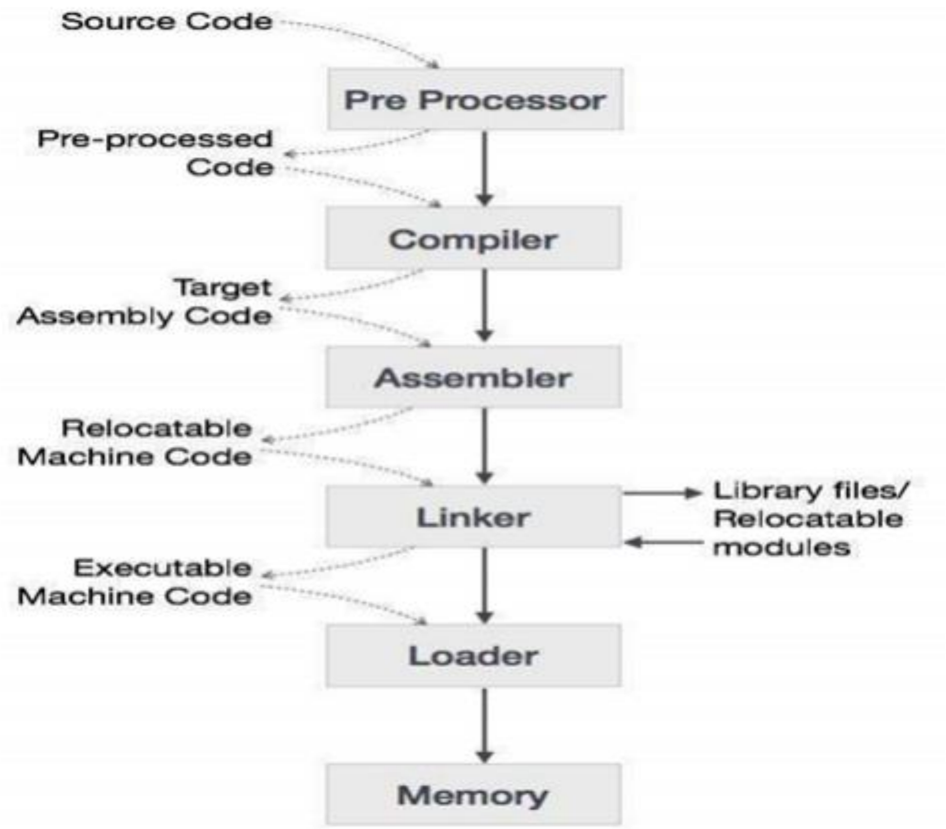


Fig 1.1- Language Processing System (courtesy- geeksfor geeks)

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.

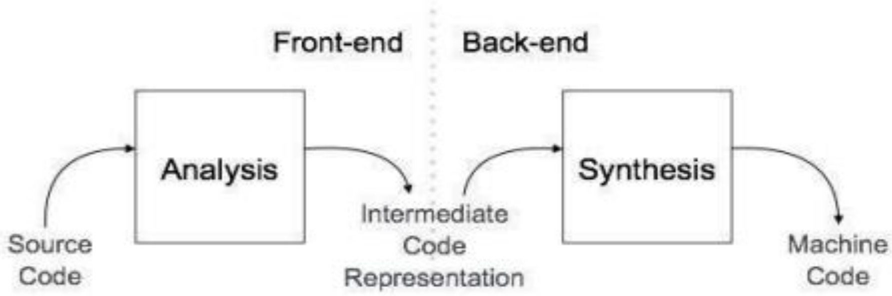


Fig 1.2- Front-end and Back-end of a compiler (courtesy- geeksfor geeks)

1.2 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.

1.3 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:

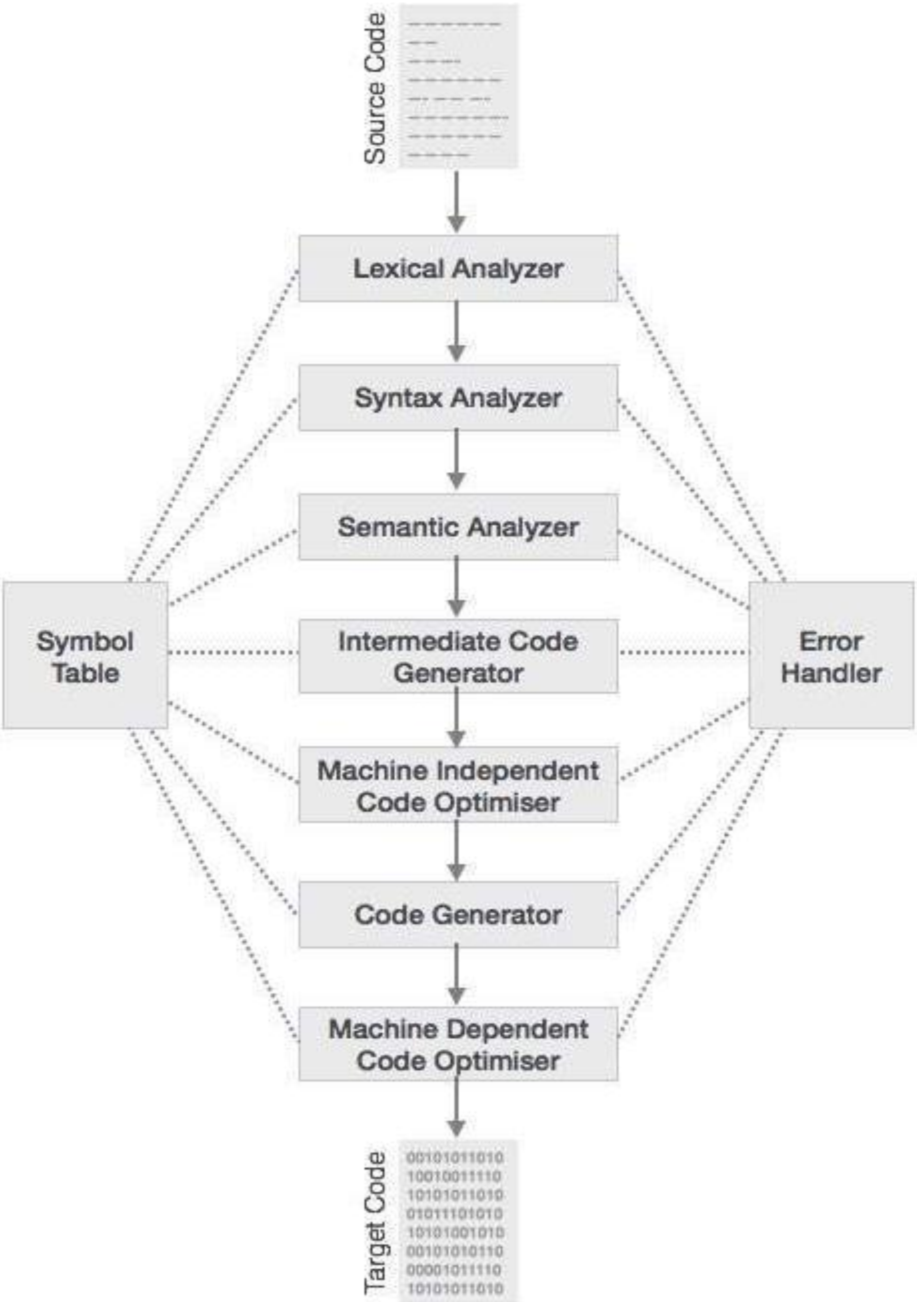


Fig 1.3 – Phases of Compiler (courtesy- geeksfor geeks)

a) 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 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.

b) 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. 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.

c) 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.

d) 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.

e) 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.

f) 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).

1.4 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.

1.5 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.

1.6 Example

1. Lexical Analysis:
   Input: stream of characters
   Output: Token

Token Template:

<token-name, attribute-value>

(e.g.) c=a+b*5;

Lexemes	**Tokens**
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

Table 1.1 Lexemes and tokens

Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

2. Syntax Analysis:

   Input: Tokens

   Output: Syntax tree

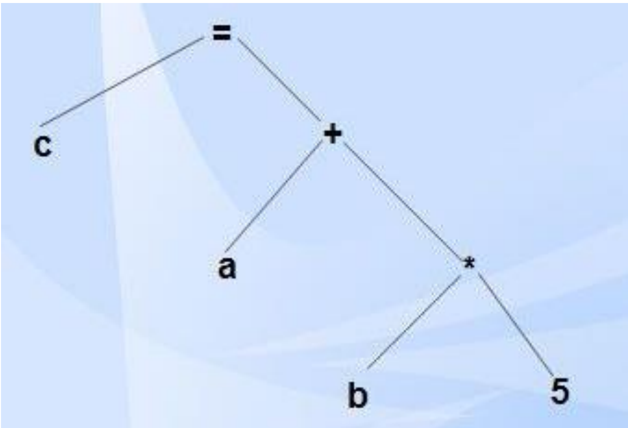


Fig. 1.5 (Courtesy: http://ecomputernotes.com/compiler-design/phases-of-compiler)

3. Intermediate Code generator

   Most commonly used form is the three address code.

   t_1 = inttofloat (5)

   t_2 = id_3* tl

   t_3 = id_2 + t_2

   id_1 = t_3

4. Code optimization

   t_1 = id_3* 5.0

   id_1 = id_2 + t_1

1.7 Frequently Asked Questions

a) Which program converts code from one language to another?

Ans. Translator.

b) Name three most commonly used translators.

Ans. Compiler, Interpreter and Assembler.

c) Which translator is used to generate machine code in assembly language?

