



LABORATORY MANUAL

B.Tech. Semester- VI

COMPILER DESIGN LAB

Subject code: LC-CSE-324G

Prepared by:

Prof. Vimmi Malhotra

Checked by:

Dr. Ashima Mehta

Approved by:

Name : Prof. (Dr.) Isha Malhotra

Sign.:

Sign.:

Sign.:

**DEPARTMENT OF CSE/CSIT/IT/IOT
DRONACHARYA COLLEGE OF ENGINEERING
KHENTAWAS, FARRUKH NAGAR, GURUGRAM (HARYANA)**

Table of Contents

1. Vision and Mission of the Institute
2. Vision and Mission of the Department
3. Programme Educational Objectives (PEOs)
4. Programme Outcomes (POs)
5. Programme Specific Outcomes (PSOs)
6. University Syllabus
7. Course Outcomes (COs)
8. CO- PO and CO-PSO mapping
9. Course Overview
10. List of Experiments
11. DOs and DON'Ts
12. General Safety Precautions
13. Guidelines for students for report preparation
14. Lab assessment criteria
15. Details of Conducted Experiments
16. Lab Experiments

Vision and Mission of the Institute

Vision:

To impart Quality Education, to give an enviable growth to seekers of learning, to groom them as World Class Engineers and managers competent to match the expending expectations of the Corporate World has been ever enlarging vision extending to new horizons of Dronacharya College of Engineering

Mission:

1. To prepare students for full and ethical participation in a diverse society and encourage lifelong learning by following the principle of 'Shiksha evam Sahayata' i.e. Education & Help.
2. To impart high-quality education, knowledge and technology through rigorous academic programs, cutting-edge research, & Industry collaborations, with a focus on producing engineers& managers who are socially responsible, globally aware, & equipped to address complex challenges.
3. Educate students in the best practices of the field as well as integrate the latest research into the academics.
4. Provide quality learning experiences through effective classroom practices, innovative teaching practices and opportunities for meaningful interactions between students and faculty.
5. To devise and implement programmes of education in technology that are relevant to the changing needs of society, in terms of breadth of diversity and depth of specialization.

Vision and Mission of the Department

Vision:

“To become a Centre of Excellence in teaching and research in Information Technology for producing skilled professionals having a zeal to serve society”

Mission:

M1: To create an environment where students can be equipped with strong fundamental concepts, programming and problem-solving skills.

M2: To provide an exposure to emerging technologies by providing hands on experience for generating competent professionals.

M3: To promote Research and Development in the frontier areas of Information Technology and encourage students for pursuing higher education

M4: To inculcate in students ethics, professional values, team work and leadership skills.

Programme Educational Objectives (PEOs)

PEO1: To provide students with a sound knowledge of mathematical, scientific and engineering fundamentals required to solve real world problems.

PEO2: To develop research oriented analytical ability among students and to prepare them for making technical contribution to the society.

PEO3: To develop in students the ability to apply state-of-the-art tools and techniques for designing software products to meet the needs of Industry with due consideration for environment friendly and sustainable development.

PEO4: To prepare students with effective communication skills, professional ethics and managerial skills.

PEO5: To prepare students with the ability to upgrade their skills and knowledge for life-long learning.

Programme Outcomes (POs)

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

PSO1: Analyze, identify and clearly define a problem for solving user needs by selecting, creating and evaluating a computer-based system through an effective project plan.

PSO2: Design, implement and evaluate processes, components and/or programs using modern techniques, skills and tools of core Information Technologies to effectively integrate secure IT-based solutions into the user environment.

PSO3: Develop impactful IT solutions by using research-based knowledge and research methods in the fields of integration, interface issues, security & assurance and implementation.

University Syllabus

1. Write a Program for Token separation with a given expression.
2. Write a Program for Token separation with a given file.
3. Write a Program for Lexical analysis using LEX tools.
4. Write a Program to identify whether a given line is a comment or not.
5. Write a Program to check whether a given identifier is valid or not.
6. Write a Program to recognize strings under 'a', 'a*b+', 'abb'.
7. Write a Program to simulate lexical analyzer for validating operators.
8. Write a Program for implementation of Operator Precedence Parser.
9. Study of LEX and YACC tools:
 - (i) Write a Program for implementation of calculator using YACC tool.
 - (ii) Write a Program for implementation of Recursive Descent Parser using LEX tool.
10. Write a Program for implementation of LL (1) Parser.
11. Write a Program for implementation of LALR Parser.

Course Outcomes (COs)

Upon successful completion of the course, the students will be able to:

C324.1: Help in improving the programming skills of the students.

C324.2: The implementation of different parsers.

C324.3: Acquire knowledge of different phases of compiler.

C324.4: Able to use the compiler tools like LEX, YACC, etc.

C324.5: Identify different types of grammars.

CO-PO Mapping

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C324.1	3	3	1		3	1	3		3	2	1	3
C324.2	3	2	1	1	3	1			3			3
C324.3	2	2		1	3		1	1		1		2
C324.4	2	2	3		3	1	1		2	1	1	2
C324.5	1	3	3	1	3	1		1	1			
C324	2.2	2.4	1.8	1	3	1	1	1	1.8	1	1	2

CO-PSO Mapping

	PSO1	PSO2	PSO3
C324.1	1	3	2
C324.2	1	3	1
C324.3	1	3	2
C324.4	1	3	1
C324.5	1	3	2
C324	1.8	3	1.6

Course Overview

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult and cumbersome task for computer programmers to write such codes, which is why we have compilers to write such codes.

This course is intended to teach his course covers all the phases of a compiler such as lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, target code generation, symbol table and error handler in details.

Compilers have become part and parcel of today's computer systems. They are responsible for making the user's computing requirements, specified as a piece of program, understandable to the underlying machine. There tools work as interface between the entities of two different domains – the human being and the machine. The actual process involved in this transformation is quite complex. Automata Theory provides the base of the course on which several automated tools can be designed to be used at various phases of a compiler. Advances in computer architecture, memory management and operating systems provide the compiler designer large number of options to try out for efficient code generation. This course on compiler design is to address all these issues, starting from the theoretical foundations to the architectural issues to automated tools.

List of Experiments mapped with COs

S No.	Name of the Experiment	Course Outcome
1	Write a Program for Token separation with a given expression.	C324.1, C324.3
2	Write a Program for Token separation with a given file.	C324.3
3	Write a Program for Lexical analysis using LEX tools.	C324.4
4	Write a Program to identify whether a given line is a comment or not.	C324.3
5	Write a Program to check whether a given identifier is valid or not.	C324.2, C324.1
6	Write a Program to recognize strings under 'a', 'a*b+', 'abb'.	C324.3
7	Write a Program to simulate lexical analyser for validating operators.	C324.5
8	Write a Program for implementation of Operator Precedence Parser.	C324.4
9	Study of LEX and YACC tools: (i) Write a Program for implementation of calculator using YACC tool. (ii) Write a Program for implementation of Recursive Descent Parser using LEX tool.	C324.3, C324.2
10	Write a Program for implementation of LL (1) Parser.	C324.3
11	Write a Program for implementation of LALR Parser.	C324.3

DOs and DON'Ts

DOs

1. Login-on with your username and password.
2. Log off the computer every time when you leave the Lab.
3. Arrange your chair properly when you are leaving the lab.
4. Put your bags in the designated area.
5. Ask permission to print.

