NAME: SUNIL.DIBAKAR.PANIGRAHI

ROLL NUMBER: 509

COURSE: MSc CS

SUBJECT: DESIGN AND IMPLEMENTATION OF MODERN COMPILERS

PRACTICAL: 1-8

SR No	Name	Date	Sign
1	Tokenizing a file.		
2	Implementation of lexical analyser using Lex tool.		
3	Study the Lex and Yacc tool and evaluate an arithmetic expression with parentheses, unary and binary operators using Flex and Yacc (calculator).		
4	Using JFLAP, create a DFA from a given regular expression.		
5	Create LL(1) parse table for a given CFG and hence simulate LL(1) parsing.		
6	Using JFLAP, create SLR(1) parse table for a given grammar. Simulate parsing and output the parse tree in proper format.		
7	Write functions to find FIRST and FOLLOW of all the variables.		
8	Using JFLAP, create SLR(1) parse table for a given grammar. Simulate parsing and output the parse tree in proper format.		

Hardware and Software Requirement

1. Software Requirement:

Turbo C / C++ compiler.

Download from following

links:

http://www.megaleecher.net/Download_Turbo_For_Windows

LEX tool--flex-2.5.4a-1.exe

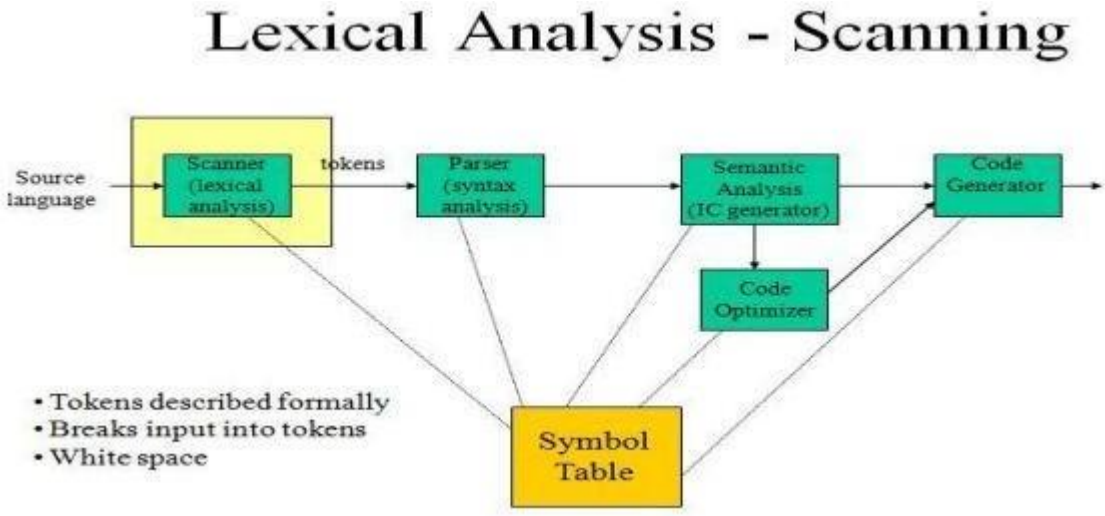
YACC tool--bison-2.4.1-setup.exe

JFLAP Tool for Automata--www.JFLAP.org

Practical 1: Tokenizing a file.

Aim: (Tokenizing). A program that reads a source code in C/C++ from an unformatted file andextract various types of tokens from it (e.g. keywords/variable names, operators, constant values).

Description: Lexical analysis is the process of converting a sequence of characters (such as in acomputer program of web page) into a sequence of tokens (strings with an identified
"meaning"). A program that perform lexical analysis may be called a lexer, tokenize or scanner.



Token

A token is a structure representing a lexeme that explicitly indicates its categorization for the Purpose of parsing. A category of token is what in linguistics might be called a part-of-speech. Examples of token categories may include "identifier" and "integer literal", although the set of Token differ in different programming languages. The process of forming tokens from an input stream of characters is called tokenization. Consider this expression in the C programming language:Sum=3 + 2;

Tokenized and represented by the following table:

Lexeme	Token category
Sum	“identifier”
=	“assignment operator”
3	“integer literal”
+	“addition operator”
2	“integer literal”
;	“end of the statement”

Source code:

```
import nltk
 # import RegexpTokenizer() method from nltk
 from nltk.tokenize import RegexpTokenizer
# Create a reference variable for Class RegexpTokenize
r tk = RegexpTokenizer('\s+', gaps = True)
# Create a string input str = "I love to study Compiler code in Python"
# Use tokenize method
 tokens = tk.tokenize(str)
print(tokens)
```

Practical 2: Implementation of lexical analyser using Lex tool.

Aim: (Tokenizing) Use Lex and yacc to extract tokens from a given source code.

Description:

- A language for specifying lexical analyzer.
- There is a wide range of tools for construction of lexical analyzer. The majority of these tools arebased on regular expressions.
- The one of the traditional tools of that kind is lex.

Lex:-

- The lex is used in the manner depicted. A specification of the lexical analyzer is preferred by creating a program lex.1 in the lex language.
- Then lex.1 is run through the lex compiler to produce a ‘c’ program lex.yy.c.
- The program lex.yy.c consists of a tabular representation of a transition diagram constructed fromthe regular expression of lex.1 together with a standard routine that uses table of recognize leximes.
- Lex.yy.c is run through the ‘C’ compiler to produce as object program a.out, which is the lexical
  analyzer that transforms as input stream into sequence of tokens.

Algorithm:

1. First, a specification of a lexical analyzer is prepared by creating a program lexp.l in the LEX language.
2. The Lexp.l program is run through the LEX compiler to produce an equivalent code inC language named Lex.yy.c
3. The program lex.yy.c consists of a table constructed from the Regular Expressions ofLexp.l, together with standard routines that uses the table to recognize lexemes.
4. Finally, lex.yy.c program is run through the C Compiler to produce an object programa.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