Ans. Assembler.

d) Out of compiler and interpreter which one translates the code line by line?

Ans. Interpreter.

e) In case of which translator the object code is generated from the source code?

Ans. Compiler.

Date: 12-02-2019

Practical No. 2

Aim: Implement a program for constructing a transition table and recognizing the string 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**

2.1 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.

2.2 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

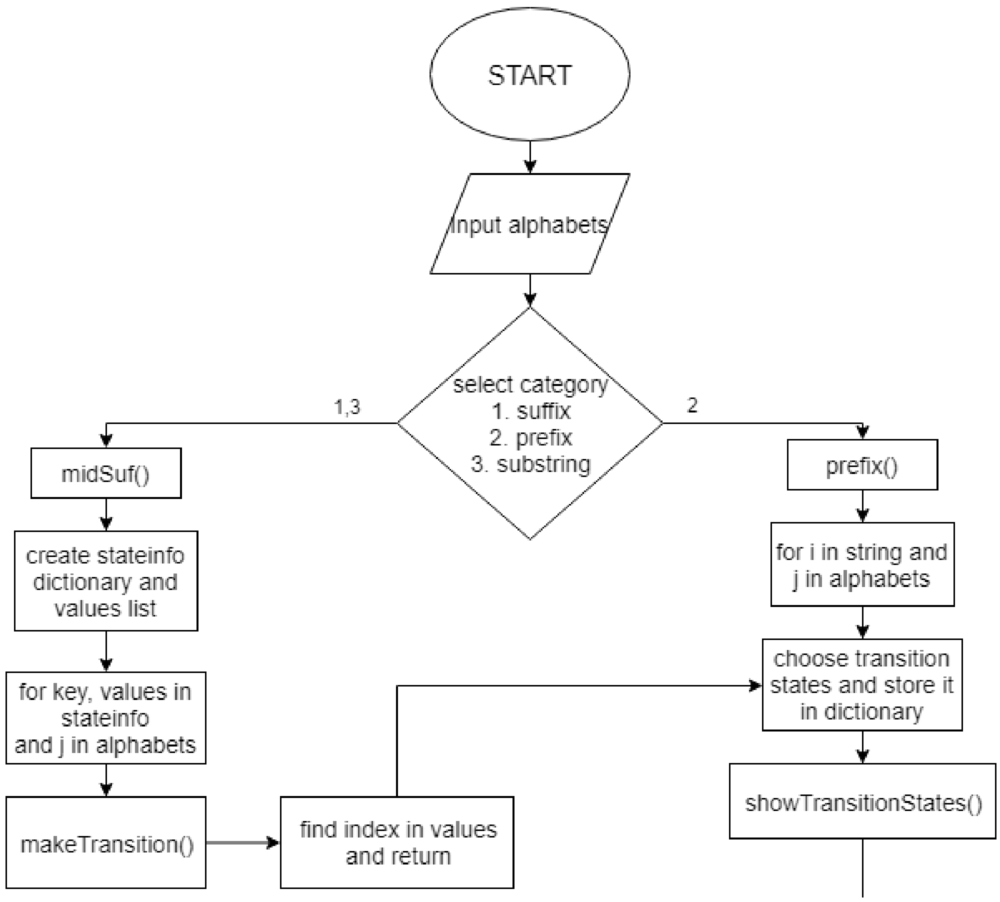
```
Input:
        1.      Read the alphabets
        2.      Read the choice for the type of DFA (i.e. suffix, prefix, substring)
        3.      Read the string for that particular DFA

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
```

2.3 Flowchart:



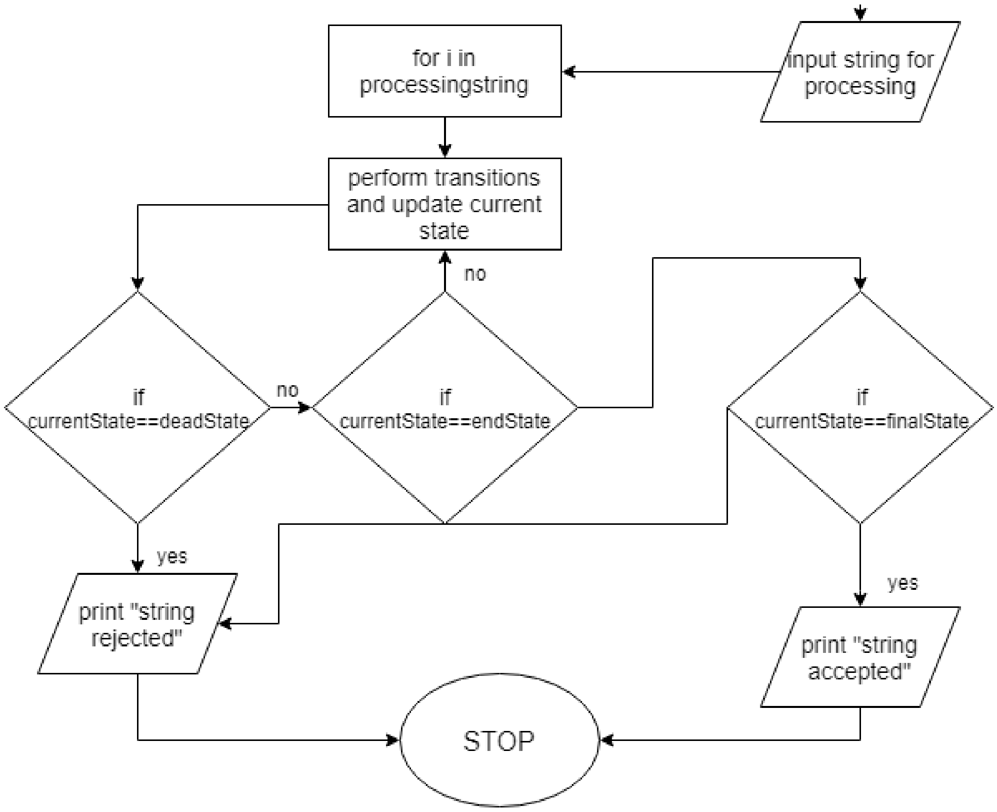


Fig 2.2 – Flowchart for algorithm 2.1

2.4 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)
```

2.5 Output

a) 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: abababaab

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

String accepted
```

Fig 2.3 Output for String Ending with suffix ab

b) 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

c) 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

2.6 Frequently Asked Questions

a) What is a string?

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

b) Which mathematical expression is used to define regular languages?

Ans. Regular expressions.

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

Ans. (a+b)* aaa (a+b)*

d) At max how many transitions are possible from a state for a given input symbol?

Ans. Three.

e) What is meant by equivalent FAs?

Ans. FAs that accept the same set of languages are called Equivalent FAs.

Date: 26-02-2019

Practical No. 3

Aim – To count single and multiple line comment in a given source code.

3.1 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.

3.2 Expected Output

The program should output the following:

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

Algorithm 3.1: 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
```

3.3 Flowchart

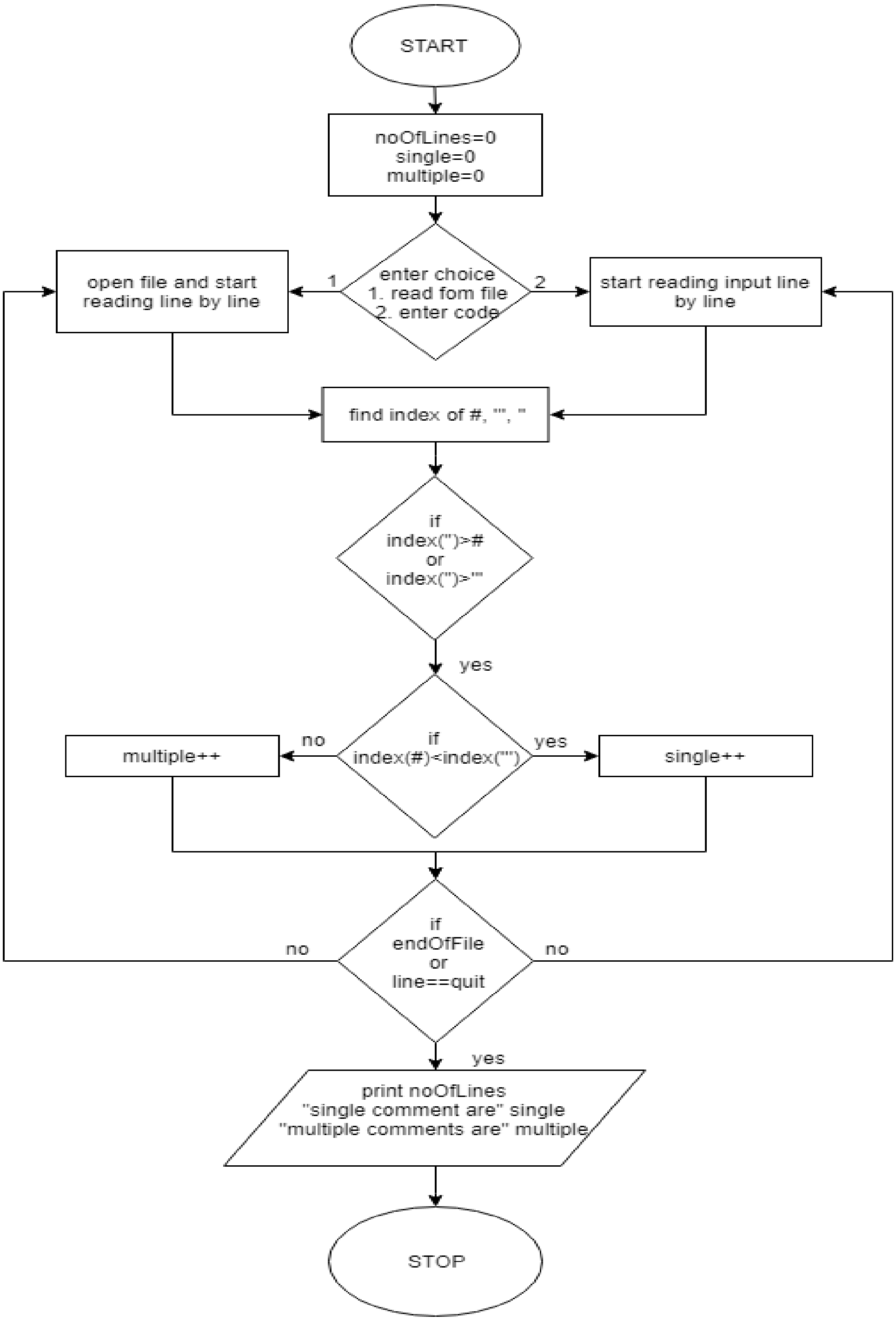


Fig 3.1 Flowchart for the algorithm 3.1

3.4 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))
```