DON'Ts

1. Do not share your username and password.
2. Do not remove or disconnect cables or hardware parts.
3. Do not personalize the computer setting.
4. Do not run programs that continue to execute after you log off.
5. Do not download or install any programs, games or music on computer in Lab.
6. Personal Internet use chat room for Instant Messaging (IM) and Sites is strictly prohibited.
7. No Internet gaming activities allowed.
8. Tea, Coffee, Water & Eatables are not allowed in the Computer Lab.

General Safety Precautions

Precautions (In case of Injury or Electric Shock)

1. To break the victim with live electric source, use an insulator such as fire wood or plastic to break the contact. Do not touch the victim with bare hands to avoid the risk of electrifying yourself.
2. Unplug the risk of faulty equipment. If main circuit breaker is accessible, turn the circuit off.
3. If the victim is unconscious, start resuscitation immediately, use your hands to press the chest in and out to continue breathing function. Use mouth-to-mouth resuscitation if necessary.
4. Immediately call medical emergency and security. Remember! Time is critical; be best.

Precautions (In case of Fire)

1. Turn the equipment off. If power switch is not immediately accessible, take plug off.
2. If fire continues, try to curb the fire, if possible, by using the fire extinguisher or by covering it with a heavy cloth if possible isolate the burning equipment from the other surrounding equipment.
3. Sound the fire alarm by activating the nearest alarm switch located in the hallway.
4. Call security and emergency department immediately:

Emergency	:	Reception
(Reception) Security	:	Front Gate

Guidelines to students for report preparation

All students are required to maintain a record of the experiments conducted by them. Guidelines for its preparation are as follows: -

- 1) All files must contain a title page followed by an index page. *The files will not be signed by the faculty without an entry in the index page.*
- 2) Student's Name, Roll number and date of conduction of experiment must be written on all pages.
- 3) For each experiment, the record must contain the following
 - (i) Aim/Objective of the experiment
 - (ii) Pre-experiment work (as given by the faculty)
 - (iii) Lab assignment questions and their solutions
 - (iv) Test Cases (if applicable to the course)
 - (v) Results/ output

Note:

1. Students must bring their lab record along with them whenever they come for the lab.
2. Students must ensure that their lab record is regularly evaluated.

Lab Assessment Criteria

An estimated 10 lab classes are conducted in a semester for each lab course. These lab classes are assessed continuously. Each lab experiment is evaluated based on 5 assessment criteria as shown in following table. Assessed performance in each experiment is used to compute CO attainment as well as internal marks in the lab course.

Grading Criteria	Exemplary (4)	Competent (3)	Needs Improvement (2)	Poor (1)
AC1: Pre-Lab written work (this may be assessed through viva)	Complete procedure with underlined concept is properly written	Underlined concept is written but procedure is incomplete	Not able to write concept and procedure	Underlined concept is not clearly Understood
AC2: Program Writing/ Modeling	Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied, Program/solution written is readable	Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied	Assigned problem is properly analyzed & correct solution designed	Assigned problem is properly analyzed
AC3: Identification & Removal of errors/ bugs	Able to identify errors/ bugs and remove them	Able to identify errors/ bugs and remove them with little bit of guidance	Is dependent totally on someone for identification of errors/ bugs and their removal	Unable to understand the reason for errors/ bugs even after they are explicitly pointed out
AC4: Execution & Demonstration	All variants of input /output are tested, Solution is well demonstrated and implemented concept is clearly explained	All variants of input /output are not tested, However, solution is well demonstrated and implemented concept is clearly explained	Only few variants of input /output are tested, Solution is well demonstrated but implemented concept is not clearly explained	Solution is not well demonstrated and implemented concept is not clearly explained
AC5: Lab Record Assessment	All assigned problems are well recorded with objective, design constructs and solution along with Performance analysis using all variants of input and output	More than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output	Less than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output	Less than 40 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output

LAB EXPERIMENTS

LAB EXPERIMENT 1

OBJECTIVE:

Write a Program for Token separation with a given expression.

BRIEF DESCRIPTION:

Token is a group of characters having collective meaning: typically, a word or punctuation mark, separated by a lexical analyzer and passed to a parser. A lexeme is an actual character sequence forming a specific instance of a token, such as num.

The basic token includes:

- Terminal Symbols (TRM)- Keywords and Operators,
- Literals (LIT), and
- Identifiers (IDN).

The output of Lexical Analyzer serves as an input to Syntax Analyzer as a sequence of tokens and not the series of lexemes because during the syntax analysis phase individual unit is not vital but the category or class to which this lexeme belongs is considerable.

PRE-EXPERIMENT QUESTIONS:

1. What is tokens and explain its category.
2. What is lexical analysis in compiler design?
3. What is expression?

Explanation:

```
int a = 10; //Input Source code
```

Tokens:

int (keyword), a(identifier), =(operator), 10(constant) and ;(punctuation-semicolon)

Answer: Total number of tokens = 5

Program Code:

```

#include <stdio.h>
#include <string.h>
#include <ctype.h>
void main()
{
    int i, ic=0, m, cc=0, oc=0, j;
    char b[30], operator[30], identifier[30], constant[30];
    printf("Enter the string: ");
    scanf("%s", &b);
    for(int i=0; i<strlen(b); i++){
        if(isspace(b[i])){
            continue;
        }
        else if(isalpha(b[i])){
            identifier[ic] = b[i];
            ic++;
        }
        else if(isdigit(b[i])){
            m = (b[i]-'0');
            i = i+1;
            while(isdigit(b[i])){
                m = m*10+(b[i]-'0');
                i++;
            }
            i = i-1;
            constant[cc] = m;
            cc++;
        }
        else{
            if(b[i]=='*'){
                operator[oc] = '*';
                oc++;
            }
            else if(b[i]=='+'){
                operator[oc] = '+';
                oc++;
            }
            else if(b[i]=='-'){
                operator[oc] = '-';
                oc++;
            }
            else if(b[i]=='='){
                operator[oc] = '=';
                oc++;
            }
        }
    }
    printf("Identifier: ");
    for(int i=0; i<ic; i++){

```

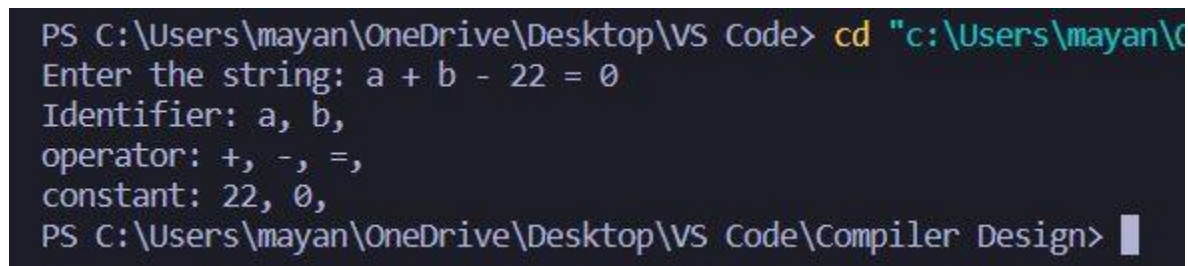
```

    printf("%c, ", identifier[i]);
}
printf("\n");
printf("operator: ");
for(int i=0;i<oc;i++){
    printf("%c, ", operator[i]);
}
printf("\n");
printf("constant: ");
for(int i=0;i<cc;i++){
    printf("%d, ", constant[i]);
}
}
}

```

Output:

Students will be able to recognize tokens as shown below:



```

PS C:\Users\mayan\OneDrive\Desktop\VS Code> cd "c:\Users\mayan\
Enter the string: a + b - 22 = 0
Identifier: a, b,
operator: +, -, =,
constant: 22, 0,
PS C:\Users\mayan\OneDrive\Desktop\VS Code\Compiler Design>

```

POST EXPERIMENT QUESTIONS:

1. Differentiate between Tokens, Lexemes, and Pattern?
2. How many tokens are there in given expression: printf (“%d %d”, hello”); ?
3. How are regular sets different from non-regular sets?

