Table of Contents

BPUT Syllabus

PART-1

1. Using JFLAP, create a DFA from a given regular expression. All types of error must be checked during the conversion.
2. Read a regular expression in standard form and check its validity by converting it to postfix form. Scan a string and check whether the string matches against the given regular expression or not.
3. (Tokenizing). A programs that reads a source code in C/C++ from an unformatted file and extract various types of tokens from it (e.g. keywords/variable names, operators, constant values).
4. Read a regular expression in its standard form and find out an Ɛ-NFA from it. Need to use adjacency list data structure of graph to store NFA. Thompson’s construction needs to be used too. [2 labs]
5. Evaluate an arithmetic expression with parentheses, unary and binary operators using Flex and Yacc.[Need to write yylex() function and to be used with Lex and yacc.]
6. (Tokenizing) Use Lex and yacc to extract tokens from a given source code.

PART-2

7. 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].
8. Using JFLAP create LL(1) parse table for a given CFG and hence Simulate LL(1) parsing.
9. Using JFLAP create SLR(1) parse table for a given grammar. Simulate parsing and output the parse tree proper format.

COURSE OBJECTIVES [COB]

COB	Objectives
1	Able to explain the importance of Finite Automata in Lexical Analyzer
2	Able to design and Implement Lexical Analyzer using LEX tool
3	Able to design and Implement Syntax Analyzer using YACC tool
4	Able to construct Parse Tree using Top-Down and Bottom-Up parsing techniques

COURSE OUTCOMES [CO]

On Completion of this Course, the students should be able to:

CO	Outcomes	Levels
1.	Recognize tokens by writing c code or by using JFLAP.	Knowledge, Understand & Level 1, 2
2.	Analyze how the LEX tool techniques works for pattern matching	Analyze: Level 4
3.	Examine how the YACC translator translates the code in to the compiler	Understand & Analyze: Level 2 & 4
4.	Demonstrate the top-down & bottom-up parsers with the help JFLAP.	Knowledge, Understand & Level 1, 2

Laboratory Plan

Expriment No.	No. of Period [50 min]	Topics/ Sub-Topics	COB/CO	Mode of Delivery
1	2	Simulation of Finite Automaton using JFLAP	COB:1 CO:1	JFLAP tool
2	2	Token Reorganization using C Code	COB:1 CO:1	C Compiler
3	2	Pattern Matching using C Code	COB:1 CO:1	C Compiler
4	4	Design of Lexical Analyzer using LEX Tool	COB:2 CO:2	LEX Compiler
5	4	Integrating LEX With YACC	COB:3 CO:3	LEX Compiler / YACC Compiler
6	2	Finding First() and Follow() of a given Grammar	COB:4 CO:4	JFLAP tool / C Compiler
7	2	Simulation of Top-Down Parsing using JFLAP	COB:4 CO:4	JFLAP tool
8	2	Simulation of Bottom-Up Parsing using JFLAP	COB:4 CO:4	JFLAP tool

Suggested Books:

1. Compilers: Principles, Techniques and Tools By Aho, Lam, Sethi, and Ullman, Second Edition, Pearson, 2014
2. Ronald Mak, Writing Compilers and Interpreters: A Software Engineering Approach, Wiley
3. A. W. Apple, Modern compiler implementation in C, Cambridge University Press

Lab Evaluation Policy

Distribution of Lab Mark for Each lab –

Evaluation	50 marks
Record	20 marks
Quiz/ viva	20 marks
Attendance	10 marks
Total	**100 marks**

Note: The final lab contains 50% credit and sum of all reaming labs contains rest of the credit.

All programs should be flexi coded – no hard coded string/inputs are expected. All inputs must be presented at runtime. Whenever needed, a predefined approach has to be implemented in programs to describe some parameters (e.g. how to represent ε in constructing ε – NFA from a regular expression). A list of "must do" assignments corresponding to a particular lab is represented in bold letter.

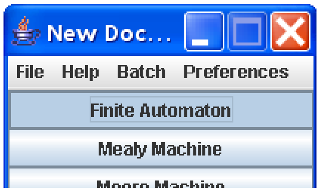
Assessment Methods in Alignment with Intended Learning Outcomes

Assessment Method	**Marks**	**Weightage %age**	**Intended Learning Outcome to be assessed**			
			CO1	**CO2**	**CO3**	**CO4**
Record	0.5x10=5	50%	15%	35%	35%	15%
Quiz / Viva	0.5x10=5		15%	35%	35%	15%
Lab Performance	1.25x10=12.5		15%	35%	35%	15%
Attendance	0.25x10=2.5		-	-	-	
Final Lab Exam	25	50%	15%	35%	35%	15%

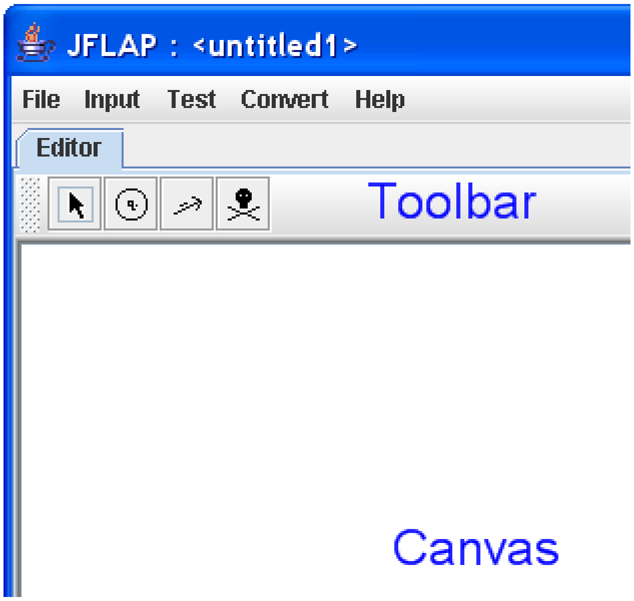
Laboratory – 1

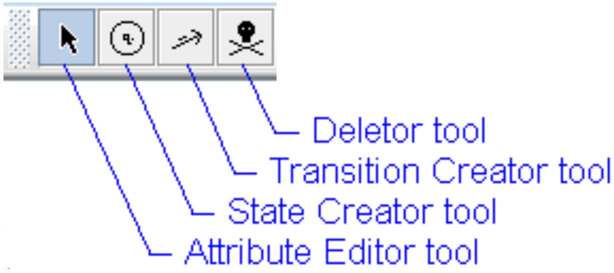
Simulating a DFA from a given regular expression

To start a new FA, start JFLAP and click the **Finite Automaton** option from the menu.