3.5 Input File

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

print("#saasfd")
```

Fig 3.2 Input file containing code

3.6 Output

a) 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 3.3 Output for finding single amd multi-line comments from file

b) 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

3.7 Frequently Asked Questions

a) Which part in the code is skipped during the process of compilation and execution.

Ans. Comments are skipped during compilation and execution.

b) Give an example of Single line comments and multi-line comments.

Ans.
```
'''this is
    multi line comment'''
    #this is single line comment
```

c) How compiler treats comment lines.

Ans. The compiler treats a comment as a single white-space character.

d) Are comments case sensitive in python?

Ans. Comments used in code are not case sensitive.

e) Is it necessary to have comments for proper execution of the code?

Ans. Comments are not necessary but highly preferable.

Date: 05-03-2019

Practical No 4

Aim: To identify different tokens in a given source.

4.1 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.

4.2 Expected Output

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

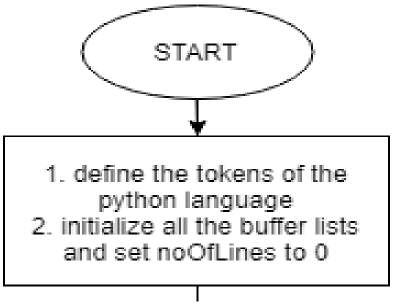
Algorithm 4.1: To construct a lexical analyzer for python language

```
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
```

4.3 Flowchart



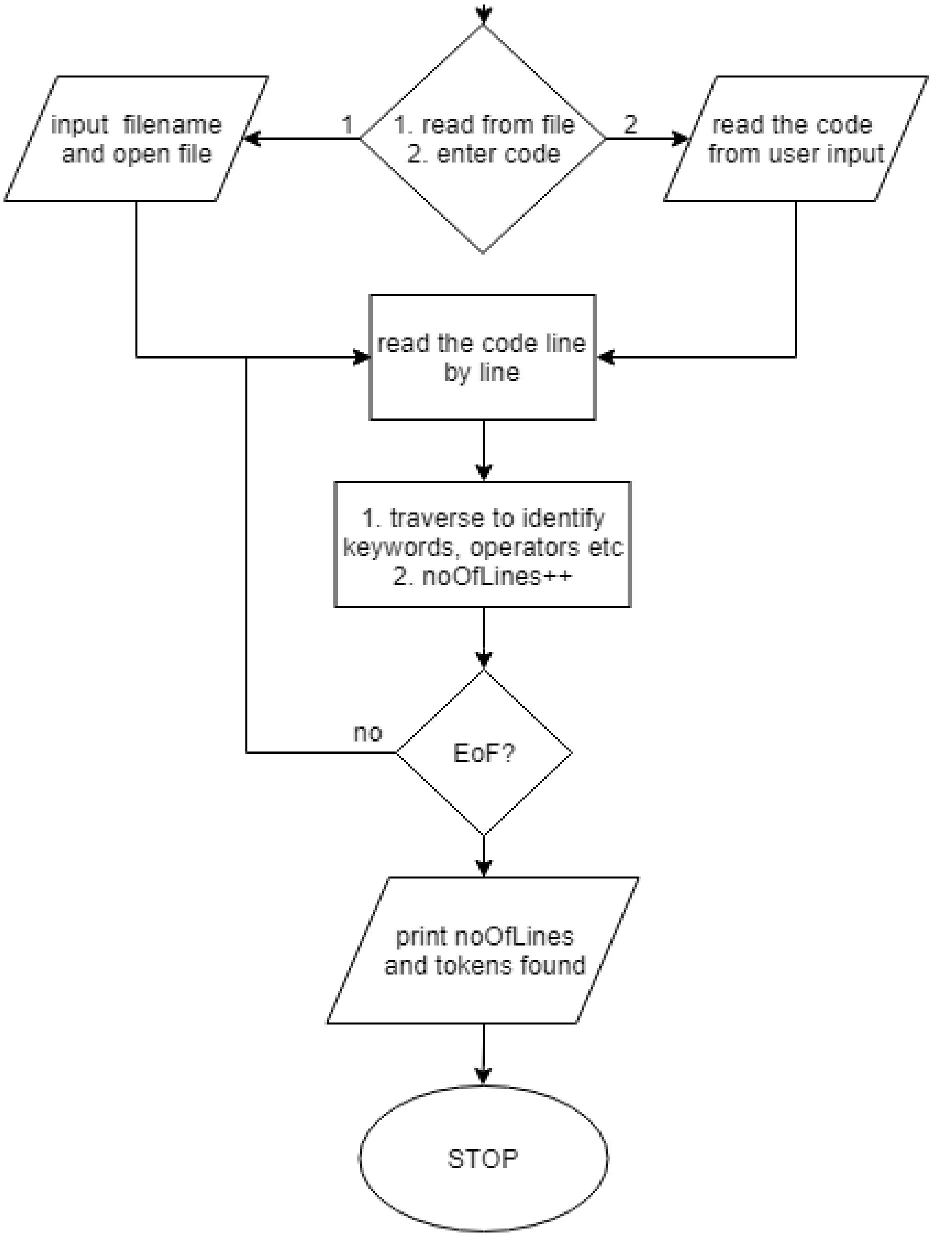


Fig 4.2 Flowchart for algorithm 4.1

4.4 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))
```

4.5 Input File

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

Fig 4.2 Input file containing code

4.6 Output

a) 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

b) 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

4.7 Frequently asked Questions

a) What is the output of lexical analysis phase?

Ans. Stream of tokens.

b) What is a token?

Ans. It is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

c) What does instance of token known as?

Ans. Lexeme.

d) The output of lexical analysis phase is further fed to which phase of compiler?

Ans. Syntax and semantic analyzer.

e) What are keywords?

Ans. Keywords are the tokens whose definition is known to the compiler

Date: 19-03-2019

Practical No 5

Aim: To take a CFG as input and parsing a given string.

5.1 Input

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

5.2 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
```

5.3 Flowchart

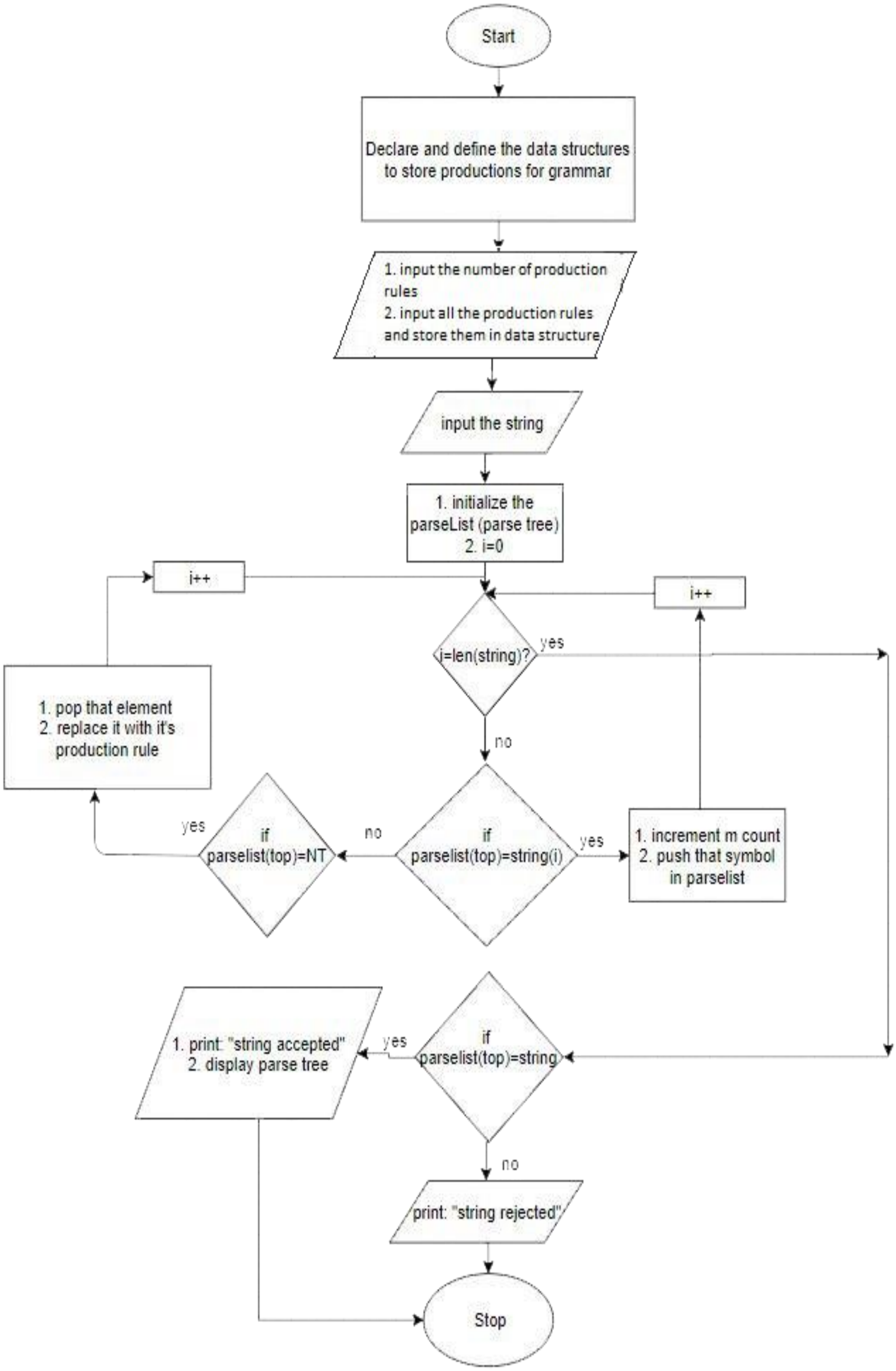


Fig 5.2 Flowchart for algorithm 5.1

5.4 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")
```