Program

```
lexp.l
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf ("\n %s is a Preprocessor Directive",yytext);} int |
float | main | if | else | printf | scanf | for | char | getch |
while {printf("\n %s is a Keyword",yytext);} "/*" {COMMENT=1;}
"*/" {COMMENT=0;}
{identifier}\( {if(!COMMENT) printf("\n Function:\t %s",yytext);}
\{ {if(!COMMENT) printf("\n Block Begins");
\} {if(!COMMENT) printf("\n Block Ends");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s is an Identifier",yytext);}
\".*\" {if(!COMMENT) printf("\n %s is a String",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a Number",yytext);}
```

```
\)(\;)? {if(!COMMENT) printf("\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT) printf("\n%s is an Assmt oprtr",yytext);}
\<= |
\>= |
\< |
== {if(!COMMENT) printf("\n %s is a Rel. Operator",yytext);}
.|\n
%%
int main(int argc, char **argv)
{
if(argc>1)
{
FILE *file; file=fopen(argv[1],"r"); if(!file)
{
printf("\n Could not open the file: %s",argv[1]); exit(0);
}
yyin=file;
}
yylex(); printf("\n\n"); return 0;
}
int yywrap()
{
return 0;
}
Output:
test.c #include<stdio.h> main()
{
int fact=1,n;
for(int i=1;i<=n;i++)
{ fact=fact*i; }
printf("Factorial Value of N is", fact); getch();
}

$ lex lexp.l
$ cc lex.yy.c
$ ./a.out test.c
#include<stdio.h> is a Preprocessor Directive Function: main( )
Block Begins
int is a Keyword fact is an Identifier
= is an Assignment Operator
1 is a Number
n is an Identifier Function: for( int is a Keyword i is an Identifier
= is an Assignment Operator 1 is a Number
i is an Identifier
<= is a Relational Operator n is an Identifier
```

```
i is an Identifier
);
Block Begins
fact is an Identifier
= is an Assignment Operator fact is an Identifier
i is an Identifier Block Ends Function: printf(
"Factorial Value of N is" is a String fact is an Identifier );
Function: getch( ); Block Ends
```

Practical 3: Study the Lex and Yacc tool and evaluate an arithmetic expression with parentheses, unary and binary operators using Flex and Yacc (calculator).

Aim: Study the LEX and YACC tool and Evaluate an arithmetic expression with parentheses, unaryand binary operators using Flex and Yacc. [Need to write yylex() function and to be used with Lex andyacc.].

Description:

LEX-A Lexical analyzer generator:

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers").Lex is commonlyused with the yacc parser generator.

Lex reads an input stream specifying the lexical analyzer and outputs source code implementing thelexer in the C programming language

1. A lexer or scanner is used to perform lexical analysis, or the breaking up of an input stream intomeaningful units, or tokens.
2. For example, consider breaking a text file up into individual words.
3. Lex: a tool for automatically generating a lexer or scanner given a lex specification (.l file).

Structure of a Lex file

The structure of a Lex file is intentionally similar to that of a yacc file; files are divided up into threesections, separated by lines that contain only two percent signs, as follows:

Definition section:
%%

Rules section:

%%

C code section:

<statements>

- The **definition section** is the place to define macros and to import header files writtenin C. It is also possible to write any C code here, which will be copied verbatim into thegenerated source file.
- The **rules section** is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text inthe input matching a given pattern, it executes the associated C code. This is the basisof how Lex operates.
- The **C code section** contains C statements and functions that are copied verbatim to thegenerated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link it in at compile time.

Description:-

The lex command reads File or standard input, generates a C language program, and writes it to a file named lex.yy.c. This file, lex.yy.c, is a compilable C language program. A C++ compiler also can compile the output of the lex command. The -C flag renames the output file to lex.yy.C for the C++ compiler. TheC++ program generated by the lex command can use either STDIO or

IOSTREAMS. If the cpp define _CPP_IOSTREAMS is true during a C++ compilation, the program usesIOSTREAMS for all I/O. Otherwise, STDIO is used.

The lex command uses rules and actions contained in File to generate a program, lex.yy.c,which can becompiled with the cc command. The compiled lex.yy.c can then receive input, break the input into the logical pieces defined by the rules in File, and run program fragments contained in the actions in File.

The generated program is a C language function called yylex. The lex command stores the yylex functionin a file named lex.yy.c. You can use the yylex function alone to recognize simple one-word input, or you can use it with other C language programs to perform more difficult input analysis functions. For example, you can use the lex command to generate a program that simplifies an input stream before sending it to a parser program generated by the yacc command. The yylex function analyzes the input stream using a program structure called a finite state machine. This structure allows the program to exist in only one state (or condition) at a time. There is a finite number of states allowed. The rules in File determine how the program moves from one state to another. If you do not specify a File, the lex command reads standard input. It treats multiple files as asingle file.
Note: Since the lex command uses fixed names for intermediate and output files, you can have only oneprogram generated by lex in a given directory.

Regular Expression Basics

. : matches any single character except \n

* : matches 0 or more instances of the preceding regular expression

+ : matches 1 or more instances of the preceding regular expression

? : matches 0 or 1 of the preceding regular expression

| : matches the preceding or following regular
expression[ ] : defines a character class

() : groups enclosed regular expression into a new regular expression

“…”: matches everything within the “ “ literally

Special Functions

- yytext

– where text matched most recently is stored

- yyleng

– number of characters in text most recently matched

• yylval

– associated value of current token

• yymore()

– append next string matched to current contents of yytext

• yyless(n)

– remove from yytext all but the first n characters

• unput(c)

– return character c to input stream

• yywrap()

– may be replaced by user

– The yywrap method is called by the lexical analyser whenever it inputs an EOF as the first characterwhen trying to match a regular expression

Files

y.output--Contains a readable description of the parsing tables and a report on conflicts generated bygrammar ambiguities.

y.tab.c -- Contains an output file.
y.tab.h----Contains definitions for token names.

yacc.tmp -- Temporary file.

yacc.debug-- Temporary file.

yacc.acts -- Temporary file.

/usr/ccs/lib/yaccpar -Contains parser prototype for C programs.

/usr/ccs/lib/liby.a-- Contains a run-time library.

YACC: Yet Another Compiler-Compiler

Yacc is written in portable C. The class of specifications accepted is a very general one:LALR(1) grammarswith disambiguating rules.

Basic specification

Names refer to either tokens or non-terminal symbols. Yacc requires tokens names to be declared as such. In addition, for reasons discussed in section 3, it is often desirable to include

the lexical analyzer aspart of the specification file, I may be useful to include other programs as well. Thus, the sections are separated by double percent "%%" marks. (the percent"%" is generally used in yacc specifications as an escape character). In other words, a full specification file looks like.

In other words a full specification file
looks likeDeclarations
%%
Rules

%%

Programs

The declaration section may be empty. More over if the programs section is omitted, the second %%mark may be omitted also thus the smallest legal yacc specification is

%%
Rules

Blanks, tabs and newlines are ignored except that they may not appear in

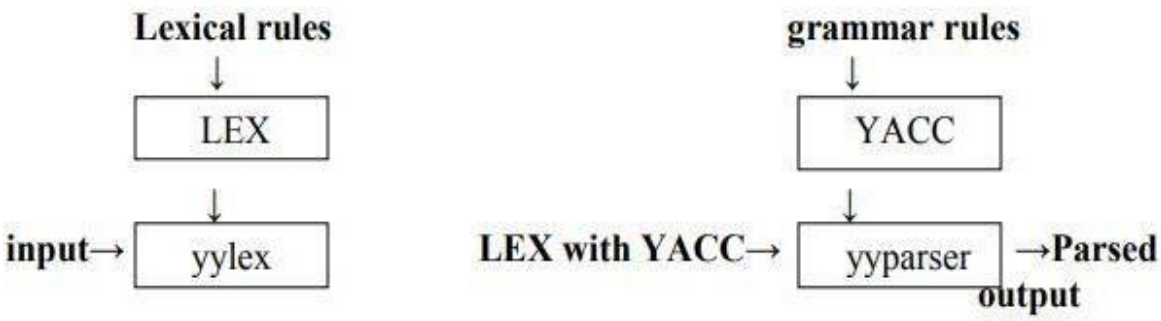
names or multi-character reserved symbols. Comments may appear wherever legal,

they are enclosed in /*….*/ as in C and PL/l

The rules section is made up of one or more grammar rule has the
form: A:BODY:

USING THE LEX PROGRAM WITH THE YACC PROGRAM

The Lex program recognizes only extended regular expressions and formats them into character packages called tokens, as specified by the input file. When using the Lex program to make a lexical analyzer for a parser, the lexical analyzer (created from the Lex command) partitions the input stream.The parser(from the yacc command) assigns structure to the resulting pieces. You can also use other programs along with programs generated by Lex or yacc commands.



A token is the smallest independent unit of meaning as defined by either the parser or the lexical analyzer. A token can contain data, a language keyword, an identifier or the parts of language syntax.

The yacc program looks for a lexical analyzer subroutine named yylex, which is generated by the lex command. Normally the default main program in the Lex library calls the yylex subroutines. However ifthe yacc command is loaded and its main program is used , yacc calls the yylex subroutines. In this caseeach Lex rule should end with:

return (token);

Where the appropriate token value is returned

The yacc command assigns an integer value to each token defined in the yacc grammar file through a # define preprocessor statement.

The lexical analyzer must have access to these macros to return the tokens to the parser. Use the yacc – d option to create a y.tab.h file and include the y.tab.h file in the Lex specification file by adding the following lines to the definition section of the Lex specification file:

%{

#include “y.tab.h”

%}

Alternatively you can include the lex.yy.c file the yacc output file by adding the following lines after the second %% (percent sign, percent sign) delimiter in the yacc grammar file:

#include”lex.yy.c”

The yacc library should be loaded before the Lex library to get a main program that invokes the yacc parser. You can generate Lex and yacc programs in either order.

Evaluating Arithmetic Expression using different Operators (Calculator)

Algorithm:

1) Get the input from the user and Parse it token by token.

2) First identify the valid inputs that can be given for a program.

3) The Inputs include numbers, functions like LOG, COS, SIN, TAN, etc. and operators.

4) Define the precedence and the associativity of various operators like +,-,/,* etc.

5) Write codes for saving the answer into memory and displaying the result on the screen.

6) Write codes for performing various arithmetic operations.

7) Display the possible Error message that can be associated with this calculation.

8) Display the output on the screen else display the error message on the screen.

Program: CALC.L

```
%{
#include<stdio.h>
 #include<stdlib.h>
void yyerror(char *);
#include "y.tab.h" int yylval;
%}
%%
[a-z] {yylval=*yytext='&'; return VARIABLE;} [0-9]+ {yylval=atoi(yytext); return
INTEGER;} CALC.Y
[\t] ;
%%
int yywrap(void)
{
return 1;
}
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%{
int yylex(void);
void yyerror(char *); int sym[26];
%}
%% PROG:
PROG STMT '\n'
;
STMT: EXPR {printf("\n %d",$1);}
```

```
| VARIABLE '=' EXPR {sym[$1] = $3;}
;
EXPR: INTEGER
| VARIABLE {$$ = sym[$1];}
| EXPR '+' EXPR {$$ = $1 + $3;}
| '(' EXPR ')' {$$ = $2;}
%%

void yyerror(char *s)
{
printf("\n %s",s); return;
}
int main(void)
{
printf("\n Enter the Expression:"); yyparse();
return 0;
}
Output:
$ lex calc.l
$ yacc -d calc.y
$ cc y.tab.c lex.yy.c -ll -ly -lm
$ . / a . out
Enter the Expression: ( 5 + 4 ) * 3 Answer: 27
```

Practical 4: Using JFLAP, create a DFA from a given regular expression.

Aim: Using JFLAP, create a DFA from a given regular expression. All types of error must be checked during the conversion.

What is JFLAP: -

JFLAP program makes it possible to create and simulate automata. Learning about automata with pen and paper can be difficult, time consuming and error-prone. With JFLAP we can create automata of different types and it is easy to change them as we want. JFLAP supports creation of DFA and NFA, Regular Expressions, PDA, Turing Machines, Grammars and more.

Setup: -

JFLAP is available from the homepage: (www.JFLAP.org). From there press “Get FLAP” and follow the instructions. You will notice that JFLAP have a .JAR extension. This means that you need Java to run JFLAP.

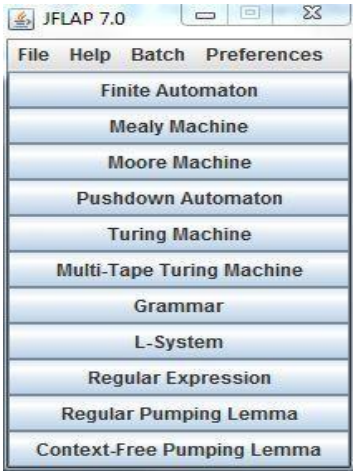
With Java correctly installed you can simply select the program to run it. You can also use a command console run it from the files current directory with, Java –jar JFLAP.jar.