This should bring up a new window that allows you to create and edit an FA. The editor is divided into two basic areas: the canvas, which you can construct your automaton on, and the toolbar, which holds the tools you need to construct your automaton.

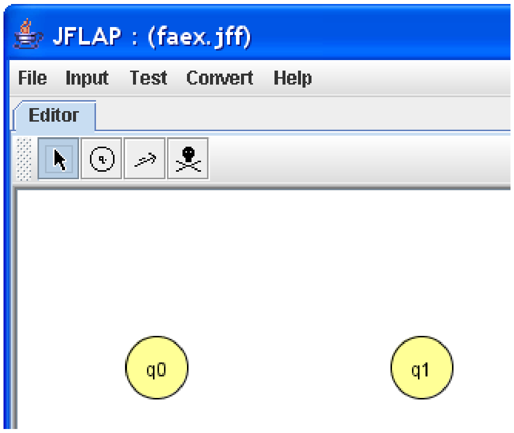




As you can see, the toolbar holds four tools:

- Attribute Editor tool : sets initial and final states
- State Creator tool : creates states
- Transition Creator tool : creates transitions
- Deletor tool : deletes states and transitions

To select a tool, click on the corresponding icon with your mouse. When a tool is selected, it is shaded, as the Attribute Editor tool is above. Selecting the tool puts you in the corresponding mode. For instance, with the toolbar above, we are now in the Attribute Editor mode.



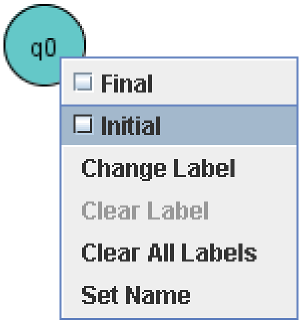
The different modes dictate the way mouse clicks affect the machine. For example, if we are in the State Creator mode, clicking on the canvas will create new states. These modes will be described in more detail shortly.

Creating States

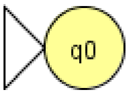
First, let's create several states. To do so we need to activate that State Creator tool by clicking the button on the toolbar. Next, click on the canvas in different locations to create states. We are not very sure how many states we will need, so we created four states. Your editor window should look something like this:

Defining Initial and Final States

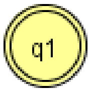
Arbitrarily, we decide that q_0 will be our initial state. To define it to be our initial state, first select the Attribute Editor tool on the toolbar. Now that we are in Attribute Editor mode, right-click on q_0. This should give us a pop-up menu that looks like this:



From the pop-up menu, select the checkbox **Initial**. A white arrowhead appreas to the left of q_0 to indicate that it is the inital state.



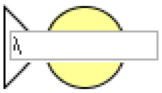
Next, let's create a final state. Arbitrarily, we select q_1 as our final state. To define it as the final state, right-click on the state and click the checkbox **Final**. It will have a double outline, indicating that it is the final state.



Creating Transitions

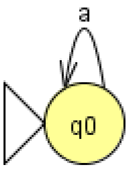
We know strings in our language can start with a's, so, the initial state must have an outgoing transition on a. We also know that it can start with any number of a's, which means that the FA should be in the same state after processing input of any number of a's. Thus, the outgoing transition on a from q_0 loops back to itself.

To create such a transition, first select the Transition Creator tool from the toolbar. Next, click on q_0 on the canvas. A text box should appear over the state:

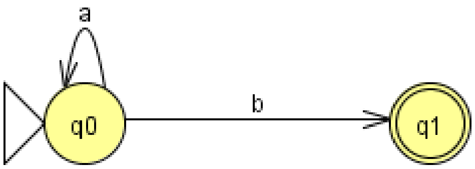


Note that λ, reprsenting the empty string, is initially filled in for you. If you prefer ε representing the empty string, select **Preferences : Preferences** in the main menu to change the symbol representing the empty string.

Type "a" in the text box and press **Enter**. If the text box isn't selected, press **Tab** to select it, then enter "a". When you are done, it should look like this:

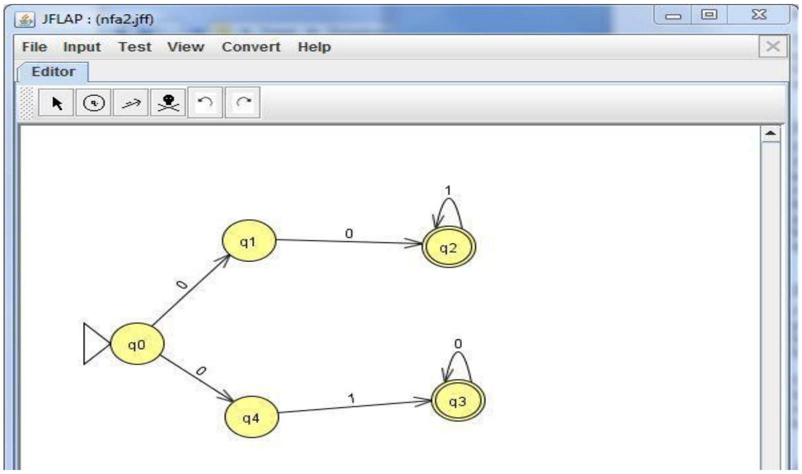


Next, we know that strings in our language must end with a odd number of b's. Thus, we know that the outgoing transtion on b from q_0 must be to a final state, as a string ending with one b should be accepted. To create a transition from our initial state q_0 to our final state q_1, first ensure that the Transition Creator tool is selected on the toolbar. Next, click and hold on q_0, and drag the mouse to q_1 and release the mouse button. Enter "b" in the textbox the same way you entered "a" for the previous transition.