5.5 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

5.6 Frequently asked Questions

a) What is a CFG?

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

b) What is process of deriving string from the start symbol using the production rules of Grammar G?

Ans. String DerivationGive one example of a CFG.

c) Any language that can be generated using CFG can also be generated using regular expressions? (True/False)

Ans. False.

d) Which derivation is used in bottom up parsing?

Ans. RMD

e) Which derivation is used in top down parsing?

Ans. LMD

Date: 26-03-2019

Practical No 6

Aim: To write python program to implement LL Parser and recognize string.

6.1 Input

1. A file containing CFG.
2. A string to be processed.

6.2 Expected Output

The program should output:

1. Grammar production rules
2. FIRST and FOLLOW of each non-terminal.
3. Parsing table.
4. Sequence of moves made by the parser and print
    a. String is accepted by the parser or
    b. String is not accepted by the parser.

Algorithm 6.1: To make the LL parsing table and check if string is accepted

```
INPUT
      1.    A file containing CFG
      2.    A string to be processed
OUTPUT
      1.    FIRST and FOLLOW of each non-terminal
      2.    Parsing table
      3.    Sequence of moves made by the parser and print
            a.    String is accepted by parser, or
            b.    String is not accepted by the parser
main()
      1.    START
      2.    Store all production rules for a grammar in a file.
      3.    Declare and define a data structure to store all the
            production
            rules of the grammar
      4.    Compute FIRST and FOLLOW for each non-terminal
      5.    Prompt and input a string appended with $
      6.    start with symbol and parse the string
      7.    do until top!=$
                  let top be the top element and current_input be
                  the next input symbol
                  if top is a terminal or $
                        if top == a
                              pop top from stack and remove
                              current_input from input
```

```
                    else
                          print "string not accepted" and exit
              else
                    for the given input current_input, consult
                    the data to find the corresponding top-
                    production
                    if production rule is present
                          pop top from the stack
                          reverse the production and push it
                          onto stack
                    else
                          print "string not accepted" and exit
      8.    if top == $ and current_input == $
              print "String accepted"
            else
              print "String not accepted"
      9. END
```

Algorithm 6.2: To return the FIRST() of any non-terminal

```
INPUT
      A non-terminal α
OUTPUT
      list containing FIRST(α)
main()
      1.    START
      2.    if α is a terminal, then FIRST(α) = { α }
      3.    if α is non-terminal and α -> Ɛ is a production
                  then FIRST(α) = { Ɛ }
      4.    if α is a non-terminal and α → 𝜸1 𝜸2 𝜸3 … 𝜸n and
            any FIRST(𝜸) contains t
                  then t is in FIRST(α)
      5.    return FIRST(α)
      6.    STOP
```

Algorithm 6.3: To return the FOLLOW() of any non-terminal

```
INPUT
      A non-terminal α
OUTPUT
      list containing FIRST(α)
main()
      1.    START
      2.    if α is a start symbol, then FOLLOW(α) = $
      3.    if α is a non-terminal and has a production α → AB
                  then FIRST(B) is in FOLLOW(A) except Ɛ
      4.    if α is a non-terminal and has a production α → AB,
            where B Ɛ
                  then FOLLOW(A) is in FOLLOW(α).
      5.    return FOLLOW()
      6.    STOP
```