LAB EXPERIMENT 2

OBJECTIVE:

Write a Program for Token separation with a given file.

BRIEF DESCRIPTION:

Token separation at phone, syllable and word levels. A Separator is made of 3 entries phone, syllable and word defining the token separators for each of these levels within an utterance. A token separator can be a string or None. If not None, the entries 'phone', 'syllable' and 'word' must be all different.

PRE-EXPERIMENT QUESTIONS:

1. What is token separation?
2. Difference between tokens and terminals.
3. How to count total number of tokens?

Explanation:

Scanning is the first phase of a compiler in which the source program is read character by character and then grouped in to various tokens. Token is defined as sequence of characters with collective meaning. The various tokens could be identifiers, keywords, operators, punctuations, constants, etc. The input is a program written in any high level language and the output is stream of tokens. Regular expressions can be used for implementing this token separation

Algorithm

Step 1: Read the content of the file using File Reader.

Step 2: Separate the string with the delimiter space using String Tokenizer.

Step 3: Match the String with the pattern using Regular Expression.

Step 4: Group the tokens as identifier, keywords, operators , punctuations etc... and display it using the given format <keyword, int>

Program Code:

```

#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

bool isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ';' || ch == ':' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')')
        return (true);
    return (false);
}

bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}

bool validIdentifier(char* str)
{
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isDelimiter(str[0]) == true)
        return (false);
    return (true);
}

bool isKeyword(char* str)
{
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
        !strcmp(str, "continue") || !strcmp(str, "int")
        || !strcmp(str, "double") || !strcmp(str, "float")
        || !strcmp(str, "return") || !strcmp(str, "char")
        || !strcmp(str, "case") || !strcmp(str, "char")
        || !strcmp(str, "sizeof") || !strcmp(str, "long")
        || !strcmp(str, "short") || !strcmp(str, "typedef")
        || !strcmp(str, "switch") || !strcmp(str, "unsigned")
        || !strcmp(str, "void") || !strcmp(str, "static")
        || !strcmp(str, "struct") || !strcmp(str, "goto"))
        return (true);
}

```

```

    return (false);
}

bool isInteger(char* str)
{
    int i, len = strlen(str);

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}

```

```

bool isRealNumber(char* str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' && str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

```

```

char* subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc(
        sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

```

```

void parse(char* str)
{
    int left = 0, right = 0;
    int len = strlen(str);

    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;

        if (isDelimiter(str[right]) == true && left == right) {
            if (isOperator(str[right]) == true)
                printf("%c' IS AN OPERATOR\n", str[right]);

            right++;
            left = right;
        }
        else if (isDelimiter(str[right]) == true && left != right || (right == len && left != right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("%s' IS A KEYWORD\n", subStr);

            else if (isInteger(subStr) == true)
                printf("%s' IS AN INTEGER\n", subStr);

            else if (isRealNumber(subStr) == true)
                printf("%s' IS A REAL NUMBER\n", subStr);

            else if (validIdentifier(subStr) == true
                && isDelimiter(str[right - 1]) == false)
                printf("%s' IS A VALID IDENTIFIER\n", subStr);

            else if (validIdentifier(subStr) == false
                && isDelimiter(str[right - 1]) == false)
                printf("%s' IS NOT A VALID IDENTIFIER\n", subStr);
            left = right;
        }
    }
    return;
}

int main()
{
    char str[100] = " int a = b + 1c; ";
    parse(str);
    return (0);
}

```

Output:

Students will be able to recognize tokens in a file as shown below:

```
PS C:\Users\mayan\OneDrive\Desktop\VS Code> cd "c:\Users\maya
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'1c' IS NOT A VALID IDENTIFIER
PS C:\Users\mayan\OneDrive\Desktop\VS Code\Compiler Design>
```

POST EXPERIMENT QUESTIONS:

1. What are the various functions of lexical analyzer?
2. Which mathematical model is used in the lexical analyzer?
3. What elements are included in non-tokens components?

LAB EXPERIMENT 3

OBJECTIVE:

Write a Program for Lexical analysis using LEX tools.

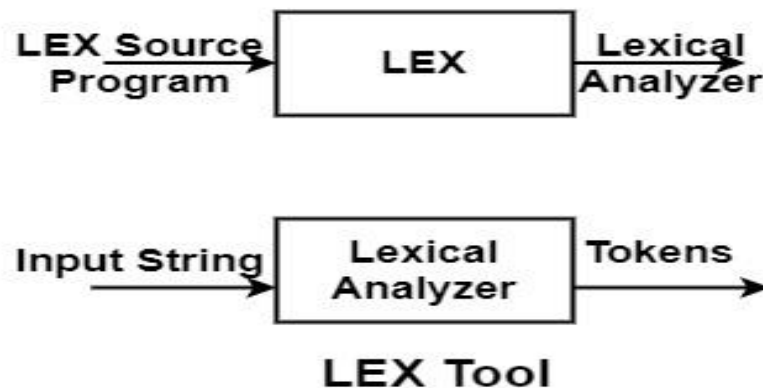
BRIEF DESCRIPTION:

LEX is a program generator designed for lexical processing of character input/output stream. Anything from simple text search program that looks for pattern in its input-output file to a C compiler that transforms a program into optimized code.