1. Create a NFA for representing the language *(001*)|(010*) using JFLAP*

Creating DFA and NFA (both combined as FA in JFLAP) is fast and you do not have to set any variables or labels by adding items. Users do not have to set any alphabet before debugging which makes this simulator very straightforward. By double left clicking any transition the user can easily setup every transition.



Assignment

Design NFA, DFA, minimized DFA for the following set of languages:

1. R=(b|ab*ab*)* ∑={a,b}
2. R=(ab+a)*bab ∑={a,b}
3. Set of all strings over ∑={0,1} where each end with '00'
4. Set of all strings over ∑={0,1} having only even no of 0's.
5. Set of all strings over ∑={0,1} having only odd no of 1's.
6. Set of all strings over ∑={0,1} where each string contain substring '0101'
7. Set of all strings over ∑={0,1} where length of each string is at most 5.
8. R=((1+0)1)*100 ∑={0,1}
9. R=ab(a+ba)*a* ∑={a,b}
10. R=(a|b)*ab(a|b)* ∑={a,b}
11. Set of all strings over {0, 1} which not start with 10.
12. Set of all strings over {0, 1} having odd number of 0's and 1's.
13. Set of all strings over {0, 1} having even number of 0's and odd number of 1's.
14. Set of all strings over {0, 1} which end with 3 consecutive 1's.
15. Set of all strings over {0, 1} which do not have the substring 110.
16. Set of all strings of the form $a^{2n}bb$, $n \geq 0$ over {a, b}
17. Accept set of all strings start with a followed by any number of b's and ends with a, or set of all strings start with b followed by any number of a's and ends with b.
18. Set of all strings over {0, 1} which contain at least 3 a's.

Laboratory – 2

Acceptance of a string from a given regular expression

Regular Expression represents patterns of strings of characters. A regular expression *r* is completely defined by the set of strings that it matches. This is called the **language generated by the regular expression** and is written as L(r). A regular expression *r* will also contain characters from, the alphabet, but such characters have a different meaning: in regular expression all symbols indicate patterns.

A regular expression is built up out of simpler regular expression using as set of defining rules. Each regular expression *r* denotes a language L(r). The defining rules specify how L(*r*) is formed by combining in various ways the language denoted by the sub expression of *r.*

Finite Automata(FA) is the simplest machine to recognize patterns (regular expression).

A Finite Automata consists of the following :

```
Q : Finite set of states.
∑ : set of Input Symbols.
q : Initial state.
F : set of Final States.
δ : Transition Function.
```

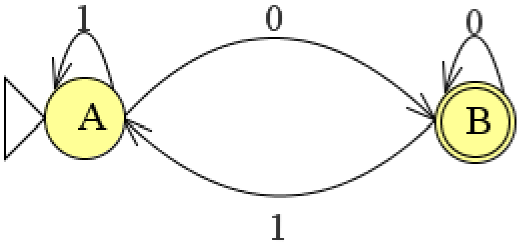
Formal specification of machine is M = { Q, ∑, δ, q, F }.

Deterministic Finite Automata (DFA)

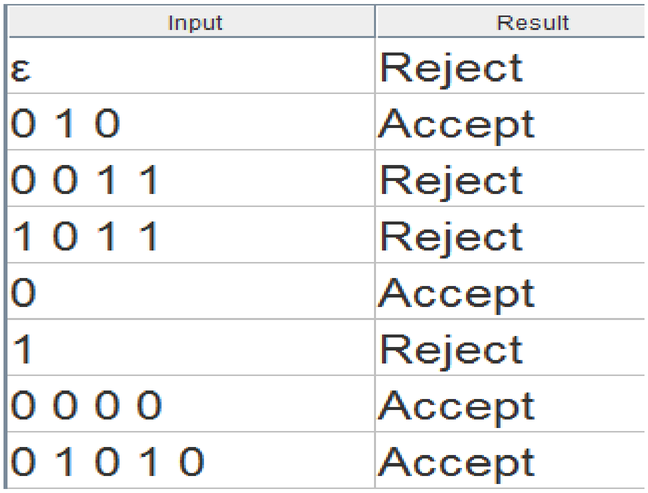
```
DFA consists of 5 tuples {Q, ∑, q, F, δ}.
Q : set of all states.
∑ : set of input symbols. ( Symbols which machine takes as input )
q : Initial state. ( Starting state of a machine )
F : set of final state.
δ : Transition Function, defined as δ : Q X ∑ --> Q.
```

In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

For example, below DFA with ∑ = {0, 1} accepts all strings ending with 0.