Using JFLAP: -

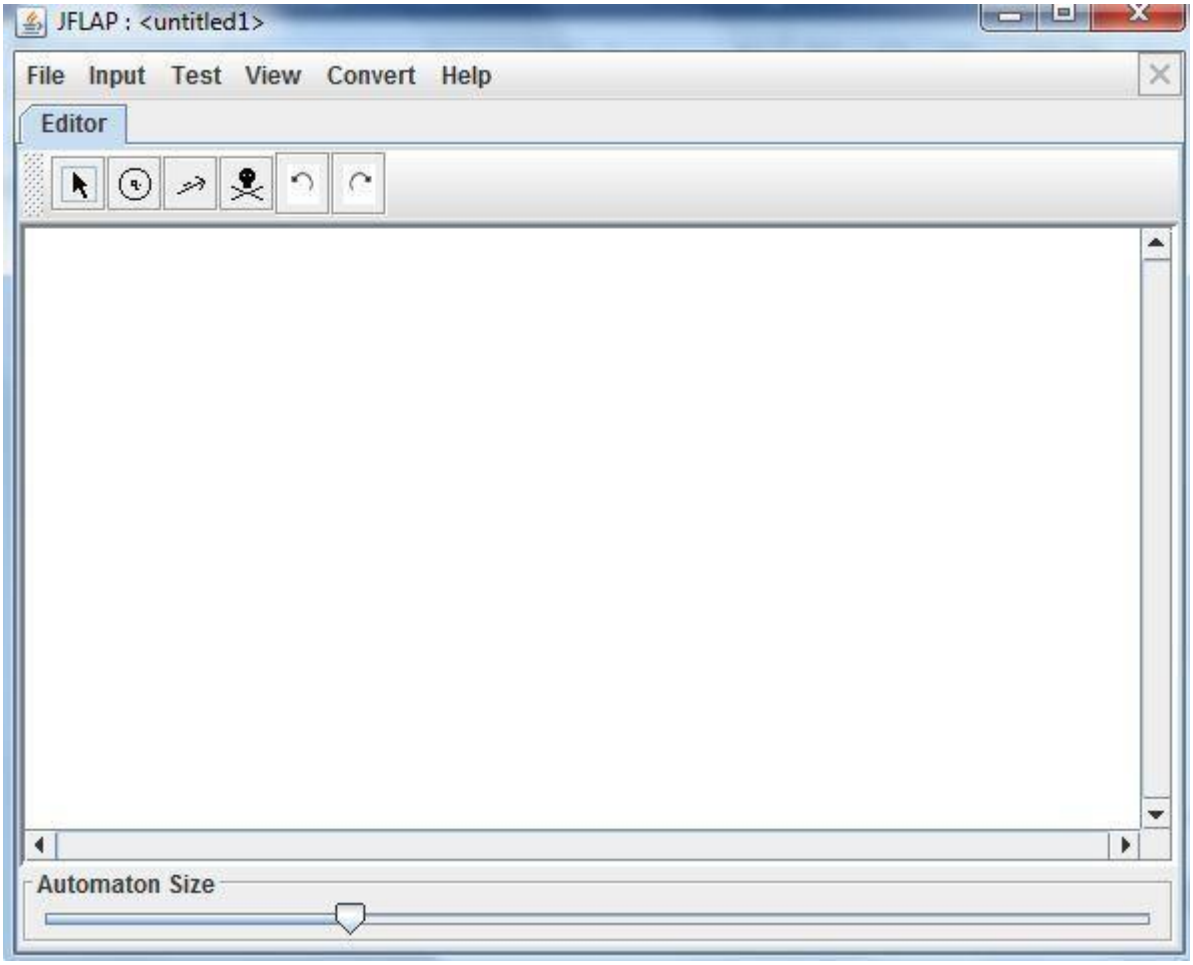
When you first start JFLAP you will see a small menu with a selection of eleven different automata and rule sets. Choosing one of them will open the editor where you create chosen type of automata. Usually you can create automata containing states and transitions but there is also creation of Grammar and Regular Expression which is made with a text editor.

DFA from a given regular expression: -

First, we need to select Regular Expression from the JFLAP Menu.



Now you should have an empty window in front of you. You will have a couple of tools and features at your disposal.



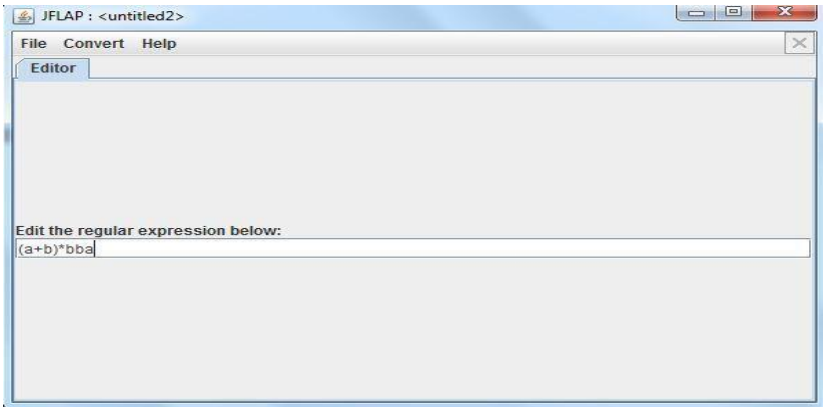
The toolbar contains six tools, which are used to edit automata.

Attribute Editor Tool, changes properties and position of existing states and transitions.

State Creator Tool, creates new states. **Transition Creator Tool,** creates transitions. **Deletion Tool,** deletes states and transitions.

Undo/Redo, changes the selected object prior to their history.

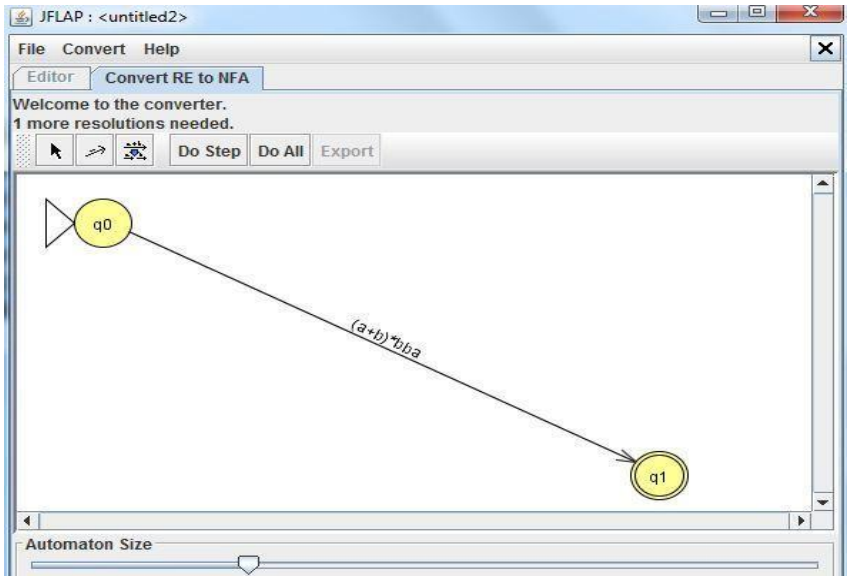
Regular Expressions can be typed into JFLAP to be converted to an NFA