PRE-EXPERIMENT QUESTIONS:

1. What is LEX?
2. Differentiate between LEX and FLEX.
3. Define phases of LEX.

Explanation:



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 in C language named Lex.yy.c
3. The program lex.yy.c consists of a table constructed from the Regular Expressions of Lexp.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 program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

Program Code:

```
% {
    #include <stdio.h>
% }

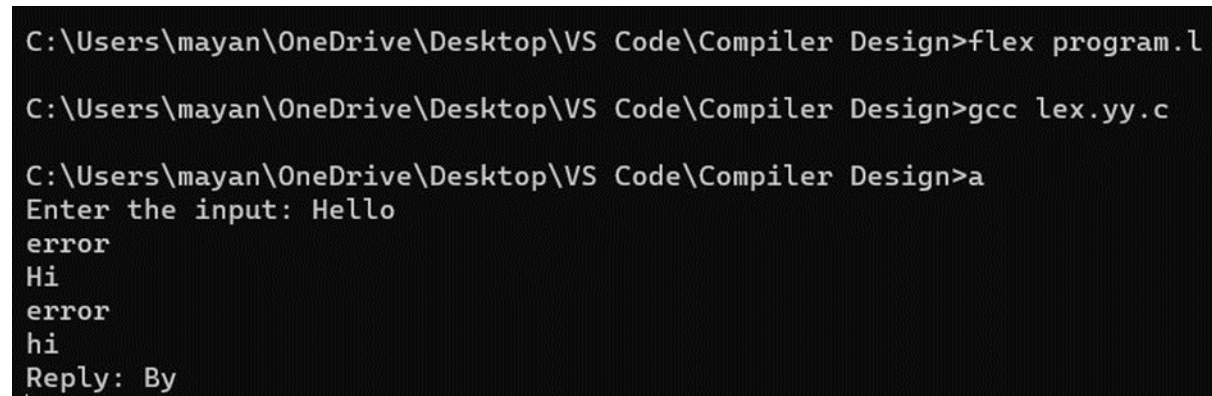
%%
"hi" {
    printf("Reply: By");
}
.* {
    printf("error");
}
%%

int main(){
    printf("Enter the input: ");
    yylex();
}

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

Output:

Students will be able to do programming in LEX as shown below:



```
C:\Users\mayan\OneDrive\Desktop\VS Code\Compiler Design>flex program.l
C:\Users\mayan\OneDrive\Desktop\VS Code\Compiler Design>gcc lex.yy.c
C:\Users\mayan\OneDrive\Desktop\VS Code\Compiler Design>a
Enter the input: Hello
error
Hi
error
hi
Reply: By
```

POST EXPERIMENT QUESTIONS:

1. What is Lex tool used for?
2. Why do we need a lexical analyzer?
3. What is a lexical error?

LAB EXPERIMENT 4

OBJECTIVE:

Write a Program to identify whether a given line is a comment or not.

BRIEF DESCRIPTION:

Comment lines are of the form:

%% {START} {whatever symbols, alphanumeric and letters} {END} %%

START symbol can be // or /* in C.

END symbol is */

If it is "/" any other symbol can follow it in the LINE including a second occurrences of // (which is ignored). IF it is "/* " it will find the next immediate match for "*/ " and all the rest of the patterns that appear in between is matched (This matching may include blank spaces, tabs, "/" and "/* " in it)

Given a string S, representing a line in a program, the task is to check if the given string is a comment or not. The following are the types of comments in programs:

- a) Single Line Comment: Comments preceded by a Double Slash ('//')
- b) Multi-line Comment: Comments starting with ('/*') and ending with ('*/').

PRE-EXPERIMENT QUESTIONS:

1. What are the different types of comment?
2. Why lexical and syntax analyzers are separated out?
3. What is yylex in lex?

Explanation:

The idea is to check whether the input string is a comment or not. Below are the steps:

1. Check if at the first Index (i.e. index 0) the value is '/' then follow below steps else print "It is not a comment".
2. If line[0] == '/':
 - a) If line[1] == '/', then print "It is a single line comment".
 - b) If line[1] == '*', then traverse the string and if any adjacent pair of '*' & '/' is found then print "It is a multi-line comment".
3. Otherwise, print "It is not a comment".

Program Code:

```
% {
#include<stdio.h>
% }

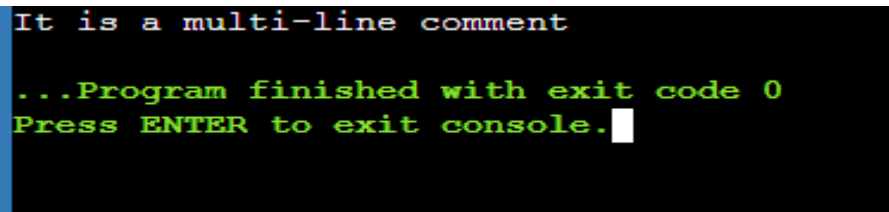
%%
[\t ]+ ;
[0-9]+|[0-9]*\.[0-9]+ { printf("\n%s is NUMBER", yytext);}
#.* { printf("\n%s is COMMENT", yytext);}
[a-zA-Z][a-zA-Z0-9]+ { printf("\n%s is IDENTIFIER", yytext);}
\"[^\n]*\" { printf("\n%s is STRING", yytext);}
\n { ECHO;}
%%

int main()
{
    while( yylex());
}

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

Output:

Students will be able to identify comment in LEX as shown below:



```
It is a multi-line comment
...Program finished with exit code 0
Press ENTER to exit console. █
```

POST EXPERIMENT QUESTIONS:

1. What is the structure of Lex and YACC?
2. In which phase yacc is used?
3. What are the uses of Lex in compiler design?

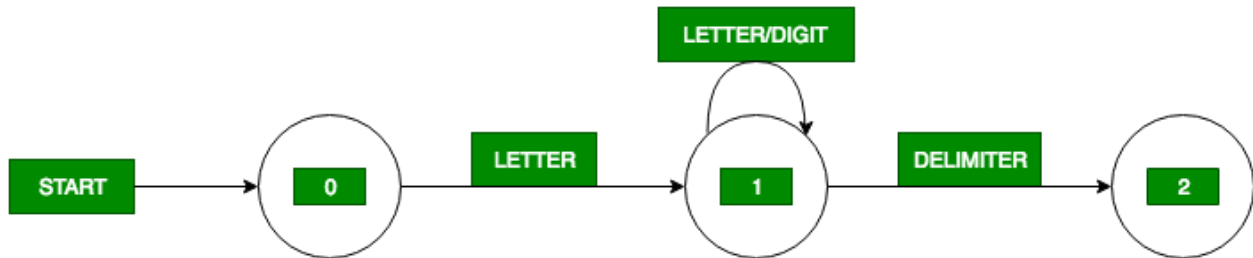
LAB EXPERIMENT 5

OBJECTIVE:

Write a Program to check whether a given identifier is valid or not.

BRIEF DESCRIPTION:

Transition diagram is a special kind of flowchart for language analysis. In transition diagram the boxes of flowchart are drawn as circle and called as states. States are connected by arrows called as edges. The label or weight on edge indicates the input character that can appear after that state. Transition diagram of identifier is given below:



Given a string str, the task is to check if the string is a valid identifier or not. In order to qualify as a valid identifier, the string must satisfy the following conditions:

- a) It must start with an either underscore (`_`) or any of the characters from the ranges [`'a'`, `'z'`] and [`'A'`, `'Z'`].
- b) There must not be any white space in the string.
- c) And, all the subsequent characters after the first character must not consist of any special characters like \$, #, % etc.

PRE-EXPERIMENT QUESTIONS:

1. What is identifier and its types?
2. What are basic identifiers?
3. What is difference between identifier and variable?

Explanation:

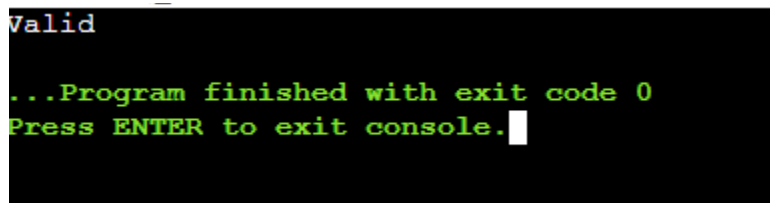
Traverse the string character by character and check whether all the requirements are met for it to be a valid identifier i.e. first character can only be either `'_'` or an English alphabet and the rest of the characters must neither be a white space or any special character.