For this DFA input and output will be as follows-



Input	Result
ε	Reject
0 1 0	Accept
0 0 1 1	Reject
1 0 1 1	Reject
0	Accept
1	Reject
0 0 0 0	Accept
0 1 0 1 0	Accept

```
#include <stdio.h>
#include <string.h>
//starting state
int dfa = 1;

// This function is for
// the starting state A of DFA
void A(char c)
{
    if (c == '0')
        dfa = 2;
    else if (c == '1')
        dfa = 1;
    // -1 is used to check for any invalid
symbol
    else
        dfa = -1;
}

// This function is for the first state B of DFA
void B(char c)
{
    if (c == '0')
        dfa = 2;
    else if (c == '1')
        dfa = 1;
    else
        dfa = -1;
}

int isAccepted(char str[])
{
    // store length of string
    int i, len = strlen(str);

    for (i = 0; i < len; i++) {
        if (dfa == 1)
            A(str[i]);

        else if (dfa == 2)
            B(str[i]);

        else
            return 0;
    }
    if (dfa == 2)
        return 1;
    else
        return 0;
}


int main()
{
    char str[50];
    printf("enter the string");
    gets(str);
    if (isAccepted(str))
        printf("%s is ACCEPTED",str);
    else
        printf("%s is NOT
ACCEPTED",str);
    return 0;
}
```

Assignment

Check the acceptance of the string for the following set of languages:

1. R=(b|ab*ab*)* ∑={a,b}
2. R=(ab+a)*bab ∑={a,b}
3. Set of all strings over ∑={0,1} where each end with '00'
4. Set of all strings over ∑={0,1} having only even no of 0's.
5. Set of all strings over ∑={0,1} having only odd no of 1's.
6. Set of all strings over ∑={0,1} where each string contain substring '0101'
7. Set of all strings over ∑={0,1} where length of each string is at most 5.
8. R=((1+0)1)*100 ∑={0,1}
9. R=ab(a+ba)*a* ∑={a,b}
10. R=(a|b)*ab(a|b)* ∑={a,b}
11. Set of all strings over {0, 1} which not start with 10.
12. Set of all strings over {0, 1} having odd number of 0's and 1's.
13. Set of all strings over {0, 1} having even number of 0's and odd number of 1's.
14. Set of all strings over {0, 1} which end with 3 consecutive 1's.
15. Set of all strings over {0, 1} which do not have the substring 110.
16. Set of all strings of the form $a^{2n}bb$, n ≥ 0 over {a, b}
17. Accept set of all strings start with a followed by any number of b's and ends with a, or set of all strings start with b followed by any number of a's and ends with b.
18. Set of all strings over {0, 1} which contain at least 3 a's.

Laboratory – 3

Token Extraction/Recognition

Recognition of Tokens
Tokens can be recognized by Finite Automata
A Finite automaton(FA) is a simple idealized machine used to recognize patterns within input taken from some character set(or Alphabet) C. The job of FA is to accept or reject an input depending on whether the pattern defined by the FA occurs in the input.
What is a token?
A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Keywords; Examples-for, while, if etc.
Identifier; Examples-Variable name, function name etc.
Operators; Examples '+', '++', '-' etc.
Separators; Examples ',' ';' etc

Example – Sample Programs to get yourself ready for the Lab

Write a C program to recognize strings under 'a*', 'a*b+', 'abb'.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
        char s[20],c;
        int state=0,i=0;
        printf("\n Enter a string:");
        gets(s);
        while(s[i]!='\0')
        {
                switch(state)
                {
                        case 0:
                                c=s[i++];
                                if(c=='a') { state=1; }
                                else if(c=='b')  {state=2;}
                                else  { state=6;}
                        break;
```

```
            case 1:
                c=s[i++];
                if(c=='a')  {state=3;}
                else if(c=='b') { state=4; }
                else { state=6; }
            break;
            case 2:
                c=s[i++];
                if(c=='a') { state=6; }
                else if(c=='b') { state=2;}
                else  { state=6; }
            break;
            case 3:
                c=s[i++];
                if(c=='a') { state=3; }
                else if(c=='b') { state=2; }
                else { state=6; }
            break;
            case 4:
                c=s[i++];
                if(c=='a') { state=6; }
                else if(c=='b') { state=5; }
                else { state=6; }
            break;
            case 5:
                c=s[i++];
                if(c=='a') { state=6; }
                else if(c=='b') { state=2; }
                else { state=6; }
            break;
            case 6:
                printf("\n %s is not recognised.",s); exit(0);
        }
    }
    if(state==1) { printf("\n %s is accepted under rule 'a'",s);}
    else if((state==2)||(state==4)) { printf("\n %s is accepted under rule 'a*b+'",s); }
    else if(state==5) { printf("\n %s is accepted under rule 'abb'",s); }
}
```

INPUT & OUTPUT:

```
Enter a String: aaaabbbbb
aaaabbbbb is accepted under rule 'a*b+'
Enter a string: cdgs
cdgs is not recognized
```

Write a C program to identify whether a given line is a comment or not. (hint : Check whether the string is starting with ‘/’ and check next character is ‘/’ or’*’.)

```
#include<stdio.h>
#include<string.h>
void main()
{
	char com[50];
	int i=2,s=0,e=0;
	printf("\n Enter comment:");
	gets(com);
	for(i=0;i<=strlen(com);i++)
	{
		if(com[i]=='/')
		{
			s=1;
			if(com[i+1]=='/')
			{
				printf("\n It is a comment");
				break;
			}

			else if(com[i+1]=='*')
			{
				for(i=i+2;i<=strlen(com);i++)
				{
					if(com[i]=='*'&&com[i+1]=='/')
					{
						printf("\n It is a comment");
						e=1;
						break;
					}
					else
						continue;
				}
				if(e==0)
					printf("\n It is not a comment");
			}
			else
				printf("\n It is not a comment");

		}
		else
		continue;
	}
	if(s==0)
		printf("\n It is not a comment");
}
```

INPUT & OUTPUT:

```
Enter comment: //hello
It is a comment
Enter comment: hello
It is not a comment
```

Assignment-

1. Write a C program to test whether a given identifier is valid or not.

   (Hint: Check the initial character of the string is numerical or any special character except '_'. Otherwise print it as valid identifier if remaining characters of string doesn't contains any special characters except '_'.)

2. Write a C program to recognize constants and keywords

   (Hint : constants are declared using #define preprocessor or const keyword.)

3. Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Length of identifiers must be restricted to some reasonable value. Simulate the same in C language and print the number of identifiers, symbols and keyword in the given language.

   (Hint : Check whether input is alphabet or digits then store it as identifier . If the input is is operator store it as symbol . Check the input for keywords)

4. Write a C program to analyze a mathematical expression and give the number of digits, brackets and operators present in the expression as output.

   (Hint: A mathematical expression is a combination of digits, operators and brackets)

5. Write a C program to analyze a regular expression and give the number of alphabets, brackets and symbols present in the expression as output.

   (Hint: A regular expression is a combination of alphabets, symbols and brackets)

6. ****Write a program that takes any C Program as an input and extracts all tokes from it. Use flat file store the output.**

   (Hint: A token starts with a char or _ and ends while a space or , or ; is found.The parentheses, braces and brackets are not part of it)

Laboratory – 4

Tokenizing using Lex

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

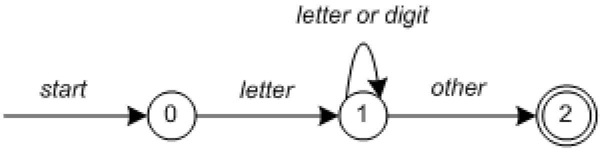
The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

```
letter(letter|digit)*
```

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*" operator
- alternation, expressed by the "|" operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.



In above Figure state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. *Any* FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

```
start:  goto state0

state0: read c
        if c = letter goto state1
        goto state0

state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2

state2: accept string
```

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next *input* character and *current state* the next state is easily determined by indexing into a computer-generated state table.

Table 1: Special Characters

Pattern	Matches
`.`	any character except newline
`\.`	literal .
`\n`	newline
`\t`	tab
`^`	beginning of line
`$`	end of line

Table 2: Operators

Pattern	Matches
`?`	zero or one copy of the preceding expression
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`a|b`	`a` or `b` (alternating)
`(ab)+`	one or more copies of `ab` (grouping)
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`"abc*"`	literal `abc*`
`abc+`	`abc abcc abccc abcccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`

Table 3: Character Class

Pattern	Matches
`[abc]`	one of: `a b c`
`[a-z]`	any letter a through z
`[a\-z]`	one of: `a - z`
`[-az]`	one of: `- a z`
`[A-Za-z0-9]+`	one or more alphanumeric characters
`[ \t\n]+`	whitespace
`[^ab]`	anything except: `a b`
`[a^b]`	one of: `a ^ b`
`[a|b]`	one of: `a | b`

Regular expressions are used for pattern matching. A character class defines a single character and normal operators lose their meaning. Two operators supported in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string the longest match wins. In case both matches are the same length, then the first pattern listed is used.

... *definitions* ...
%%
... *rules* ...
%%
... *subroutines* ...

Input to Lex is divided into three sections with `%%` dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output one character at a time. The first `%%` is always required as there must always be a rules section. However if we don’t specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are `stdin` and `stdout`, respectively. Here is the same example with defaults explicitly coded:

```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by *whitespace* (space, tab or newline) and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "`.`" and "`\n`", with an `ECHO` action associated for each pattern. Several macros and variables are predefined by lex. `ECHO` is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, `ECHO` is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable `yytext` is a pointer to the matched string (NULL-terminated) and `yyleng` is the length of the matched string. Variable `yyout` is the output file and defaults to `stdout`. Function `yywrap` is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a `main` function. In this case we simply call `yylex` that is the main entry-point for lex . Some implementations of lex include copies of `main` and `yywrap` in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Table 4: Lex Predefined Variables

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN condition`	switch start condition
`ECHO`	write matched string

Execution of lex program:

1. write the lex program in a file and save it as file.l (where file is the name of the file).

2. open the terminal and navigate to the directory where you have saved the file.l

3. type - lex file.l

4. then type - cc lex.yy.c -ll

5. then type - ./a.out

Your lex program will be running now.

ASSIGNMENT

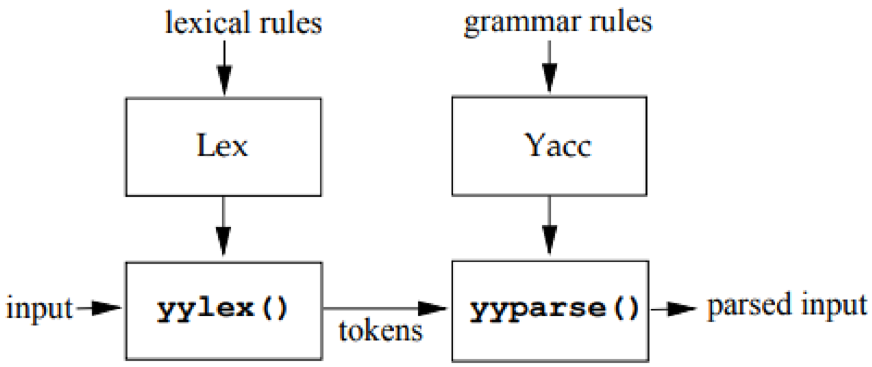
1. Write a LEX program to count the total number of identifier.
2. Write a LEX program to count the number of characters, words, and lines in a file.
3. Write a LEX program to find out the identifiers.
4. Write a LEX program to find out the identifiers, Reserve words and constants.
5. Write a LEX program to Identify Comment Lines
6. Write a LEX program that find the longest string and then reverse it
7. Write a LEX program to change the case of alphabet.
8. Write a LEX program to count the number of vowels and consonants in a string
9. Write a LEX program to Check Valid Email id.
10. Write a LEX program that matches words.
11. Write a LEX program to count the number of Positive numbers, Negative numbers & Fractions.