6.3 Flowchart

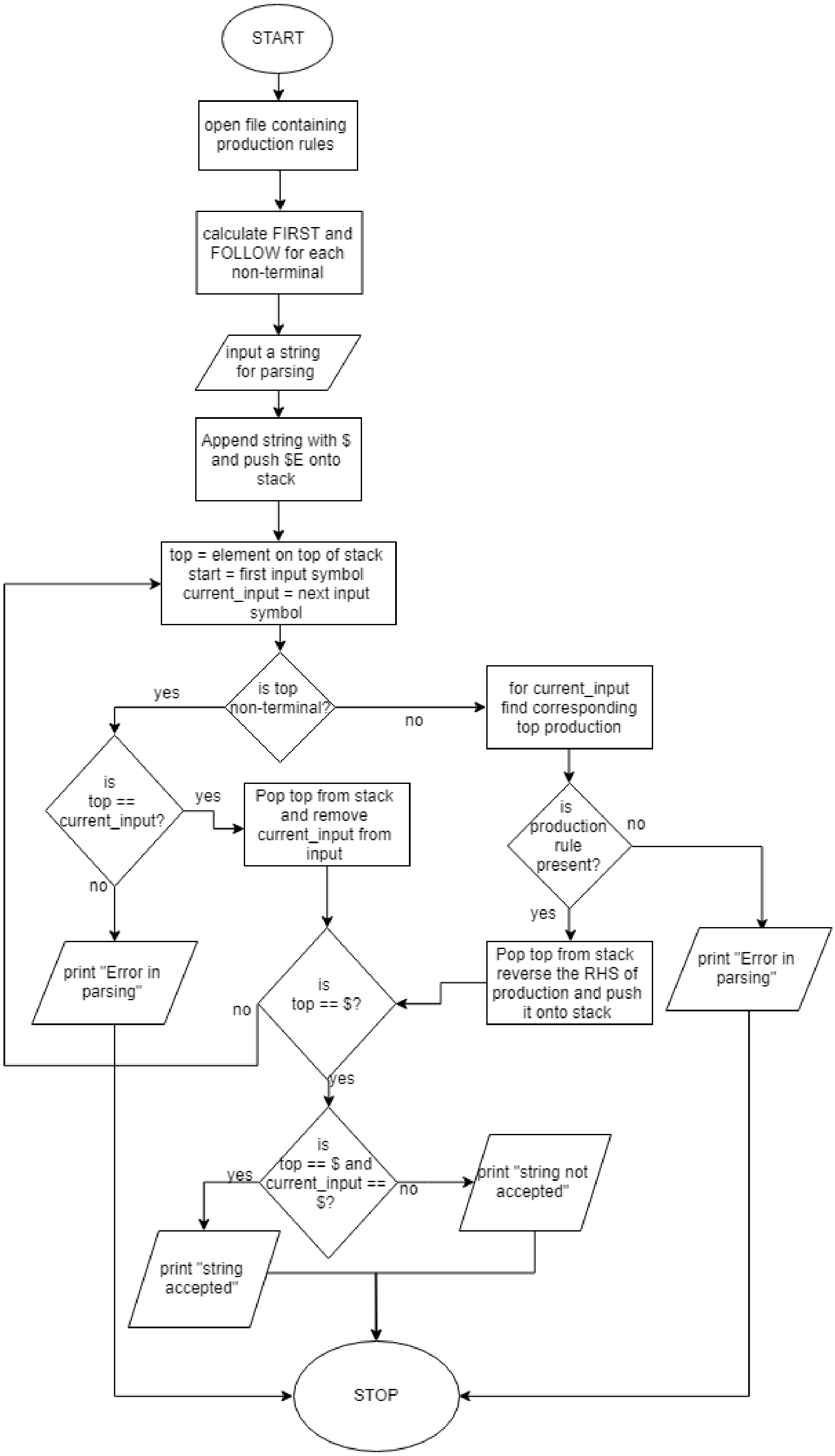


Fig 6.1 Flowchart for algorithm 6.1

6.4 Source Code

```
import re
import string
import pandas as pd
def parse(user_input,start_symbol,parsingTable):
    #flag
    flag = 0
    #appending dollar to end of input
    user_input = user_input + "$"
    stack = []
    stack.append("$")
    stack.append(start_symbol)
    input_len = len(user_input)
    index = 0
    while len(stack) > 0:
        #element at top of stack
        top = stack[len(stack)-1]
        print ("Top =>",top)
        #current input
        current_input = user_input[index]
        print ("Current_Input => ",current_input)
        if top == current_input:
            stack.pop()
            index = index + 1
        else:
            #finding value for key in table
            key = top , current_input
            print (key)
            #top of stack terminal => not accepted
            if key not in parsingTable:
                flag = 1
                break
            value = parsingTable[key]
            if value !='@':
                value = value[::-1]
                value = list(value)
                #poping top of stack
                stack.pop()
                #push value chars to stack
                for element in value:
                    stack.append(element)
            else:
                stack.pop()
    if flag == 0:
        print ("String accepted!")
    else:
        print ("String not accepted!")
def ll1(follow, productions):
    print ("\nParsing Table\n")
    table = {}
    for key in productions:
        for value in productions[key]:
            if value!='@':
```

```
                for element in first(value, productions):
                    table[key, element] = value
            else:
                for element in follow[key]:
                    table[key, element] = value
    for key,val in table.items():
        print (key,"=>",val)
    new_table = {}
    for pair in table:
        new_table[pair[1]] = {}
    for pair in table:
        new_table[pair[1]][pair[0]] = table[pair]
    print ("\n")
    print ("\nParsing Table in matrix form\n")
    print (pd.DataFrame(new_table).fillna('-'))
    print ("\n")
    return table
def follow(s, productions, ans):
    if len(s)!=1 :
        return {}
    for key in productions:
        for value in productions[key]:
            f = value.find(s)
            if f!=-1:
                if f==(len(value)-1):
                    if key!=s:
                        if key in ans:
                            temp = ans[key]
                        else:
                            ans = follow(key, productions, ans)
                            temp = ans[key]
                        ans[s] = ans[s].union(temp)
                else:
                    first_of_next = first(value[f+1:], productions)
                    if '@' in first_of_next:
                        if key!=s:
                            if key in ans:
                                temp = ans[key]
                            else:
                                ans = follow(key, productions, ans)
                                temp = ans[key]
                            ans[s] = ans[s].union(temp)
                            ans[s] = ans[s].union(first_of_next)-
                                    {'@'}
                    else:
                        ans[s] = ans[s].union(first_of_next)
    return ans
def first(s, productions):
    c = s[0]
    ans = set()
    if c.isupper():
        for st in productions[c]:
            if st == '@' :
                if len(s)!=1 :
```

```
                    ans = ans.union( first(s[1:], productions) )
                else :
                    ans = ans.union('@')
            else :
                f = first(st, productions)
                ans = ans.union(x for x in f)
    else:
        ans = ans.union(c)
    return ans
if __name__=="__main__":
    productions=dict()
    grammar = open("grammar", "r")
    first_dict = dict()
    follow_dict = dict()
    flag = 1
    start = ""
    print("\nGrammar is:\n")
    for line in grammar:
        print(line)
        l = re.split("( |->|\n|\||)*", line)
        lhs = l[0]
        rhs = set(l[1:-1])-{''}
        if flag :
            flag = 0
            start = lhs
        productions[lhs] = rhs
    print ('\nFirst\n')
    for lhs in productions:
        first_dict[lhs] = first(lhs, productions)
    for f in first_dict:
        print (str(f) + " : " + str(first_dict[f]))
    print ("")
    print ('\nFollow\n')

    for lhs in productions:
        follow_dict[lhs] = set()

    follow_dict[start] = follow_dict[start].union('$')

    for lhs in productions:
        follow_dict = follow(lhs, productions, follow_dict)

    for lhs in productions:
        follow_dict = follow(lhs, productions, follow_dict)

    for f in follow_dict:
        print (str(f) + " : " + str(follow_dict[f]))

    ll1Table = ll1(follow_dict, productions)

    string = str(input("Enter parsing string (in ''): "))
    print("\nParsing string: ",string," \n")
    parse(string,start,ll1Table)
```

6.5 Input File:

```
grammar.txt - Notepad
File  Edit  Format  View
S -> aA
A -> =B
B -> b+C
C -> c*d | (C)
```

Fig 6.2 Input file containing grammar production rules

6.6 Output

```
Grammar is:

S -> aA

A -> =B

B -> b+C

C -> c*d | (C)

First

A : set(['='])
S : set(['a'])
B : set(['b'])
C : set(['(', 'c'])

Follow

A : set(['$'])
S : set(['$'])
B : set(['$'])
C : set([')', '$'])
```

Fig 6.3 Output showing Grammar, FIRST and FOLLOW

```
Parsing Table

(('S', 'a'), '=>', 'aA')
(('C', 'c'), '=>', 'c*d')
(('B', 'b'), '=>', 'b+C')
(('C', '('), '=>', '(C)')
(('A', '='), '=>', '=B')



Parsing Table in matrix form

      (    =    a    b    c
A     -   =B    -    -    -
B     -    -    -   b+C   -
C    (C)   -    -    -   c*d
S     -    -   aA    -    -
```

Fig 6.4 Output showing Parsing table

```
Enter parsing string (in ''): 'a=b+(c*d)'
('\nParsing string: ', 'a=b+(c*d)', ' \n')
('Top =>', 'S')
('Current_Input => ', 'a')
('S', 'a')
('Top =>', 'a')
('Current_Input => ', 'a')
('Top =>', 'A')
('Current_Input => ', '=')
('A', '=')
('Top =>', '=')
('Current_Input => ', '=')
('Top =>', 'B')
('Current_Input => ', 'b')
('B', 'b')
('Top =>', 'b')
('Current_Input => ', 'b')
('Top =>', '+')
('Current_Input => ', '+')
('Top =>', 'C')
('Current_Input => ', '(')
('C', '(')
('Top =>', '(')
('Current_Input => ', '(')
('Top =>', 'C')
('Current_Input => ', 'c')
('C', 'c')
('Top =>', 'c')
('Current_Input => ', 'c')
('Top =>', '*')
('Current_Input => ', '*')
('Top =>', 'd')
('Current_Input => ', 'd')
('Top =>', ')')
('Current_Input => ', ')')
('Top =>', '$')
('Current_Input => ', '$')
String accepted!
```

Fig 6.5 Output showing each step while parsing string

6.7 Frequently Asked Questions

a) What is a Predictive Parser?

A predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

b) Define LL Parser.

Ans. An LL parser is a top-down parser for a subset of context-free languages. It parses the input from Left to right, performing Leftmost derivation of the sentence.

c) What is LL(1) grammar.

A grammar G is LL(1) if $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

I. For no terminal, both α and β derive strings beginning with a.

II. At most one of α and β can derive empty string.

III. If $\beta \stackrel{*}{\Longrightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A)

d) What are the different methods of top down parsing?

Ans. Backtracking, Predictive Parsing and Recursive Descent Parsing.

e) Another name for shift reduce parsing is?

Ans. LL parsing

Date: 02-04-2019

Practical No 7

Aim: To write python program to implement SLR Parser and recognize string.

7.1 Input

Enter the grammar production rules

7.2 Expected Output

After reading language grammar rules from text file, the grammar rules are displayed. Then a collection of sets of LR(0) items for input grammar is generated and displayed, then using this collection the SLR parsing table is generated and displayed.

Algorithm 7.1: To make the SLR parsing table and check if string is accepted

1. Construct the collection of sets of LR(0) items (C={I_0, I_1,…….., In}) for provided grammar (suppose G`).
   a. Apply **start** operation.
   b. Complete the state:
      i. Use one **read operation** on each item C (non-terminal or terminal) in the current state to create more states.
      ii. Apply the **complete operation** on the new states.
      iii. Repeat steps a and b until no more new states can be formed.
2. Construct the Parse table. (To create the parse table, you need the FIRST and FOLLOW sets for each non-terminal in your grammar.)

- **start**: If S is a symbol with [S -> w] as a production rule, then [S -> .w} is the item associated with the start state.
- **read:** if [A --> x.Cz] is an item in some state, then [A --> xC.z] is associated with some other state. When performing a read, all the items with the dot before the same C are associated with the same state. (Note that the dot is before anything, either terminal or non-terminal.)
- **complete:** if [A --> x.Xy] is an item, then every rule of the grammar with the form [X --> .z] must be included within this state. Repeat adding items until no new items can be added. (Note that the dot is before a non-terminal)).