Program Code:

```
#include <bits/stdc++.h>
using namespace std;
bool isValid(string str, int n)
{
    // If first character is invalid
    if (!(str[0] >= 'a' && str[0] <= 'z')
        || (str[0] >= 'A' && str[0] <= 'Z')
        || str[0] == '_'))
        return false;
    // Traverse the string for the rest of the characters
    for (int i = 1; i < str.length(); i++) {
        if (!(str[i] >= 'a' && str[i] <= 'z')
            || (str[i] >= 'A' && str[i] <= 'Z')
            || (str[i] >= '0' && str[i] <= '9')
            || str[i] == '_'))
            return false;
    }
    // String is a valid identifier
    return true;
}
int main()
{
    string str = "_geeks123";
    int n = str.length();
    if (isValid(str, n))
        cout << "Valid";
    else
        cout << "Invalid";
    return 0;
}
```

Output:

Students will be able to identify comment in LEX as shown below:



```
Valid
...Program finished with exit code 0
Press ENTER to exit console.
```

POST EXPERIMENT QUESTIONS:

1. What is an identifier example?
2. Differentiate between keyword and identifier.
3. Which symbol is used in identifier?

LAB EXPERIMENT 6

OBJECTIVE:

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

BRIEF DESCRIPTION:

A deterministic finite automaton (DFA) is a finite-state machine that accepts or rejects a given string of symbols by running through a state sequence that is uniquely determined by the string in the theory of computation.

For each input symbol, the state to which the machine will move can be determined using DFA. It's called as a Deterministic Automaton as a result of this. As it contains a finite number of states, the machine is called a Deterministic Finite Machine or Deterministic Finite Automaton.

It is represented as 5 tuples $(Q, \Sigma, \delta, q_0, F)$ where:

- a) Q represents the finite states
- b) Σ represents the is a finite set of symbols, also called the alphabet
- c) δ represents the transition function where $\delta: Q \times \Sigma \rightarrow Q$
- d) q_0 represents the initial state from where any input is processed
- e) F represents the set of final state of Q .

PRE-EXPERIMENT QUESTIONS:

1. What is NFA?
2. Where is DFA used in compiler?
3. What is difference between DFA and NFA?

Explanation:

The LEX Code that accepts the string ending with 'abb' over input alphabet {a, b} and will see the implementation using LEX code and will understand the approach.

LEX provides us with an INITIAL state by default. So to make a DFA, use this as the initial state of the DFA. We define four more states: A, B, C, and DEAD, where the DEAD state would be used if encountering a wrong or invalid input. When the user inputs an invalid character, move to DEAD state, and then print "Invalid". If the input string ends at C then display the message "Accepted". Else if the input string ends at state INITIAL, A, or B then displays the message "Not Accepted".

Program Code:

```

% {
% }

%s A B C DEAD

// not accepted state after visiting A
%%
<INITIAL>a BEGIN A;
<INITIAL>b BEGIN INITIAL;
<INITIAL>[^ab\n] BEGIN DEAD;
<INITIAL>\n BEGIN INITIAL; {printf("Not Accepted\n");}

// not accepted state after visiting A and B state
<A>a BEGIN A;
<A>b BEGIN B;
<A>[^ab\n] BEGIN DEAD;
<A>\n BEGIN INITIAL; {printf("Not Accepted\n");}

// // not accepted state after visiting A and C state
<B>a BEGIN A;
<B>b BEGIN C;
<B>[^ab\n] BEGIN DEAD;
<B>\n BEGIN INITIAL; {printf("Not Accepted\n");}

// Accepted case
<C>a BEGIN A;
<C>b BEGIN INITIAL;
<C>[^ab\n] BEGIN DEAD;
<C>\n BEGIN INITIAL; {printf("Accepted\n");}

// Invalid Case
<DEAD>[^^\n] BEGIN DEAD;
<DEAD>\n BEGIN INITIAL; {printf("Invalid\n");}

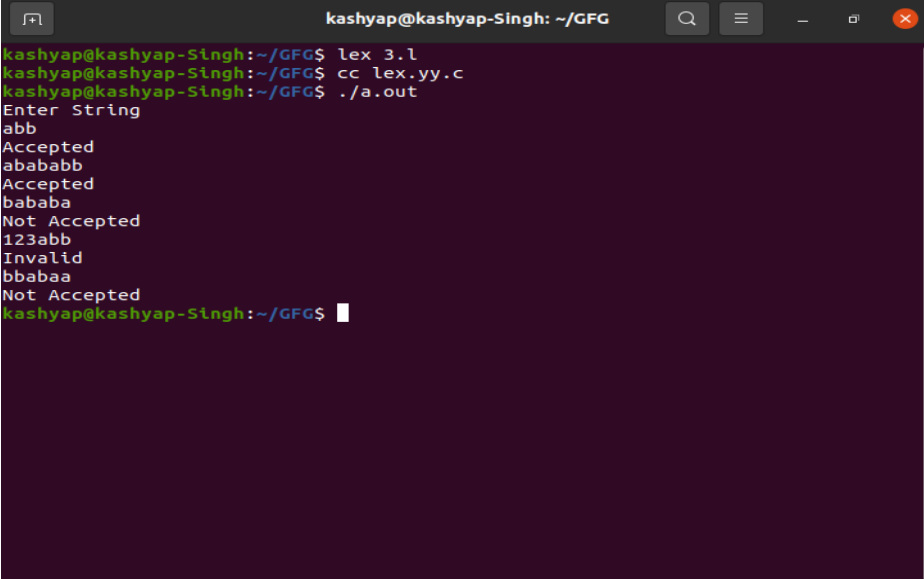
%%
// yywrap method
int yywrap()
{
return 1;
}

// main method
int main()
{
printf("Enter String\n");
// called yylex
yylex();
return 0;
}

```


Output:

Students will be able to recognize string in LEX as shown below:



```
kashyap@kashyap-Singh: ~/GFG
kashyap@kashyap-Singh:~/GFG$ lex 3.1
kashyap@kashyap-Singh:~/GFG$ cc lex.yy.c
kashyap@kashyap-Singh:~/GFG$ ./a.out
Enter String
abb
Accepted
abababb
Accepted
bababa
Not Accepted
123abb
Invalid
bbabaa
Not Accepted
kashyap@kashyap-Singh:~/GFG$
```

POST EXPERIMENT QUESTIONS:

1. Define dead state.
2. Draw a DFA as well as NFA which accept a string containing “ing” at the end of a string in a string of {a-z}, e.g., “anything” but not “anywhere”.
3. Which is more powerful DFA or NFA?

LAB EXPERIMENT 7

OBJECTIVE:

Write a Program to simulate lexical analyzer for validating operators.

BRIEF DESCRIPTION:

Lexical Analysis is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences. In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code. The role of Lexical Analyzer in compiler design is to read character streams from the source code, check for legal tokens, and pass the data to the syntax analyzer when it demands.

PRE-EXPERIMENT QUESTIONS:

1. What is 2 pass compiler?
2. What is the role of symbol table in lexical analyzer?
3. How keywords are recognized in lexical analyzer?