Laboratory – 5

Integrating LEX with YACC for Evaluation of Arithmetic Expression

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Takes a specification for a CFG, produces an LALR parser.



A YACC source program is structurally similar to a LEX one.

declarations

%%

Grammar rules

%%

routines

Input to yacc is divided into three sections. The *definitions* section consists of token declarations and C code bracketed by "%{" and "%}". The BNF grammar is placed in the *rules* section and user subroutines are added in the *subroutines* section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

```
%token INTEGER
```

This definition declares an `INTEGER` token. Yacc generates a parser in file `y.tab.c` and an include file `y.tab.h`:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex includes this file and utilizes the definitions for token values. To obtain tokens yacc calls `yylex`. Function `yylex` has a return type of `int` that returns a token. Values associated with the token are returned by lex in variable `yylval`. For example,

```
[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }
```

would store the value of the integer in `yylval`, and return token `INTEGER` to yacc. The type of `yylval` is determined by `YYSTYPE`. Since the default type is integer this works well in this case. Token values 0-255 are reserved for character values. For example, if you had a rule such as

```
[-+]       return *yytext;    /* return operator */
```

the character value for minus or plus is returned. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator. Generated token values typically start around 258 because lex reserves several values for end-of-file and error processing. Here is the complete lex input specification for our calculator:

```
%{
    #include "y.tab.h"
    #include <stdlib.h>
    void yyerror(char *);
%}

%%

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+\n]      return *yytext;

[ \t]       ; /* skip whitespace */

.           yyerror("invalid character");
```

```
%%

int yywrap(void) {
    return 1;
}
```

Internally yacc maintains two stacks in memory; a parse stack and a value stack. The parse stack contains terminals and nonterminals that represent the current parsing state. The value stack is an array of `YYSTYPE` elements and associates a value with each element in the parse stack. For example when lex returns an `INTEGER` token yacc shifts this token to the parse stack. At the same time the corresponding `yylval` is shifted to the value stack. The parse and value stacks are always synchronized so finding a value related to a token on the stack is easily accomplished. Here is the yacc input specification for our calculator:

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}


%token INTEGER

%%

program:
        program expr '\n'         { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                   { $$ = $1; }
        | expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified and followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces.

With left-recursion we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected we print the value of the expression. When we apply the rule

```
expr: expr '+' expr              { $$ = $1 + $3; }
```

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case we pop "`expr '+' expr`" and push "`expr`". We have reduced the stack by popping three terms off the stack and pushing back one term. We may reference positions in the value stack in our C code by specifying "`$1`" for the first term on the right-hand side of the production, "`$2`" for the second, and so on. "`$$`" designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. As a consequence the parse and value stacks remain synchronized.

Numeric values are initially entered on the stack when we reduce from `INTEGER` to `expr`. After `INTEGER` is shifted to the stack we apply the rule

```
expr: INTEGER                   { $$ = $1; }
```

The `INTEGER` token is popped off the parse stack followed by a push of `expr`. For the value stack we pop the integer value off the stack and then push it back on again. In other words we do nothing. In fact this is the default action and need not be specified. Finally, when a newline is encountered, the value associated with `expr` is printed.

In the event of syntax errors yacc calls the user-supplied function `yyerror`. If you need to modify the interface to `yyerror` then alter the canned file that yacc includes to fit your needs. The last function in our yacc specification is `main` … in case you were wondering where it was. This example still has an ambiguous grammar. Although yacc will issue shift-reduce warnings it will still process the grammar using shift as the default operation.

Example Program -

Program to evaluate an arithmetic expression involving operators +, -, *,/

Lex Part:

```
%{
#include<stdio.h>
#include"y.tab.h"
extern int yylval;
%}

%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUM;
      }
[\t] ;
\n return 0;
. return yytext[0];
%%
```

Yacc Part:

```
%{
   #include<stdio.h>
%}
%token NUM
%left '+' '-'
%left '*' '/'
%left '(' ')'
%%
expr: e{
         printf("result:%d\n",$$);
         return 0;
        }
e:e'+'e {$$=$1+$3;}
 |e'-'e {$$=$1-$3;}
 |e'*'e {$$=$1*$3;}
 |e'/'e {$$=$1/$3;}
 |'('e')' {$$=$2;}
 | NUM {$$=$1;}
;
%%


main()
{
   printf("\n enter the arithematic expression:\n");
   yyparse();
   printf("\nvalid expression\n");
}
yyerror()
{
   printf("\n invalid expression\n");
   exit(0);
}
```

OUTPUT

```
enter the arithmetic expression: 5+6
result:11
valid expression
```

for compiling lex and yacc together

1. write lex program in a file file.l and yacc in a file file.y

2. open the terminal and navigate to the directory where you have saved the files.

3. type lex_file.l

4. type yacc_file.y -d

5. type cc lex.yy.c y.tab.c -ll

6. type ./a.out

ASSIGNMENT

1. Write a YACC program for Simple calculator.
2. Write a YACC program to handle parenthesis.
3. Write a YACC program to handle binary and order of operations
4. Write a YACC program to handle conditional statements.
5. Write a YACC program to handle Boolean expressions.
6. Write a program that will check whether an arithmetic expression is valid or not and evaluate it when it is valid, else post an error.
7. Write a program to recognize valid arithmetic expression and identify the identifiers and operators present.
8. Write a LEX program t**o check the validity of date.**