7.3 Flowchart

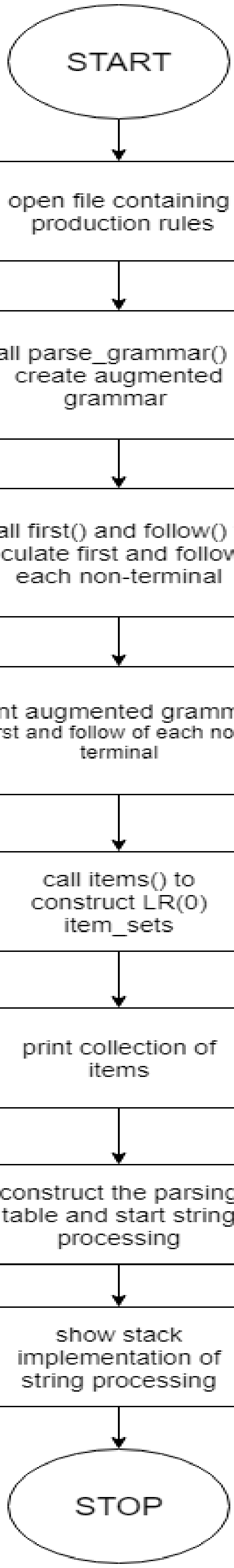


Fig 7.1 Flowchart for algorithm 7.1

7.4 Source Code

a) First_follow

```
from re import *
from collections import OrderedDict

t_list=OrderedDict()
nt_list=OrderedDict()
production_list=[]

class Terminal:

    def __init__(self, symbol):
        self.symbol=symbol

    def __str__(self):
        return self.symbol

class NonTerminal:

    def __init__(self, symbol):
        self.symbol=symbol
        self.first=set()
        self.follow=set()

    def __str__(self):
        return self.symbol

    def add_first(self, symbols): self.first |= set(symbols)

    def add_follow(self, symbols): self.follow |= set(symbols)

def compute_first(symbol):

    global production_list, nt_list, t_list

    if symbol in t_list:
        return set(symbol)

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

        if head!=symbol: continue

        if body=='':
            nt_list[symbol].add_first(chr(1013))
            continue

        if body[0]==symbol: continue

        for i, Y in enumerate(body):

            t=compute_first(Y)
```

```
            nt_list[symbol].add_first(t-set(chr(1013)))
            if chr(1013) not in t:
                break

            if i==len(body)-1:
                nt_list[symbol].add_first(chr(1013))

    return nt_list[symbol].first

def get_first(symbol):

    return compute_first(symbol)


def compute_follow(symbol):

    global production_list, nt_list, t_list

    if symbol == list(nt_list.keys())[0]:
        nt_list[symbol].add_follow('$')

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

        for i, B in enumerate(body):
            if B != symbol: continue

            if i != len(body)-1:
                nt_list[symbol].add_follow(get_first(body[i+1]) –
                  set(chr(1013)))

            if i == len(body)-1 or chr(1013) in get_first(body[i+1])
            and B != head:
                nt_list[symbol].add_follow(get_follow(head))

def get_follow(symbol):

    global nt_list, t_list

    if symbol in t_list.keys():
        return None

    return nt_list[symbol].follow

def main(pl=None):

    print('''Enter the grammar productions (enter 'end' or return to
            stop)
(Format: "A->Y1Y2..Yn" {Yi - single char} OR "A->" {epsilon})''')

    global production_list, t_list, nt_list
    ctr=1

    t_regex, nt_regex=r'[a-z\W]', r'[A-Z]'
```

```
    if pl==None:

        while True:

            production_list.append(input().replace(' ', ''))

            if production_list[-1].lower() in ['end', '']:
                del production_list[-1]
                break

            head, body=production_list[ctr-1].split('->')

            if head not in nt_list.keys():
                nt_list[head]=NonTerminal(head)

            for i in finditer(t_regex, body):
                s=i.group()
                if s not in t_list.keys(): t_list[s]=Terminal(s)

            for i in finditer(nt_regex, body):
                s=i.group()
                if s not in nt_list.keys():
                  nt_list[s]=NonTerminal(s)

            ctr+=1

    if pl!=None:

        for i, prod in enumerate(pl):

            if prod.lower() in ['end', '']:
                del pl[i:]
                break

            head, body=prod.split('->')

            if head not in nt_list.keys():
                nt_list[head]=NonTerminal(head)

            for i in finditer(t_regex, body):
                s=i.group()
                if s not in t_list.keys(): t_list[s]=Terminal(s)

            for i in finditer(nt_regex, body):
                s=i.group()
                if s not in nt_list.keys():
                  nt_list[s]=NonTerminal(s)

        return pl

if __name__=='__main__':

    main()
```

b) SLR

```
from collections import deque
from collections import OrderedDict
from pprint import pprint
import firstfollow
from firstfollow import production_list, nt_list as ntl, t_list as
                        tl
nt_list, t_list=[], []

class State:

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

def closure(items):

    global production_list

    while True:
        flag=0
        for i in items:

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

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

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

                if head!=Y: continue

                newitem=Y+'->.'+body

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

    return items

def goto(items, symbol):

    global production_list
    initial=[]

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

        head, body=i.split('->')
```

```
        seen, unseen=body.split('.')

        if unseen[0]==symbol and len(unseen) >= 1:
            initial.append(head+'->'+seen+unseen[0]+'.'+unseen[1:])

    return closure(initial)


def calc_states():

    def contains(states, t):

        for s in states:
            if sorted(s)==sorted(t): return True

        return False

    global production_list, nt_list, t_list

    head, body=production_list[0].split('->')
    states=[closure([head+'->.'+body])]

    while True:
        flag=0
        for s in states:
            for e in nt_list+t_list:
                t=goto(s, e)
                if t == [] or contains(states, t): continue

                states.append(t)
                flag=1

        if not flag: break

    return states

def make_table(states):

    global nt_list, t_list

    def getstateno(closure):

        for s in states:
            if sorted(s.closure)==sorted(closure): return s.no

    def getprodno(closure):

        closure=''.join(closure).replace('.', '')
        return production_list.index(closure)

    SLR_Table=OrderedDict()

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

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

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

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

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

                else: SLR_Table[s.no][nextsym] = str(getstateno(t))

    return SLR_Table

def augment_grammar():

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

def main():

    global production_list, ntl, nt_list, tl, t_list
```

```
    firstfollow.main()

    print("\tFIRST AND FOLLOW OF NON-TERMINALS")
    for nt in ntl:
        firstfollow.compute_first(nt)
        firstfollow.compute_follow(nt)
        print(nt)
        print("\tFirst:\t", firstfollow.get_first(nt))
        print("\tFollow:\t", firstfollow.get_follow(nt), "\n")


    augment_grammar()
    nt_list=list(ntl.keys())
    t_list=list(tl.keys()) + ['$']

    table=make_table(calc_states())

    print("\tSLR(1) TABLE\n")
    for i, j in table.items():
        print(i, "\t", j)

    return
if __name__=="__main__":
    main()
```

7.5 Output

```
GRAMMAR:
 A -> = B
 S -> a A
 B -> b + C
 C -> c * d | ( C )

AUGMENTED GRAMMAR:
0:  A -> = B
1:  S -> a A
2:  B -> b + C
3: S' -> S
4:  C -> c * d
5:  C -> ( C )

TERMINALS   : ['a', '=', 'b', '+', 'c', '*', 'd', '(', ')']
NONTERMINALS: ["S'", 'S', 'A', 'B', 'C']
SYMBOLS     : ['a', '=', 'b', '+', 'c', '*', 'd', '(', ')', "S'", 'S', 'A', 'B', 'C']

FIRST:
 A = { = }
 S = { a }
 B = { b }
S' = { a }
 C = { c ,  ( }

FOLLOW:
 A = { $ }
 S = { $ }
 B = { $ }
S' = { $ }
 C = { $ ,  ) }
```

Fig 7.2 Output showing Augmented Grammar, FIRST, FOLLOW

```
ITEMS:
I0:
 S -> . a A
S' -> . S

I1:
 A -> . = B
 S -> a . A

I2:
S' -> S .

I3:
 A -> = . B
 B -> . b + C

I4:
 S -> a A .

I5:
 B -> b . + C

I6:
 A -> = B .

I7:
 C -> . c * d
 C -> . ( C )
 B -> b + . C

I8:
 C -> c . * d

I9:
 C -> ( . C )
 C -> . c * d
 C -> . ( C )

I10:
 B -> b + C .

I11:
 C -> ( C . )

I12:
 C -> c * . d

I13:
 C -> ( C ) .

I14:
 C -> c * d .
```

Fig 7.3 Output showing Items I_0 to I_{14}

PARSING TABLE:

STATE	ACTION										GOTO			
	a	=	b	+	c	*	d	(	)	$	S	A	B	C
0	s1										2			
1		s3										4		
2										acc				
3			s5										6	
4										r1				
5				s7										
6										r0				
7					s8			s9						10
8						s12								
9					s8			s9						11
10										r2				
11									s13					
12							s14							
13									r5	r5				
14									r4	r4				

Fig 7.4 Output showing Parsing Table

```
Enter Input: a = b + ( c * d )

+---------+----------------------------+----------------------------+-----------+
|  STEP   |           STACK            |           INPUT            |  ACTION   |
+---------+----------------------------+----------------------------+-----------+
|    1    | 0                          |              a=b+(c*d)$    |    s1     |
|    2    | 0a1                        |               =b+(c*d)$    |    s3     |
|    3    | 0a1=3                      |                b+(c*d)$    |    s5     |
|    4    | 0a1=3b5                    |                 +(c*d)$    |    s7     |
|    5    | 0a1=3b5+7                  |                  (c*d)$    |    s9     |
|    6    | 0a1=3b5+7(9                |                   c*d)$    |    s8     |
|    7    | 0a1=3b5+7(9c8              |                    *d)$    |    s12    |
|    8    | 0a1=3b5+7(9c8*12           |                     d)$    |    s14    |
|    9    | 0a1=3b5+7(9c8*12d14        |                      )$    |    r4     |
|   10    | 0a1=3b5+7(9C11             |                      )$    |    s13    |
|   11    | 0a1=3b5+7(9C11)13          |                       $    |    r5     |
|   12    | 0a1=3b5+7C10               |                       $    |    r2     |
|   13    | 0a1=3B6                    |                       $    |    r0     |
|   14    | 0a1A4                      |                       $    |    r1     |
|   15    | 0S2                        |                       $    | ACCEPTED  |
+---------+----------------------------+----------------------------+-----------+
```

Fig 7.5 Output showing parsing of string

7.6 Frequently Asked Questions

a) What is SLR parser?

Ans. SLR stands for simple LR parser.

b) Which string derivation is used in SLR parser?

Ans. RMD.

c) SLR is bottom up parser. (True/False)

Ans. True.

d) Fill in the blank.

In SLR parsing canonical collection of ______ items are formed.

Ans. LR(0)

e) What is an augmented grammar?

Ans. Augmented grammar is the grammar having the augmented production rule, which is of the form S’-> S. Where S is the start symbol of original grammar G.

Date: 09-04-2019

Practical No 8

Aim: To write python program to implement CLR parser.

8.1 Input

Input grammar productions rules.