Explanation:

Algorithm

Step 1: Start the program

Step 2: Include necessary header files.

Step 3: The ctype header file is to load the file with predicate isdigit.

Step 4: The define directive defines the buffer size, numeric, assignment operator, relational operator.

Step 5: Initialize the necessary variables.

Step 6: To return index of new string S, token t using insert () function.

Step 7: Initialize the length of every string.

Step 8: Check the necessary condition.

Step 9: Call the initialize () function. This function loads the keywords into the symbol table.

Step 10: Check the conditions such as white spaces, digits, letters and alphanumeric.

Step 11: To return index of entry for string S, or 0 if S is not found using lookup () function.

Step 12: Check this until EOF is found.

Step 13: Otherwise initialize the token value to be none.

Step 14: In the main function if look ahead equals numeric then the value of attribute num is given by the global variable token value.

Step 15: Check the necessary conditions such as arithmetic operators, parenthesis, identifiers, assignment operators and relational operators.

Step 16: Stop the program.

Program Code:

```
% {
    #include <stdio.h>
% }
%%
">"|"<"|"<="|">="|"=="|"!=" {
    printf("Relational Operator = %s", yytext);
}
.* {
    printf("Wrong");
}
%%
int yywrap() {
    return 1;
}
int main () {
    printf("Enter the input: ");
    yylex();
}
```

Output:

Students will be able to recognize valid operators in LEX as shown below:

```
C:\Users\mayan\OneDrive\Desktop\VS Code\Compiler Design>flex operator.l
C:\Users\mayan\OneDrive\Desktop\VS Code\Compiler Design>gcc lex.yy.c
C:\Users\mayan\OneDrive\Desktop\VS Code\Compiler Design>a
Enter the input: ==
Relational Operator = ==
!+
Wrong
!=
Relational Operator = !=
```

POST EXPERIMENT QUESTIONS:

1. Define output of lexical analyzer.
2. Explain BNF.
3. What is type 0 grammar?

LAB EXPERIMENT 8

OBJECTIVE:

Write a Program for implementation of Operator Precedence Parser.

BRIEF DESCRIPTION:

A grammar that is used to define mathematical operators is called an operator grammar or operator precedence grammar. Such grammars have the restriction that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side.

An operator precedence parser is a bottom-up parser that interprets an operator grammar. This parser is only used for operator grammars. Ambiguous grammars are not allowed in any parser except operator precedence parser. There are two methods for determining what precedence relations should hold between a pair of terminals:

- a) Use the conventional associativity and precedence of operator.
- b) The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

This parser relies on the following three precedence relations: $<$, \doteq , $>$ $a < b$ This means a “yields precedence to” b. $a > b$ This means a “takes precedence over” b. $a \doteq b$ This means a “has same precedence as” b.

PRE-EXPERIMENT QUESTIONS:

1. What is operator grammar?
2. What is the role of operator precedence parser in BUP?
3. Differentiate between TDP and BUP?

Explanation:

Algorithm

1. Include the necessary header files.
2. Declare the necessary variables with the operators defined before.
3. Get the input from the user and compare the string for any operators.
4. Find the precedence of the operator in the expression from the predefined operator.
5. Set the operator with the maximum precedence accordingly and give the relational operators for them.
6. Parse the given expression with the operators and values.
7. Display the parsed expression.
8. Exit the program.

Program Code:

```

#include<stdlib.h>
#include<stdio.h>
#include<string.h>
// function f to exit from the loop
// if given condition is not true
void f()
{
    printf("Not operator grammar");
    exit(0);
}
void main()
{
    char grm[20][20], c;
    // Here using flag variable,
    // considering grammar is not operator grammar
    int i, n, j = 2, flag = 0;
    // taking number of productions from user
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%s", grm[i]);
    for (i = 0; i < n; i++) {
        c = grm[i][2];
        while (c != '\0') {
            if (grm[i][3] == '+' || grm[i][3] == '-'
                || grm[i][3] == '*' || grm[i][3] == '/')

                flag = 1;

            else {

                flag = 0;
                f();
            }

            if (c == '$') {
                flag = 0;
                f();
            }

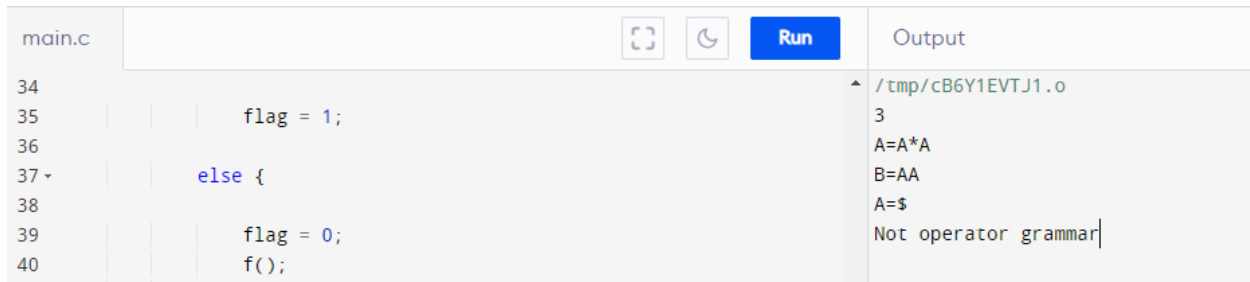
            c = grm[i][++j];
        }
    }

    if (flag == 1)
        printf("Operator grammar");
}

```

Output:

Students will be able to implement Operator grammar and its parser in LEX as shown below:



The screenshot shows a code editor with a file named 'main.c'. The code is as follows:

```
34
35     flag = 1;
36
37 -   else {
38
39     flag = 0;
40     f();
```

At the top right of the editor are icons for a refresh button, a moon icon, and a blue 'Run' button. To the right of the code editor is an 'Output' window. The output window shows the following text:

```
/tmp/cB6Y1EVTJ1.o
3
A=A*A
B=AA
A=$
Not operator grammar|
```

POST EXPERIMENT QUESTIONS:

1. How do you parse operator precedence?
2. Can we change operator precedence?
3. What is the level of precedence?

LAB EXPERIMENT 9 (i)

OBJECTIVE:

Write a Program for implementation of calculator using YACC tool.

BRIEF DESCRIPTION:

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). LEX is commonly used with the YACC parser generator. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

YACC is written in portable C. The class of specifications accepted is a very general one LALR (1) grammars with disambiguating rules.