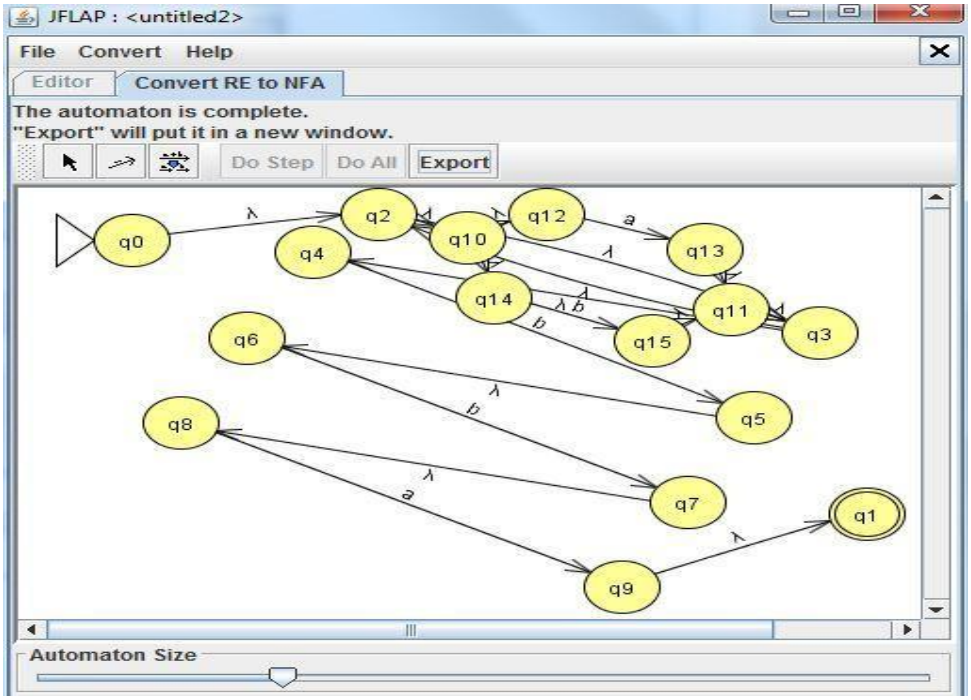
Choose Regular Expression in the main menu, then just type the expression in the textbox. Definitions for Regular Expressions in JFLAP:

- **Kleene Star**
- **+ Union**
- **! Empty String**

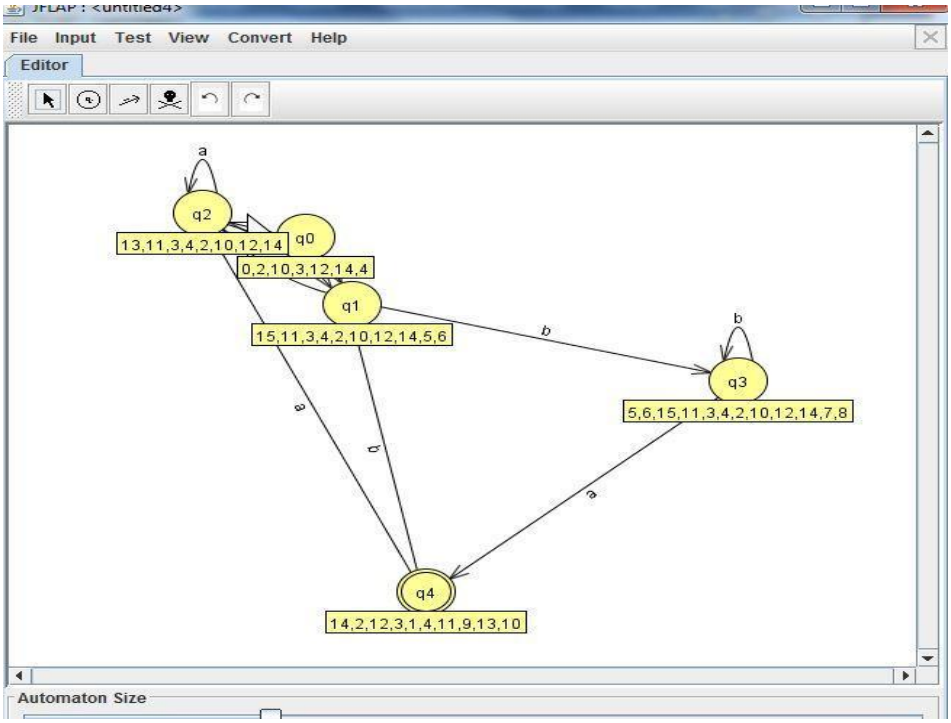
Correctly written expressions can then be converted to an NFA. To convert your expression select Convert → Convert to NFA. The conversion will begin with two states and a transition with your Regular Expression. With the (D)e-expressionify Transition tool you can break down the Regular Expression into smaller parts. Each transition will contain a sub expression. The next step is to link every rule with lambda transitions. Add new transition between states that should be connected with the Transition Tool. If you are unsure what to do you can select Do Step to automatically make the next step. If you want the NFA immediately Do All creates the whole NFA for you.



You can notice how the conversion differs depending on how the Regular Expression looks. For example the expression a+b results in a fork, were either ‘a’ or ‘b’ can be chosen.