Laboratory – 6

Finding First and Follow of a given grammar

first(x) for all grammar symbols x
Apply following rules:
If X is terminal, FIRST(X) = {X}.
If X → ε is a production, then add ε to FIRST(X).
If X is a non-terminal, and X → Y_1 Y_2 … Y_k is a production, and ε is in all of FIRST(Y_1), …, FIRST(Y_k), then add ε to FIRST(X).
If X is a non-terminal, and X → Y_1 Y_2 … Y_k is a production, then add a to FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1), …, FIRST(Y_{i-1}).
Applying rules 1 and 2 is obvious. Applying rules 3 and 4 for FIRST(Y_1 Y_2 … Y_k) can be done as follows:
Add all the non-ε symbols of FIRST(Y_1) to FIRST(Y_1 Y_2 … Y_k). If ε ∈ FIRST(Y_1), add all the non-ε symbols of FIRST(Y_2). If ε ∈ FIRST(Y_1) and ε ∈ FIRST(Y_2), add all the non-ε symbols of FIRST(Y_3), and so on. Finally, add ε to FIRST(Y_1 Y_2 … Y_k) if ε ∈ FIRST(Y_i), for all $1 \leq i \leq k$.
Example:
Consider the following grammar.
E → E + T | T
T → T * F | F
F → (E) | id
Grammar after removing left recursion:
E → TX
X → +TX | ε
T → FY
Y → *FY | ε
F → (E) | id
For the above grammar, following the above rules, the FIRST sets could be computed as follows:
FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST(X) = {+, ε}
FIRST(Y) = {*, ε}
follow(a) for all non-terminals a
Apply the following rules:
If $ is the input end-marker, and S is the start symbol, $ ∈ FOLLOW(S).
If there is a production, A → αBβ, then (FIRST(β) – ε) ⊆ FOLLOW(B).
If there is a production, A → αB, or a production A → αBβ, where ε ∈ FIRST(β), then FOLLOW(A) ⊆ FOLLOW(B).
Note that unlike the computation of FIRST sets for non-terminals, where the focus is on *what a non-terminal generates*, the computation of FOLLOW sets depends upon *where the non-terminal appears on the RHS of a production*.
Example:
For the above grammar, the FOLLOW sets can be computed by applying the above rules as follows.
FOLLOW(E) = {$,)}
FOLLOW(E) ⊆ FOLLOW(X) [in other words, FOLLOW(X) contains FOLLOW(E)]
Since there is no other rule applicable to FOLLOW(X),

FOLLOW(X) = {$,)}
FOLLOW(T) ⊆ FOLLOW(Y) …. (1)
(FIRST(X) – ε) ⊆ FOLLOW(T) i.e., {+} ⊆ FOLLOW(T) …. (2)
Also, since ε ∈ FIRST(X), FOLLOW(E) ⊆ FOLLOW(T)
i.e., {$,)} ⊆ FOLLOW(T) …. (3)
Putting (2) and (3) together, we get:
FOLLOW(T) = {$,), +}
Since, there is no other rule applying to FOLLOW(Y), from (1), we get:
FOLLOW(Y) = {$,), +}
Since ε ∈ FIRST(Y), FOLLOW(T) ⊆ FOLLOW(F) and FOLLOW(Y) ⊆ FOLLOW(F). Also, (FIRST(Y) – ε) ⊆ FOLLOW(F). Putting all these together:
FOLLOW(F) = FOLLOW(T) ∪ FOLLOW(Y) ∪ (FIRST(Y) – ε) = {$,), +, *}

A suitable data structure for storing a graph may look like:

```
typedef struct production{
            char var;
            char right[10];
}Production;

typedef struct grammar{
           int no_of_prod;
          Production *prod;
}Grammar;
```

Assignment –

1. Write a C program to find first and follow of a given context-free grammar.
2. Construct First and follow of the grammar
   S→aABb
   A→c | ε
   B→d | ε
3. Construct First and follow of the grammar
   E→TE’
   E’→+TE’ | ε
   T→FT’
   T’→*FT’ | ε
   F→(E) | a
4. Construct First and follow of the grammar
   S→aAB |bA | ε
   A→aAb | ε
   B→bB | ε

Laboratory – 7

Simulation of LL(1) parsing using JFLAP

The type of LL parsing in JFLAP is LL(1) Parsing.

The first L means the input string is processed from left to right.
The second L means the derivation will be a leftmost derivation (the leftmost variable is replaced at each step).
1 means that one symbol in the input string is used to help guide the parse.

We will begin by loading the grammar.

Enter FIRST Sets

Now, the first step is to enter in **FIRST** sets. A **FIRST** set of a string of variables and terminals is the set of terminals that can first appear in the derived string.

***NOTE** : When you want to enter two terminals into a set, just write them without any spaces. For example in order to put lambda and terminal "a" into the **FIRST** set of "A", you would simply type "!a" in the appropriate column.

Enter FOLLOW Sets

Now, we need to enter the proper **FOLLOW** sets for each variable as we did for the **FIRST** sets. The **FOLLOW** set of variable "X" is a set of terminals that can immediately follow "X" in some derivation.

***NOTE** : $ is a special marker that is always in the **FOLLOW** set of the start variable. This marker is placed at the end of a string to indicate the end of the string in parsing.
The **FOLLOW** set for start variable "S" is only $, since there are no variables or terminals that follow start variable "S".

After entering all the **FOLLOW** sets, click **Next**.

Enter LL(1) Parse Table

There are two main rules for filling in the table. 1) For production "A->w", for each terminal in **FIRST**(w), put w as an entry in the A row under the column for that terminal. 2) For production "A->w", if lambda is in **FIRST**(w), put w as an entry in the A row under the columns for all symbols in the **FOLLOW**(A).

We can apply these rules to all of the productions and we end up with the parse table :

Now we are finished with creating LL(1) Parse table, we can use this table to parse. Click on **Parse** button.

LL(1) Parsing

Now, let's parse the string "aacbbcb". Enter this string in the **Input** text box and click on **Start**.

After clicking **Start**, at first nothing is going to show on the tree panel. However, JFLAP will notify you what input string is remaining to be parsed.