8.2 Expected Output

The program should print

3. FIRST and FOLLOW of each non-terminal and Items (I_0, I_1, I_2 …)
4. Print number of r/r conflicts and s/r conflicts
5. Parsing Table

Algorithm 8.1: To implement CLR parser

```
1. START
2. Input a grammar.
3. For each of the non-terminal.
    a. Compute FIRST and FOLLOW set.
4. For each of the non-terminal.
    a. Print FIRST and FOLLOW sets.
5. Compute the set of LR(1) items.
6. Print canonical collection of LR(1) items.
7. Construct the CLR parsing table.
8. Print the CLR parsing table.
9. STOP
```

Algorithm 8.2: To compute closure

closure(I): performs closure operation on a set of Items(I) from a grammar G

```
1.  closure(I){  repeat
        for each item [A -> α. Bβ, a] in I, each production B->γ
        in G', and each terminal b in FIRST(βa) such that [B->.
        γ, b] is not in I do
              add [B->. γ, b] to I
        until no more sets of items can be added to I
2. return I
```

Algorithm 8.3: To compute GOTO

goto(I,X): performs goto operation for a given item I and a grammar symbol X.

```
1. goto(I, X){
        let J be the set of items [A-> α X. β, a] such that [A->
        α .X β, a] is in I;
        return closure(J)
```

```
}
```

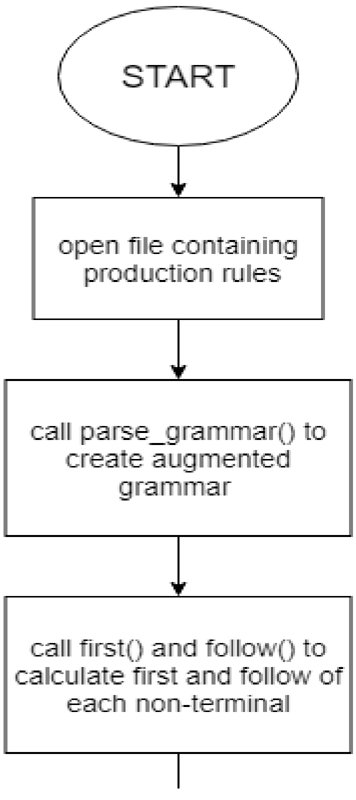
Algorithm 8.4: To compute LR(1) item set

items(G‟): This procedure returns canonical collection of sets of LR(1) items for an augmented grammar G‟.

```
items(G') {
      C := {closure({S'->. S,$})};
      repeat
            for each set of items I in C and each grammar symbol X
      such that goto(I , X) is not empty and not in C do
                  add goto(I , X) to C
            until no more sets of items can be added to C
}
```

1. The goto transition for state i are determined as follows:
   a. If goto(Ii , A)= Ij then,
      i. goto[i,A]=j.
2. All entries not defined by rules (2) and (3) are made “error.”
3. The initial state of the parser is the one constructed from the set containing item [S‟->.S, $].

8.3 Flowchart



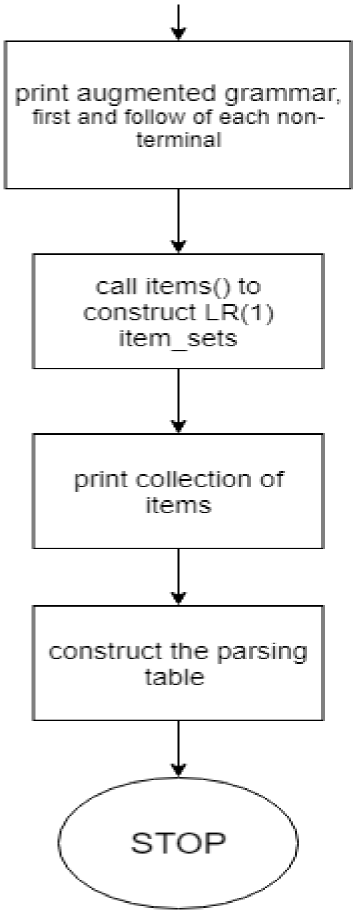


Fig 8.1 Flowchart for algorithm 8.1

8.4 Source Code

```
from collections import deque
from collections import OrderedDict
from pprint import pprint
import firstfollow
from firstfollow import production_list, nt_list as ntl, t_list as tl
nt_list, t_list=[], []

class State:

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

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

    def __str__(self):
```

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


def closure(items):

    def exists(newitem, items):

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

    global production_list

    while True:
        flag=0
        for i in items:

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

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

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

            else:
                lastr=i.lookahead

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

                if head!=Y: continue

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

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

def goto(items, symbol):

    global production_list
    initial=[]
    for i in items:
        if i.index('.')==len(i)-1: continue

        head, body=i.split('->')
        seen, unseen=body.split('.')

        if unseen[0]==symbol and len(unseen) >= 1:
            initial.append(Item(head+'->'+seen+unseen[0]+'.'+unseen[1:],
i.lookahead))

    return closure(initial)
```

```
def calc_states():

    def contains(states, t):

        for s in states:
            if len(s) != len(t): continue
            if sorted(s)==sorted(t):
                for i in range(len(s)):
                        if s[i].lookahead!=t[i].lookahead: break
                else: return True

        return False

    global production_list, nt_list, t_list
    head, body=production_list[0].split('->')
    states=[closure([Item(head+'->.'+body, ['$'])])]
    while True:
        flag=0
        for s in states:
            for e in nt_list+t_list:
                t=goto(s, e)
                if t == [] or contains(states, t): continue
                states.append(t)
                flag=1

        if not flag: break
    return states

def make_table(states):
    global nt_list, t_list
    def getstateno(t):
        for s in states:
            if len(s.closure) != len(t): continue
            if sorted(s.closure)==sorted(t):
                for i in range(len(s.closure)):
                        if s.closure[i].lookahead!=t[i].lookahead: break
                else: return s.no
        return -1

    def getprodno(closure):

        closure=''.join(closure).replace('.', '')
        return production_list.index(closure)

    SLR_Table=OrderedDict()

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

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

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

```
                    else: SLR_Table[s.no][term] |=
{'r'+str(getprodno(item))}
                continue

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

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

                else: SLR_Table[s.no][nextsym] = str(getstateno(t))

    return SLR_Table

def augment_grammar():

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

def main():

    global production_list, ntl, nt_list, tl, t_list
    firstfollow.main()
    print("\tFIRST AND FOLLOW OF NON-TERMINALS")
    for nt in ntl:
        firstfollow.compute_first(nt)
        firstfollow.compute_follow(nt)
        print(nt)
        print("\tFirst:\t", firstfollow.get_first(nt))
        print("\tFollow:\t", firstfollow.get_follow(nt), "\n")

    augment_grammar()
    nt_list=list(ntl.keys())
    t_list=list(tl.keys()) + ['$']

    print(nt_list)
    print(t_list)
    j=calc_states()
    ctr=0
```

```
    for s in j:
        print("I{}:".format(ctr))
        for i in s:
            print("\t", i)
        ctr+=1
    table=make_table(j)
    print("\n\tCLR(1) TABLE\n")
    sr, rr=0, 0
    tnt_list = t_list + nt_list
    parsing = OrderedDict()
    for i in tnt_list:
        parsing.update({i:" "})
    parser_list = []
    for i, j in table.items():
        # print(i, "\t", j)
        s, r=0, 0
        for key,val in j.items():
            parsing[key] = val
        parser_list.append(parsing)
        parsing = OrderedDict()
        for i in tnt_list:
            parsing.update({i:" "})
        for p in j.values():
            if p!='accept' and len(p)>1:
                p=list(p)
                if('r' in p[0]): r+=1
                else: s+=1
                if('r' in p[1]): r+=1
                else: s+=1
        if r>0 and s>0: sr+=1
        elif r>0: rr+=1
    # print("Parser_list")
    # for i in parser_list:
    #   print(i)
    print("\n", sr, "s/r conflicts |", rr, "r/r conflicts")
    terminals = t_list
    nonterminals = nt_list
    print ("\nPARSING TABLE:\n")
    print ("+" + "-------+" * (len(terminals) + len(nonterminals) + 1))
    print ("|{:^7}|".format('STATE'), end="")
    print ("{:^79}".format('ACTION'),end="")
    print ("|",end="")
    print ("{:^31}".format('GOTO'),end="")
    print ("|")
    print ("+" + "-------+" * (len(terminals) + len(nonterminals) + 1))
    print ("|{:^7}|".format(' '),end="")
    for terms in terminals:
        print ("{:^7}|".format(terms),end="")
    for nonterms in nonterminals:
        print ("{:^7}|".format(nonterms),end="")
    print ("\n+" + "-------+" * (len(terminals) + len(nonterminals) + 1))
    for i in range(len(parser_list)):
        print ("|{:^7}|".format(i),end="")
        for key,val in parser_list[i].items():
             val = list(val)
             print ("{:^7}|".format(val[0]),end="")
        print()
    print ("+" + "-------+" * (len(terminals) + len(nonterminals) + 1))
    return
```

```
if __name__=="__main__":
    main()
```

8.5 Output

```
Enter the grammar productions (enter 'end' or return to stop)
(Format: "A->Y1Y2..Yn" {Yi - single char} OR "A->" {epsilon})
S->aA
A->-B
B->b+C
C->c*d
C->(C)
end
        FIRST AND FOLLOW OF NON-TERMINALS
S
        First:  {'a'}
        Follow: {'$'}

A
        First:  {'-'}
        Follow: {'$'}

B
        First:  {'b'}
        Follow: {'$'}

C
        First:  {'c', '('}
        Follow: {')', '$'}

['S', 'A', 'B', 'C']
['a', '-', 'b', '+', 'c', '*', 'd', '(', ')', '$']
```