Now finally convernt NFA to DFA:-

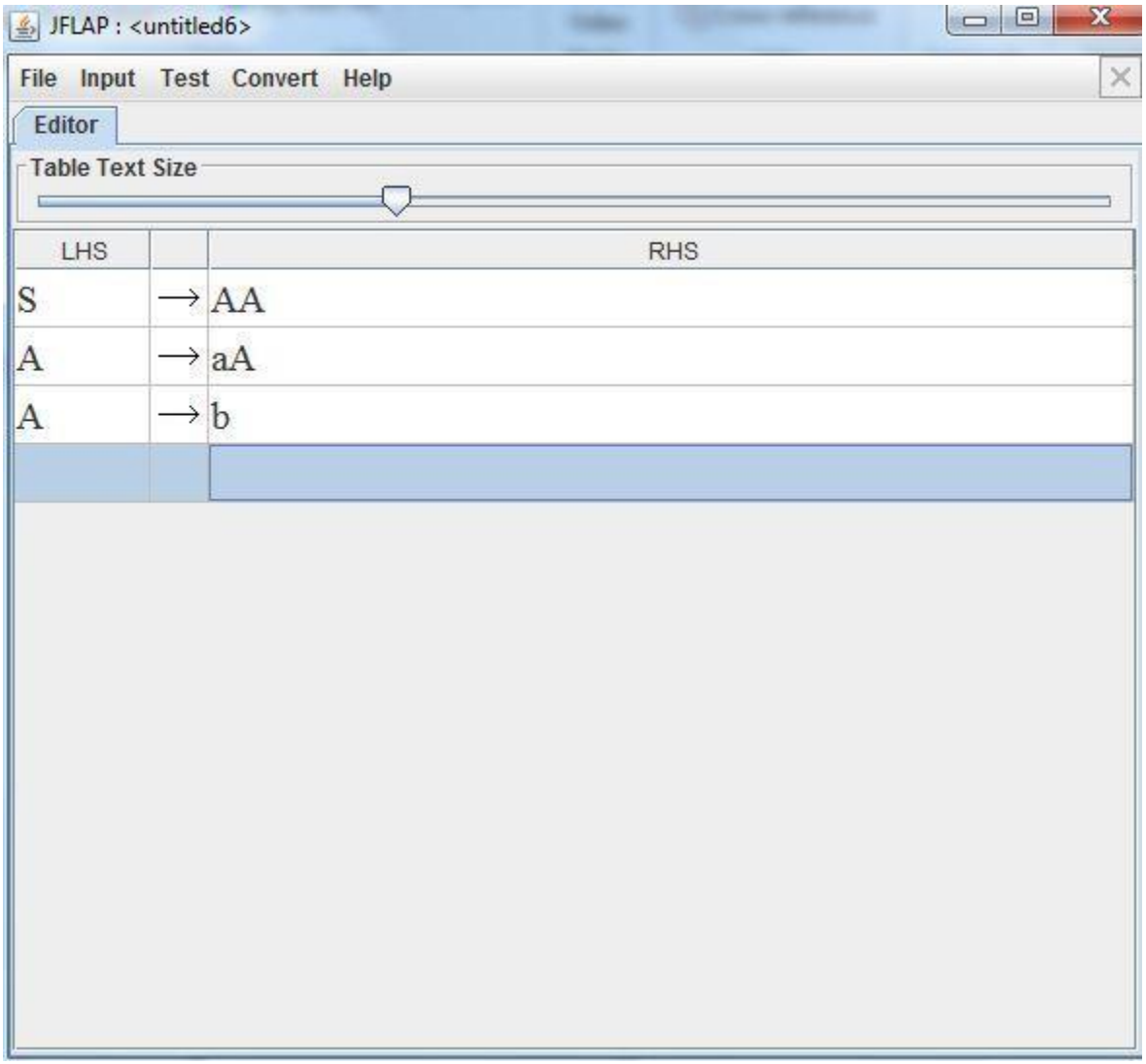

File Input Test View Convert Help
Editor
a
q2
13,11,3,4,2,10,12,14
q0
0,2,10,3,12,14,4
q1
15,11,3,4,2,10,12,14,5,6
b
b
q3
5,6,15,11,3,4,2,10,12,14,7,8
a
a
a
q4
14,2,12,3,1,4,11,9,13,10
Automaton Size

Practical 5: Create LL(1) parse table for a given CFG and hence simulate LL(1) parsing.

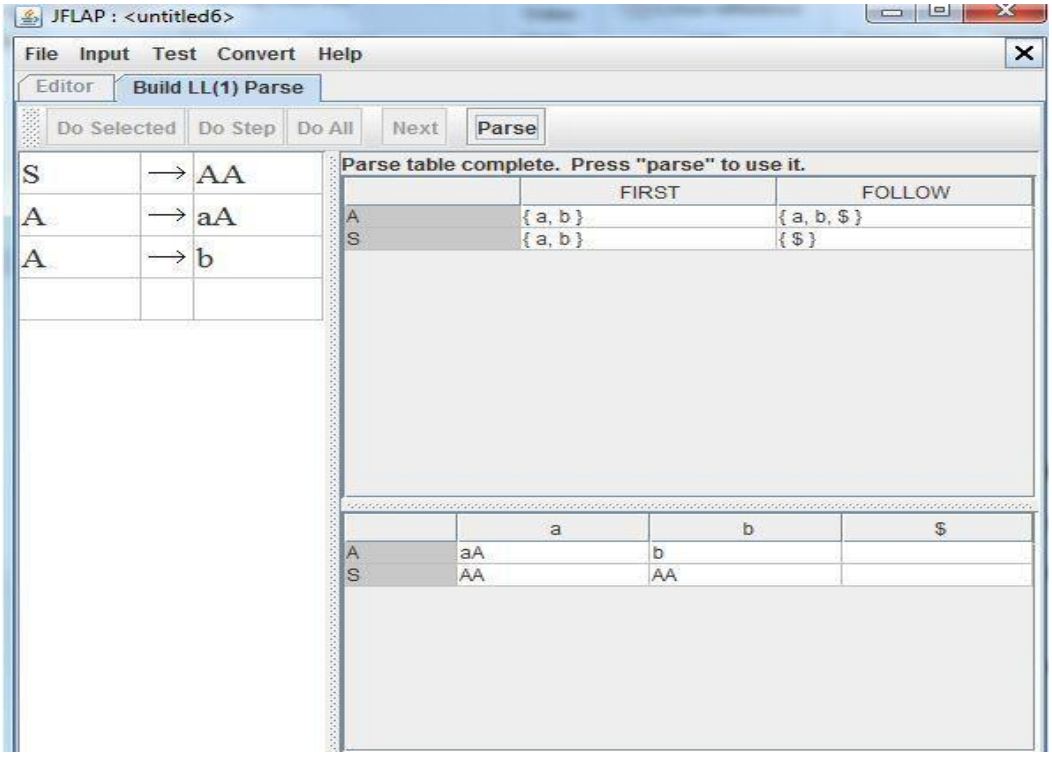
Aim: Using JFLAP create LL(1) parse table for a given CFG and hence Simulate LL(1) parsing

Implementation: -

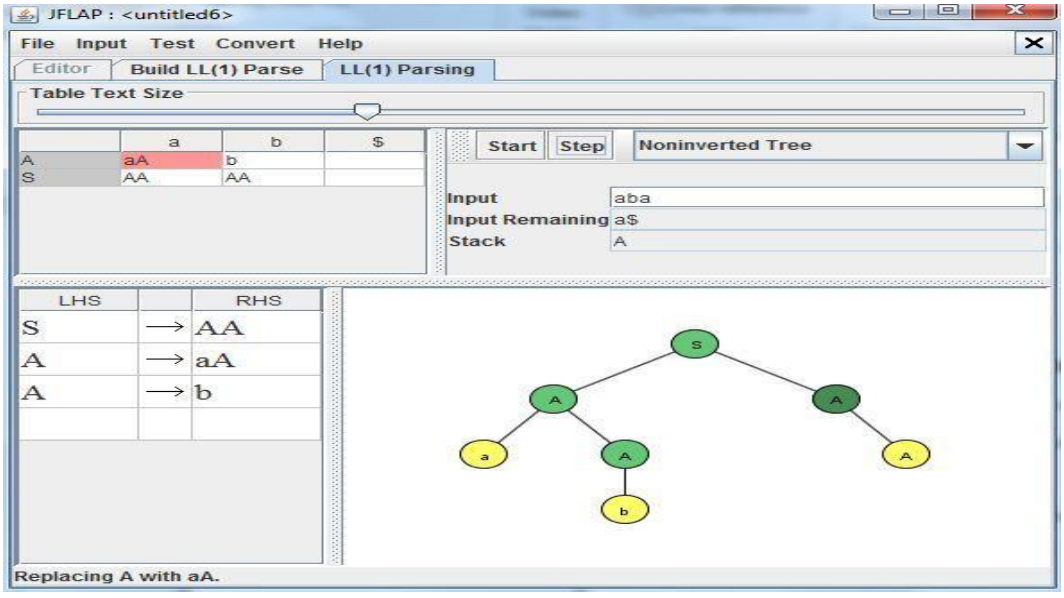
Step 1. Choose the grammar from JFLAP and insert grammar you want to create LL(1) parsing table.



Step 2: - Select the Input from Menu and select Build LL(1) parsing table from it.



Step 3: - Now select parse to use that table to create parse tree from it.