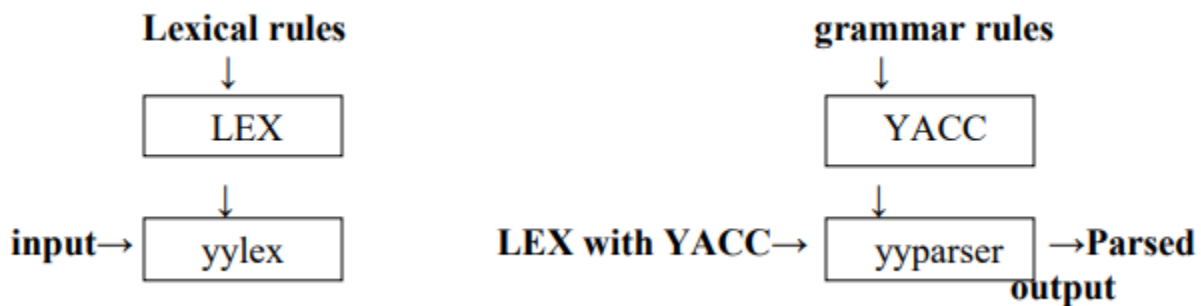
PRE-EXPERIMENT QUESTIONS:

1. What is the purpose of YACC?
2. In which phase YACC is used?
3. Which table is created by YACC?

Explanation:

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 and the yywrap method is called by the lexical analyzer whenever it inputs an EOF as the first character when trying to match a regular expression



Program Code:

```
% {
#include<stdio.h>
#include<stdlib.h>
void yyerror(char *);
#include "y.tab.h"
int yylval;
% }
40
%%
[a-z] {yylval=*yytext='&'; return VARIABLE;}
[0-9]+ {yylval=atoi(yytext); return INTEGER;}
[\t] ;
%%
int yywrap(void)
{
return 1;
}
CALC.Y
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
% {
int yylex(void);
void yyerror(char *);
int sym[26];
% }
%%
PROG:
PROG STMT '\n'
;
41
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:

Students will be able to implement applications of LEX and YACC as shown below:

```
[kss@mamsecse]$ lex calc.l
[kss@mamsecse]$ yacc -d calc.y
42
[kss@mamsecse]$ cc y.tab.c lex.yy.c -ll -ly -lm
[kss@mamsecse]$ ./a.out
Enter the Expression: ( 5 + 4 ) * 3
Answer: 27
```

POST EXPERIMENT QUESTIONS:

1. Which parsing technique is used in YACC?
2. What are the parts of a YACC file?
3. Is YACC a bottom-up parser?

LAB EXPERIMENT 9 (ii)

OBJECTIVE:

Write a Program for implementation of Recursive Descent Parser using LEX tool

BRIEF DESCRIPTION:

A recursive descent parser is a top-down parser, so called because it builds a parse tree from the top (the start symbol) down, and from left to right, using an input sentence as a target as it is scanned from left to right. (The actual tree is not constructed but is implicit in a sequence of function calls.) This type of parser was very popular for real compilers in the past, but is not as popular now. The parser is usually written entirely by hand and does not require any sophisticated tools.

This parser uses a recursive function corresponding to each grammar rule (that is, corresponding to each non-terminal symbol in the language). For simplicity one can just use the non-terminal as the name of the function. The body of each recursive function mirrors the right side of the corresponding rule. In order for this method to work, one must be able to decide which function to call based on the next input symbol.

PRE-EXPERIMENT QUESTIONS:

1. What is the purpose of a recursive descent parser?
2. What is the another name of recursive descent parser?
3. What type of parsing is recursive descent parser?

Explanation:

Lex Compiler:

- 1) Initialize identifier as [a-zA-Z][a-zA-Z0-9].
- 2) Initialize numbers as [0-9]+|([0-9]*.[0-9]+).
- 3) Assign a variable to keep track of the errors.
- 4) If the symbol is a identifier return VAR, if it is Number return NUM.
- 5) If it belongs to Relational Operators, return RELOP.
- 6) If it is a keyword then return their respective name, if it is a data type
- 7) Then return TYPE.
- 8) Ignore the white spaces and increment the errors.

YACC Compiler:

- 1) Get the input as a file name open the file in read mode.
- 2) Get the input as characters and check them.
- 3) Check for the error message and display them, Error Handling routine gets called if a statement cannot be parsed with the grammar defined.
- 4) Check the tokens for identifiers, numbers, keywords and operators.
- 5) Display the result and close the program Execution.

Program Code:

```

% {
#include<stdio.h>
35
#include<stdlib.h>
#include<string.h>
void yyerror(char *);
#include "y.tab.h"
% }
%%
[a-zA-Z][a-zA-Z0-9_]* { strcpy(yyval.str,yytext); return(ID);}
[0-9]+ { yyval.no=atoi(yytext); return(DIGIT);}
"+" {return (PLUS);}
"-" {return (MINUS);}
"/" {return (DIV);}
"*" {return (MUL);}
"(" {return (OPEN);}
")" {return (CLOSE);}
"\n" {return (0);}
[\t]
%%
int yywrap(void)
{
return 1;
36
}
YACCP.Y
%union
{
int no;
char str[10];
}
%token <no> DIGIT
%token <str> ID
%left PLUS MINUS
%left MUL DIV
%left OPEN CLOSE
%%
STMT: EXPR {printf("\n");}
;
EXPR: EXPR PLUS EXPR {printf("\n + is an ADD Operator");}
| EXPR MINUS EXPR {printf("\n - is an SUBTRACT Operator");}
| EXPR MUL EXPR {printf("\n * is an MULTIPLICATION Operator");}
| EXPR DIV EXPR {printf("\n / is an DIVISION Operator");}
| OPEN EXPR CLOSE
| DIGIT {printf("\n %d is a NUMBER",yyval.no);}
| ID {printf("\n %s is an IDENTIFIER ",yyval.str);}
37
%%
void yyerror(char *s)
{

```

```
printf("\nError: %s",s);
return;
}
int main(void)
{
yyparse();
return 0;
}
```

Output:

Students will be able to implement recursive descent parser as shown below:

```
[kss@mamsecse]$ lex lex5.l
[kss@mamsecse]$ yacc -d yaccp.y
[kss@mamsecse]$ cc y.tab.c lex.yy.c
[kss@mamsecse]$ ./a.out
( a + b ) * 9
a is an IDENTIFIER
b is an IDENTIFIER
+ is an ADD Operator
9 is a NUMBER
* is an MULTIPLICATION Operator
```

POST EXPERIMENT QUESTIONS:

1. Why do recursive-descent parsers have to backtrack?
2. What is the complexity of recursive descent parser?
3. Discuss the pros and cons of Recursive Descent Parser.

LAB EXPERIMENT 10

OBJECTIVE:

Write a Program for implementation of LL (1) Parser.

BRIEF DESCRIPTION:

The first 'L' in LL(1) stands for scanning the input from left to right, the second 'L' stands for producing a leftmost derivation, and the '1' for using one input symbol of look ahead at each step to make parsing action decisions. LL(1) grammar follows Top-down parsing method. For a class of grammars called LL(1) we can construct grammars predictive parser. That works on the concept of recursive-descent parser not requiring backtracking.

PRE-EXPERIMENT QUESTIONS:

1. Is LL(1) parser a TDP?
2. What is the purpose of 1 in LL(1)?
3. What type of functions exist in LL(1) parsing table?

Explanation:

The construction of a top-down parser is aided by FIRST and FOLLOW functions, that are associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.

Rules for First computation:

- 1) If x is terminal, then $FIRST(x) = \{x\}$
- 2) If $X \rightarrow \epsilon$ is production, then add ϵ to $FIRST(X)$
- 3) If X is a non-terminal and $X \rightarrow PQR$ then $FIRST(X) = FIRST(P)$
 - a) If $FIRST(P)$ contains ϵ , then
 - b) $FIRST(X) = (FIRST(P) - \{\epsilon\}) \cup FIRST(QR)$

Rules for Follow computation:

- 1) For Start symbol, place \$ in $FOLLOW(S)$
- 2) If $A \rightarrow \alpha B$, then $FOLLOW(B) = FOLLOW(A)$
- 3) If $A \rightarrow \alpha B \beta$, then
 - a) If ϵ not in $FIRST(\beta)$,
 $FOLLOW(B) = FIRST(\beta)$
 - else do,
 $FOLLOW(B) = (FIRST(\beta) - \{\epsilon\}) \cup FOLLOW(A)$

Program Code:

```

#include<stdio.h>
#include<string.h>
char s[30],stack[20];
int main()
{
char m[5][6][3]={{ "tb", " ", " ", "tb", " ", " " },
{ " ", "+tb", " ", " ", "n", "n"},
{ "fc", " ", " ", "fc", " ", " " },
{ " ", "n", "*fc", " ", "n", "n"},
{ "d", " ", " ", "(e)", " ", " " }};
int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};
int i,j,k,n,str1,str2;
printf("\n enter the input string:");
scanf("%s",s);
strcat(s,"$");
n=strlen(s);
stack[0]='$';
stack[1]='e';
i=1;
j=0;
printf("\n stack input\n");
printf("\n");
while((stack[i]!='$')&&(s[j]!='$'))
{
if(stack[i]=s[j])
{
i--;
j++;
}
switch(stack[i])
{
case 'e':str1=0;
break;
case 'b':str1=1;
break;
case 't':str1=2;
break;
case 'c':str1=3;
break;
case 'f':str1=4;
break; }
switch(s[j]) {
case 'd':str2=0;
break;
case '+':str2=1;
break;
case '*':str2=2;
break;
case '(':str2=3;
break;

```

```

case '):str2=4;
break;
case '$':str2=5;
break; }
if(m[str1][str2][0]=='$') {
printf("
\n error\n");
return(0); }
else if(m[str1][str2][0]='n') i--;
else if(m[str1][str2][0]='i')
stack[i]='d';
else {
for(k=size[str1][str2]
-1;k>=0;k--
)
{
stack[i]=m[str1][str2][k];
i++; }i--; }
for(k=0;k<=i;k++)
printf("%c",stack[k]);
printf(" ");
for(k=j;k<=n;k++)
printf("%c",s[k]);
printf("\n");
}
printf("\n SUCCESS");
}

```

Output:

Students will be able to implement LL(1) parser as shown below:

```

main.c
69 else if(m[str1][str2][0]='n') i--;
70 else if(m[str1][str2][0]='i')
71 stack[i]='d';

```

```

inp
enter the input string:i+i*i

stack input

+i*i$
i*i$
*i$
i$
$

SUCCESS

...Program finished with exit code 0
Press ENTER to exit console.

```

Rules for construction of parsing table:

Step 1: For each production $A \rightarrow \alpha$, of the given grammar perform Step 2 and Step 3.

Step 2: For each terminal symbol 'a' in $FIRST(\alpha)$, ADD $A \rightarrow \alpha$ in table $T[A,a]$, where 'A' determines row & 'a' determines column.

Step 3: If ϵ is present in $FIRST(\alpha)$ then find $FOLLOW(A)$, ADD $A \rightarrow \epsilon$, at all columns 'b', where 'b' is $FOLLOW(A)$. ($T[A,b]$)

Step 4: If ϵ is in $FIRST(\alpha)$ and \$ is the $FOLLOW(A)$, ADD $A \rightarrow \alpha$ to $T[A,\$]$.

POST EXPERIMENT QUESTIONS:

1. What is a lazy parser?
2. What elements makeup parsing?
3. What are the methods of parsing?

LAB EXPERIMENT 11

OBJECTIVE:

Write a Program for implementation of LALR Parser.

BRIEF DESCRIPTION:

LALR Parser is look ahead LR parser. It is the most powerful parser which can handle large classes of grammar. The size of CLR parsing table is quite large as compared to other parsing table. LALR reduces the size of this table. LALR works similar to CLR. The only difference is; it combines the similar states of CLR parsing table into one single state.

The general syntax becomes $[A \rightarrow \alpha. B, a]$ where $A \rightarrow \alpha. B$ is production and a is a terminal or right end marker $\$$ LR(1) items = LR(0) items + look ahead.

PRE-EXPERIMENT QUESTIONS:

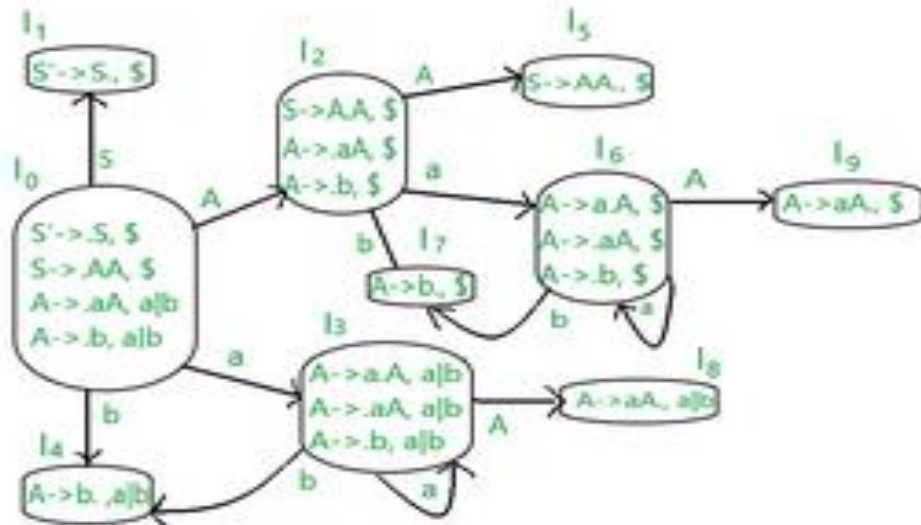
1. Is LALR(1) parser better than SLR(1) parser?
2. Which is the most powerful parser?
3. List out some applications of LALR(1) parser.

Explanation:

Construct CLR parsing table for the given context free grammar:

$S \rightarrow AA$

$A \rightarrow aA | b$



Advantages and Disadvantages of LALR Parser in Compiler Design

Advantages:

1. Efficient: LALR parsers are efficient with respect to time and space complexity.
2. Can handle large grammars: They can handle huge classes of grammars than any other parser.
3. Construction of abstract syntax tree: It is used to create an abstract syntax tree (AST) of the input source code.

Disadvantages:

1. Complexity: It requires a significant amount of effort and expertise to generate an LALR parser. Since it involves the generation of a parse table which can be large and complex.
2. Overhead: The parsing process can introduce overhead in the compiler, which can impact performance.
3. Limited look ahead: The amount of look ahead used by an LALR parser is limited, which can result in parsing errors in some cases.

Output:

Students will be able to find whether a given grammar is LALR(1) or not.

POST EXPERIMENT QUESTIONS:

1. What is the significance of an LALR handle?
2. How is an LALR handle identified?
3. What is the role of an LALR handle in the parsing process?

This lab manual has been updated by

Prof. Vimmi Malhotra (vimmi.malhotra@ggnindia.dronacharya.info)

Crosschecked By

HOD CSE

Please spare some time to provide your valuable feedback.