Fig 8.2 Printing Grammar, FIRST and FOLLOW and set of non-terminals, terminals

```
I0:
        Z->.S, $
        S->.aA, $
I1:
        Z->S., $
I2:
        S->a.A, $
        A->.-B, $
I3:
        S->aA., $
I4:
        A->-.B, $
        B->.b+C, $
I5:
        A->-B., $
I6:
        B->b.+C, $
I7:
        B->b+.C, $
        C->.c*d, $
        C->.(C), $
I8:
        B->b+C., $
I9:
        C->c.*d, $
I10:
        C->(.C), $
        C->.c*d, )
        C->.(C), )
I11:
        C->c*.d, $
I12:
        C->(C.), $
I13:
        C->c.*d, )
I14:
        C->(.C), )
        C->.c*d, )
        C->.(C), )
I15:
        C->c*d., $
I16:
        C->(C)., $
I17:
        C->c*.d, )
I18:
        C->(C.), )
I19:
        C->c*d., )
I20:
        C->(C)., )
```

Fig 8.3 Item sets

```
0 s/r conflicts | 0 r/r conflicts

PARSING TABLE:
```

STATE	ACTION										GOTO			
	a	-	b	+	c	*	d	(	)	$	S	A	B	C
0	s2										1			
1										a				
2		s4										3		
3										r1				
4			s6										5	
5										r2				
6				s7										
7					s9			s10						8
8										r3				
9						s11								
10					s13			s14						1
11							s15							
12									s16					
13						s17								
14					s13			s14						1
15										r4				
16										r5				
17							s19							
18									s20					
19									r4					
20									r5					

Fig 8.4 Parsing table

8.6 Frequently asked Questions

a) What is CLR parser?
Ans. CLR stands for canonical LR parser.

b) In CLR parsing canonical collection of ______ items are formed.
Ans. LR(1)

c) Item sets having different look ahead symbols are same in CLR. (True/False)
Ans. False.

d) Which is the most powerful parser among SLR, CLR and LALR?
Ans. CLR parser.

e) What is the number of look-ahead symbols used by CLR parser?
Ans. 1

Date: 16-04-2019

Practical No 9

Aim: To write python program to implement LALR parser.

9.1 Input

Define the grammar production rules in grammar.py file

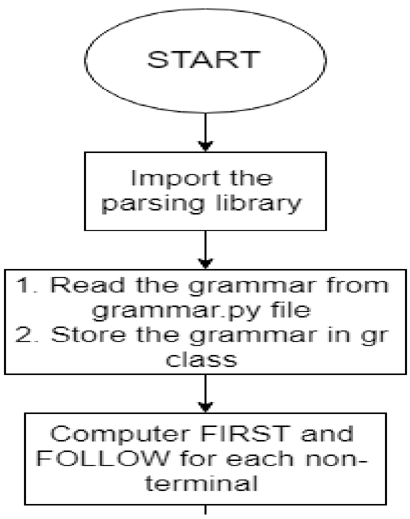
9.2 Expected Output

The program should print Parsing Table

Algorithm 9.1: To implement LALR parser

10. START
11. Import the parsing library/files.
12. The parsing files contains lalr_one file which is used for generating lalr_one parsing table
13. Read the grammar from grammar.py file.
14. Store the grammar production rules in gr class
15. gr.production has all the productions, gr.nonterms has all the non-terminals and gr.terminals has all the terminals.
16. For each of the non-terminal.
a. Compute FIRST and FOLLOW set.
b. Construct the LALR parsing table using lalr_one function.
17. Describe the grammar (i.e. how many productions, terminals and non-terminals it contains)
18. Get the conflict status
19. Store the grammar description and LALR table description in parsing-table.txt
20. Print the LALR parsing table.
21. STOP

9.3 Flowchart



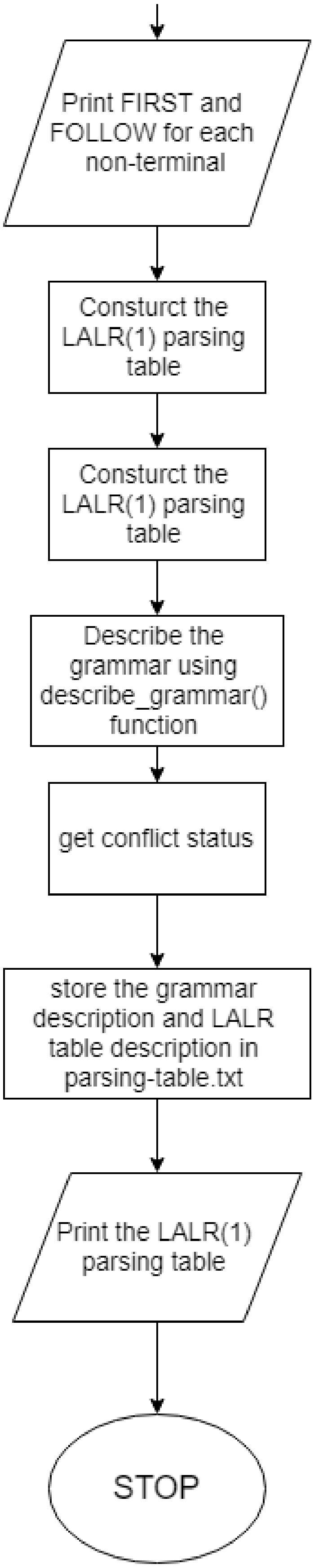


Fig 9.1 Flowchart for algorithm 9.1

9.4 Source Code

```
from parsing import *
import grammar
import pandas as pd

def get_grammar():
    return grammar.get_sample_1()

def describe_grammar(gr):
    return '\n'.join([
        'Indexed grammar rules (%d in total):' % len(gr.productions),
        str(gr) + '\n',
        'Grammar non-terminals (%d in total):' % len(gr.nonterms),
        '\n'.join('\t' + str(s) for s in gr.nonterms) + '\n',
        'Grammar terminals (%d in total):' % len(gr.terminals),
        '\n'.join('\t' + str(s) for s in gr.terminals)
    ])

def describe_parsing_table(table):
    conflict_status = table.get_conflict_status()
    print(conflict_status)
    def conflict_status_str(state_id):
        has_sr_conflict = (conflict_status[state_id] ==
lalr_one.STATUS_SR_CONFLICT)
        status_str = ('shift-reduce' if has_sr_conflict else 'reduce-
reduce')
        return 'State %d has a %s conflict' % (state_id, status_str)

    return ''.join([
        'PARSING TABLE SUMMARY\n',
        'Is the given grammar LALR(1)? %s\n' % ('Yes' if
table.is_lalr_one() else 'No'),
        ''.join(conflict_status_str(sid) + '\n' for sid in
range(table.n_states)
                if conflict_status[sid] != lalr_one.STATUS_OK) + '\n',
        table.stringify()
    ])

def main():
    print('Reading Grammar Production Rules...')
    print('Making Parsing Table...')
    gr = get_grammar()
    table = lalr_one.ParsingTable(gr)
    print("Done\n")
    output_filename = 'parsing-table'

    with open(output_filename + '.txt', 'w') as textfile:
        textfile.write(describe_grammar(gr))
        textfile.write('\n\n')
        textfile.write(describe_parsing_table(table))

    table.save_to_csv(output_filename + '.csv')

    print("Parsing table is \n")
    parsing_table = pd.read_csv(output_filename + '.csv')
    header = []
```

```
    for i in parsing_table:
        header.append(i)

    parse_table = parsing_table.iloc[:,:].values
    for i in range(len(parse_table)):
        for j in range(len(header)):
            if str(parse_table[i][j]) == "nan":
                parse_table[i][j] = "  "
    print('{:^5}|'.format("state") + '{:^59}|'.format("action") +
'{:^23}|'.format("goto"))
    for i in header:
        print('{:^5}|'.format(i),end="")
    print()
    for i in range(len(parse_table)):
        for j in range(len(header)):
            print('{:^5}|'.format(parse_table[i][j]), end="")
        print()
if __name__ == "__main__":
    main()
```

9.5 Output

```
Reading Grammar Production Rules...
Indexed grammar rules (6 in total):
0    $accept: S
1    A: '=' B
2    B: b '+' C
3    C: c '*' d
4      | '(' C ')'
5    S: a A

Grammar non-terminals (5 in total):
        $accept
        A
        B
        C
        S

Grammar terminals (9 in total):
        '('
        ')'
        '*'
        '+'
        '='
        a
        b
        c
        d
```

Fig 9.2 Description of the Grammar

```
conflict status:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Parsing table is
```

state	action										goto			
	'('	')'	'*'	'+'	'='	a	b	c	d	$end	A	B	C	S
0						s2								1.0
1										a				
2					s4						3.0			
3										r5				
4							s6					5.0		
5										r1				
6				s7										
7	s9							s10					8.0	
8										r2				
9	s9							s10					11.0	
10			s12											
11		s13												
12									s14					
13		r4								r4				
14		r3								r3				

Fig 9.3 Conflict Status and LALR(1) Parsing Table

9.6 **Frequently asked Questions**

a) What is LALR parser?

Ans. LALR stands for look ahead LR parser.

b) In LALR parsing canonical collection of ______ items are formed.

Ans. LALR(1)

c) Item sets having different look ahead symbols are same in LALR. (True/False)

Ans. True. They are merged to create a single item set.

d) What are the two functions used in creating canonical collection of items in LR parsers?

Ans. GOTO() and CLOSURE().

e) How many types of LR parsers are there? Name them.

Ans. There are basically three types of LR parsers, SLR, CLR and LALR