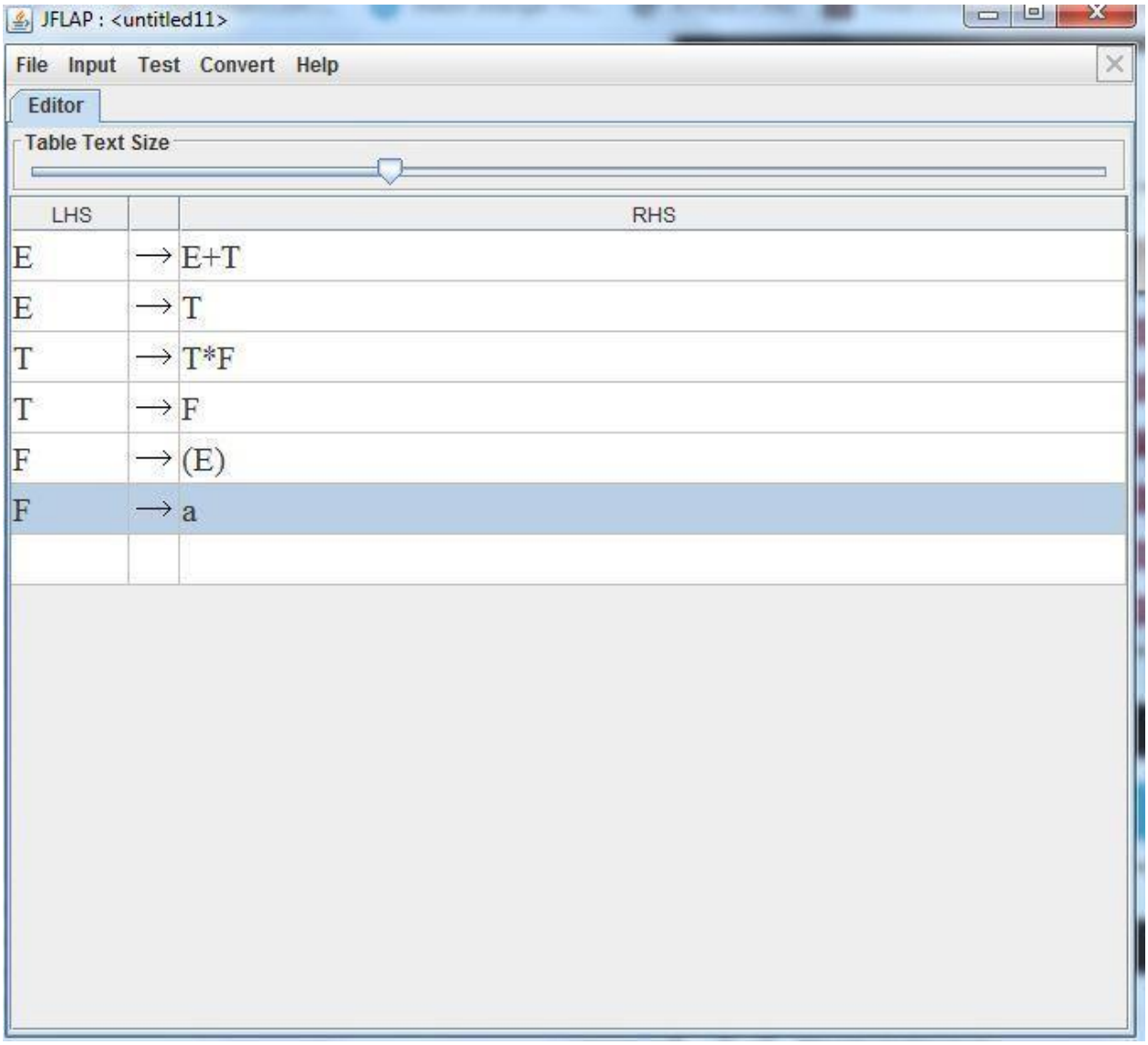
Result:- We create LL(1) parse table for a given CFG and hence Simulate LL(1) parsing

Practical 6: Using JFLAP, create SLR(1) parse table for a given grammar. Simulate parsing and output the parse tree in proper format.

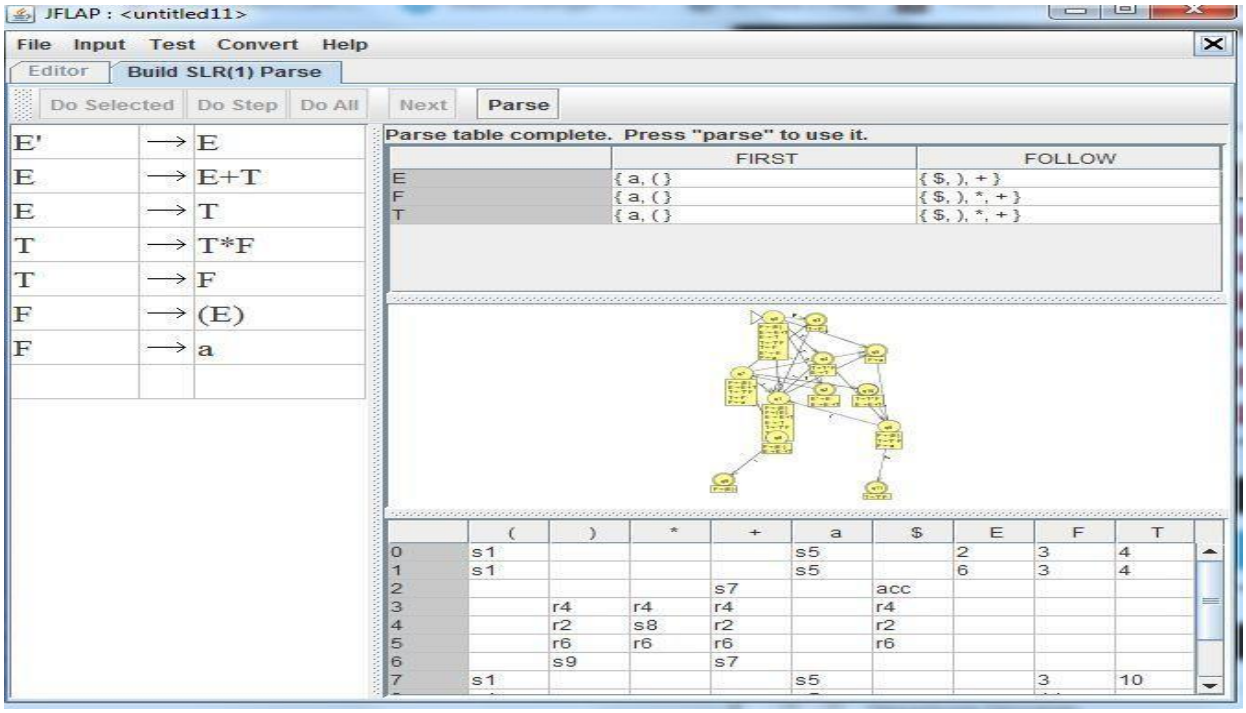
Aim: Using JFLAP create SLR(1) parse table for a given grammar. Simulate parsing and output the parse tree proper format.