Note the input string is listed by "Input Remaining" with the special ending symbol $ added to the right end. Also note the start variable "S" has been placed on the stack.

The variable "S" has been popped off the stack and is going to be replaced by its right-side. The red highlighted portion in the parse table tells you that JFLAP is going to apply that production to the current variable, in this case "S". Also, there is a message at the bottom of the window which tells you what production JFLAP is applying to the variable.

Assignment –

1. Consider the predictive parsing table for the following grammar and show the stack implementation for the input string a+a*a.

   E→TE'

   E'→+TE' | ε

   T→FT'

   T'→*FT' | ε

   F→(E) | a

2. Construct LL(1) parsing table for the following grammar & parse the string (a,a)

   S→(L) | a

   L→L,S | S

3. Construct LL(1) parsing table for the following grammar

   S→aAB |bA | ε
   A→aAb | ε
   B→bB | ε

Laboratory – 8

Simulation of SLR(1) parsing using JFLAP

The type of LR parsing in JFLAP is SLR(1) Parsing.

S stands for simple LR.
L means the input string is processed from left to right, as in LL(1) parsing.
R means the derivation will be a rightmost derivation (the rightmost variable is replaced at each step).
1 means that one symbol in the input string is used to help guide the parse.

How to Build SLR(1) Parse Table

We will begin by loading the same grammar that we used for building the LL(1) Parse Table.

After loading the grammar, click on **Input** and then **Build SLR(1) Parse Table**. A new rule is automatically added to the grammar, “S'->S”, resulting in a new start variable, “S'”, that does not appear in any right-hand sides.

Just like LL(1) Parse, we have to enter the **FIRST** and **FOLLOW** sets. After entering proper **FIRST** and **FOLLOW** sets, click on **Next**.

Building the DFA

We now build a DFA that models the SLR(1) process. Each state will have a label with marked productions called items. The marker is shown as “.”. Those symbols to the left of the marker in an item are already on the parsing stack, and those symbols to the right of the marker still need to be pushed on the parsing stack. All the items for one state are called an item set.

JFLAP now asks you whether you want to define the initial set or not. If you click “Yes”, JFLAP will pop up a small window where you can define the initial set. If you click “No”, then JFLAP will automatically create the initial set for you.

When you click on a production, you will notice that all possible items derivable from that production appear. Since nothing is added to the stack yet, click on the production “S'-> .S”. Since the marker is in front of a variable, “S”, all of the S items with the “.” as the leftmost symbol must be added to the item set.

You now have your first state “q0” created.

We should add more states and transitions to complete this diagram. First click on the “Goto Set on Symbol” button on the diagram panel, it is the button next to the “arrow” button. From our initial state q0, we know there are only two transitions possible. One is with variable S and the other one is with terminal “a”, since both of these symbols come right after the marker in the item set.

After clicking the “Goto Set on Symbol” button, click on our initial state and drag the mouse and release it at the spot you want to create a new state. Next, JFLAP asks you to enter the symbol for the transition. Let's put variable “S”. When you enter the variable S, you will see the window “Goto on S”. Now click on the production “S'->S.”, since we have moved from our initial state using input “S”, variable “S” is no longer behind the marker. It is now on the left side of the marker. Since there are no variables remaining on the right side of the marker, there is no transition outgoing from the new state.

A state is a final state if one of its items has the marker as the last symbol on the right-side of the production. Since we now know that q1 is one of our final states, we can click the attribute button (“arrow” button), then right click on q1. We can select “final” to make the state q1 final. Try adding the remaining states and their item sets.

Building the Parse Table

The DFA is used to build the parse table. Each transition in the DFA results in either a shift entry for a terminal or a goto entry for a variable. Each marked rule with the marker at the right end (found in final states) results in reduce rules.

Now, it's time to fill in entries in the parse table. We can start filling in the parse table from row 0, which corresponds to state q0. From state q0, there are two transitions resulting in two operations. The transition on the “S” relates to “go to” state 1 (q1) and the transition on the “a” relates to “shift” terminal “a” and go to state 2 (q2). Therefore, in the parse table row 0, put s2 (“s” stands for “shift”) in column “a” and put 1 (means just go to state q1) in column S. Now we are done filling the parse table for state q0. Next step is filling in the parse table for q1, which corresponds to row 1. When we reach state q1, we are done parsing since start variable “S” is popped off the stack. So, at state q1, we should accept the string. We fill the entry by typing “acc”, which stands for accepted in column “$”, which implies there are no symbols left to process.

***NOTE** : There are 4 operations that we use to fill in the parse table : “s” (shift) and “r” (reduce) followed by the destination row number, “acc” for accept, and just a destination row number to indicate transition using variable.

SLR(1) Parsing

We have finished the SLR (1) Parse table. Next, click on **Parse** and in the SLR (1) Parsing window, input “aacbbcb” as the input string. After clicking **Start**, press **Step**.

Notice that we have a node with terminal “a” in the tree panel. We achieved the first “a”, so we would remove this “a” from the input remaining. The stack is now filled with “2a0”, meaning that we have reached state 2 from state 0 using transition terminal “a”. You will also notice that the operation that is applied to the input string is highlighted in the parse table, “s2” in the “a” column. Whenever a reduction is applied, both the entry in the parse table and the production that actually performed the reduction will be highlighted. Click **Step** several times, observing what happens until the string is accepted.

Assignment –

1. Consider the SLR parsing table for the following grammar and show the stack implementation for the input string a+a*a.

   E→TE’

   E’→+TE’ | ε

   T→FT’

   T’→*FT’ | ε

   F→(E) | a

2. Construct SLR parsing table for the following grammar & parse the string (a,(a,a))

   S→(L) | a

   L→L,S | S

3. Show the following grammar

   S→AaAb | BbBa

   A→ ε
   B→ ε
   Is LL(1) but not SLR(1)