Implementation: -

Step 1: - Choose the grammar from JFLAP and insert grammar you want to create SLR(1) parsing table.


JFLAP : <untitled11>
File Input Test Convert Help
Editor
Table Text Size
LHS
RHS
E → E+T
E → T
T → T*F
T → F
F → (E)
F → a

Step 2: - Select the Input from Menu and select Build SLR (1) parsing table from it.

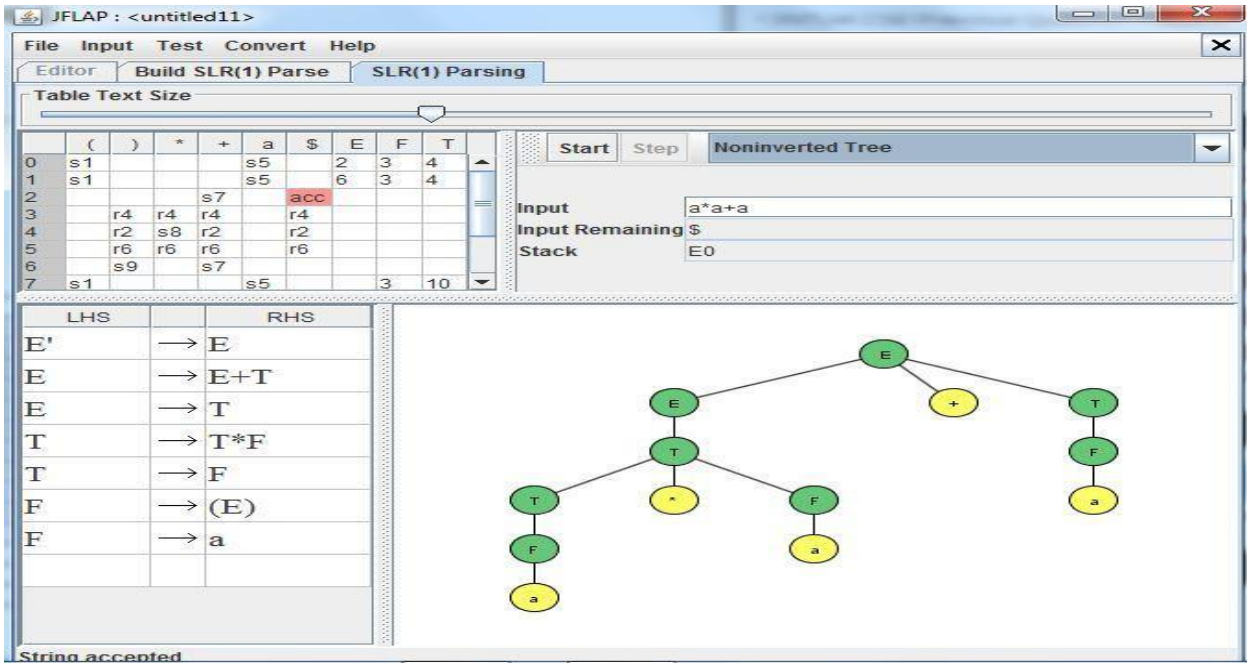


E'	⟶	E
E	⟶	E+T
E	⟶	T
T	⟶	T*F
T	⟶	F
F	⟶	(E)
F	⟶	a

	FIRST	FOLLOW
E	{ a, (}	{ $,), + }
F	{ a, (}	{ $,), *, + }
T	{ a, (}	{ $,), *, + }

	(	)	*	+	a	$	E	F	T
0	s1				s5		2	3	4
1	s1				s5		6	3	4
2				s7		acc			
3		r4	r4	r4		r4			
4		r2	s8	r2		r2			
5		r6	r6	r6		r6			
6		s9		s7					
7	s1				s5			3	10

Step 3: - Now select parse to use that table to create parse tree from it.



	(	)	*	+	a	$	E	F	T
0	s1				s5		2	3	4
1	s1				s5		6	3	4
2				s7		acc			
3		r4	r4	r4		r4			
4		r2	s8	r2		r2			
5		r6	r6	r6		r6			
6		s9		s7					
7	s1				s5			3	10

LHS		RHS
E'	⟶	E
E	⟶	E+T
E	⟶	T
T	⟶	T*F
T	⟶	F
F	⟶	(E)
F	⟶	a

Result :- We created SLR(1) parse table for a given grammar and Simulated parsing and output the parse tree proper format.

Practical 7: Write functions to find FIRST and FOLLOW of all the variables.

Aim: Write a suitable data structure to store a Context Free Grammar. Prerequisite is to eliminate left recursion from the grammar before storing. Write functions to find FIRST and FOLLOW of all the variables. [May use unformatted file / array to store the result].

Algorithm:

First ()-

If x is a terminal, then FIRST(x) = { 'x' }

If x-> Є, is a production rule, then add Є to FIRST(x).

If X->Y1 Y2 Y3….Yn is a production,

FIRST(X) = FIRST(Y1)

If FIRST(Y1) contains Є then FIRST(X) = { FIRST(Y1) – Є } U { FIRST(Y2) }

If FIRST (Yi) contains Є for all i = 1 to n, then add Є to FIRST(X).

Follow ()-

FOLLOW(S) = { $ } // where S is the starting Non-Terminal

If A -> pBq is a production, where p, B and q are any grammar symbols,

then everything in FIRST(q) except Є is in FOLLOW(B).

If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B). If A->pBq is a production and FIRST(q) contains Є,

then FOLLOW(B) contains { FIRST(q) – Є } U FOLLOW(A)

Program:

Practical 8: Using JFLAP, create SLR(1) parse table for a given grammar. Simulate parsing and output the parse tree in proper format.

Accept the input string with Regular expression of Finite Automaton: 101+.

Source code:

```
def FA(s):
#if the length is less than 3 then it can't be accepted, Therefore end the
process.
   if len(s):
        return "Rejected"
  #first three characters are fixed. Therefore, checking them using index
    if s[0]=='1':
       if s[1]=='0':
          if s[2]=='1':
             # After index 2 only "1" can appear. Therefore break the process if any other
  character is detected
             for i in range(3,len(s)):
                if s[i]!='1':
                   return "Rejected"
             return "Accepted" # if all 4 nested if true
          return "Rejected" # else of 3rd if
       return "Rejected" # else of 2nd if
    return "Rejected" # else of 1st if
  inputs=['1','10101','101','10111','01010','100','','10111101','1011111']
  for i in inputs:
    print(FA(i))
```

Output:

```
Rejected
Rejected
Accepted
Accepted
Rejected
Rejected
Rejected
Rejected
Accepted
```