SWAMI KESHVANAND INSTITUTE OF TECHNOLOGY MANAGEMENT & GRAMOTHAN, JAIPUR

LAB MANUAL

Compiler Design Lab (5CS4-22)

Semester: V

Branch: Computer Science & Engineering

Session 2020-21

Faculty Name:

Deepa Modi

(Assistant Professor-II)

Department of Computer Science & Engineering

Swami Keshvanand Institute of Technology Management & Gramothan,

Ramnagaria, Jaipur-302017

VERSION 1.0

	AUTHOR/ OWNER	REVIEWED BY	APPROVED BY
NAME	Deepa Modi		
DESIGNATION	Assistant Professor-II		
SIGNATURE			
SIGNATURE			
SIGNATURE			
SIGNATURE			
SIGNATURE			

INDEX

LAB ETHICS

Do's

1. Shut down the computers before leaving the lab.
2. Keep the bags outside in the racks.
3. Enter the lab on time and leave at proper time.
4. Maintain the decorum of the lab.
5. Utilize lab hours in the corresponding experiment.
6. Get your floppies checked by lab incharge before using it in the lab.

Don’ts

1. Don't bring any external material in the lab.
2. Don't make noise in the lab
3. Don't bring the mobile in the lab.
4. If extremely necessary then keep ringers off.
5. Don't enter in server room without permission of lab incharge.
6. Don't litter in the lab.
7. Don't delete or make any modification in system files.
8. Don't carry any lab equipments outside the lab.

INSTRUCTIONS

Before Entering in the Lab

1. All the students are supposed to prepare the theory regarding the next program.
2. Students are supposed to bring the practical file and the lab copy.
3. Previous program should be written in the practical file.
4. Algorithm of the current program should be written in the lab copy.
5. Any student not following these instructions will be denied entry in the lab.

While Working in the Lab

1. Adhere to experimental schedule as instructed by the lab incharge.
2. Get the previously executed program signed by the instructor.
3. Get the output of current program checked by the instructor in the lab copy.
4. Each student should work on his assigned computer at each turn of the lab.
5. Take responsibility of valuable accessories
6. Concentrate on the assigned practical and don't play games.
7. If anyone is caught red-handed carrying any equipment of the lab, then he/she will have to face serious consequences

Marking/Assessment System

Total Marks -50

Internal Assessment Marks Distribution

Attendance	File Work	Performance	Viva	Total
5	10	10	5	30

External Assessment Marks Distribution (Depends on Examiner)

Performance	Viva	Total
10	10	20

RAJASTHAN TECHNICAL UNIVERSITY

III Year-V Semester

5CS4-22: Compiler Design Lab

List of Experiment

1. Introduction: Objective, scope and outcome of the course.
2. To identify whether given string is keyword or not.
3. Count total no. of keywords in a file. [Taking file from user]
4. Count total no of operators in a file. [Taking file from user]
5. Count total occurrence of each character in a given file. [Taking file from user]
6. Write a C program to insert, delete and display the entries in Symbol Table.
7. Write a LEX program to identify following:
    a) Valid mobile number
    b) Valid url
    c) Valid identifier
    d) Valid date (dd/mm/yyyy)
    e) Valid time (hh:mm:ss)
8. Write a lex program to count blank spaces, words, lines in a given file.
9. Write a lex program to count the no. of vowels and consonants in a C file.
10. Write a YACC program to recognize strings aaab, abbb using a^nb^n, where b>=0.
11. Write a YACC program to evaluate an arithmetic expression involving operators +,-,* and /.
12. Write a YACC program to check validity of a strings abcd, aabbcd using grammar a^nb^nc^md^m, where n , m>0
13. Write a C program to find first of any grammar.

LAB PLAN

Total Number of Experiments	:	13
Total Number of Turns	:	12

Distribution of Lab Hours

Explanation of features of language	:	20 min.
Explanation of Experiment	:	20 min.
Performance of Experiment	:	50 min.
Attendance	:	10 min.
Viva / Quiz / Queries	:	20 min.
Total	:	120 min.

Software / Hardware Required

a. C with Windows
b. Lex with Linux
c. Yacc with Linux

Lexical Elements of the C Language

Like any other High level language, C provides a collection of basic building blocks, symbolic words called lexical elements of the language. Each lexical element may be a symbol, operator, symbolic name or word, a special character, a label, expression, reserve word, etc. All these lexical elements are arranged to form the statements using the syntax rules of the C language. Following are the lexical elements of the C language.

C Character Set

Every language has its own character set. The character set of the C language consists of basic symbols of the language. A character indicates any English alphabet, digit or special symbol including arithmetic operators. The C language character set includes:

- Letter, Uppercase A ….. Z, Lower case a….z
- Digits, Decimal digits 0….9.
- Special Characters, such as comma, period. semicolon; colon: question mark?, apostrophe‘ quotation mark “ Exclamation mark ! vertical bar | slash / backslash \ tilde ~ underscore _ dollar $ percent % hash # ampersand & caret ^ asterisk * minus – plus + <, >, (,), [,], {, }
- White spaces such as blank space, horizontal tab, carriage return, new line and form feed.

C Tokens

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in C program, the smallest individual units are known as C tokens. C has following tokens

- Keywords or Reserve words such as float, int, etc
- Constants such 1, 15,5 etc
- Identifiers such name, amount etc
- Operators such as +, -, * etc
- Separators such as :, ;, [,] etc and special characters
- Strings

Keywords

Key words or Reserve words of the C language are the words whose meaning is already defined and explained to the C language compiler. Therefore Reserve words can not be used as identifiers or variable names. They should only be used to carry the pre-defined meaning. For example int is a reserve word. It indicates the data type of the variable as integer. Therefore it is reserved to carry the specific meaning. Any attempt to use it other than the intended purpose will generate a compile time error. C language has 32 keywords. Following are some of them

auto break case char const continue default

do double else enum extern float for
goto if int long register return
short
signed sizeof static struct switch typedef
union
unsigned void volatile while

Constants

A constant can be defined as a value or a quantity which does not change during the execution of a program. Meaning and value of the constant remains unchanged throughout the execution of the program. These are also called as literals. C supports following types of constant.

1. Integer Constants

An integer constant refers to a sequence of digits. There three types of integer constants, namely decimal, octal and hexadecimal. Decimal integer constant consists of set of digits from 0 to 9 preceded by an optional + or – sign.

Ex: 123, -321, 0, 4567, + 78

Embedded spaces, commas and non-digit characters are not permitted between digits. An octal integer constant consists of any combination of digits from 0 to 7 with a leading 0(zero). Ex : 037, 0435, 0567. A sequence of digits preceded by 0x or 0X is considered as hexadecimal digit. They may also include alphabets A to F or a to f representing numbers from 10 to 15. The largest integer value that can be stored is machine dependant; It is 32767 for 16 bit computers. It is also possible to store larger integer constants by appending qualifiers such U, L and UL to the constants.

2. Floating point Constants or Real Constants

The quantities that are represented by numbers with fractional part are called floating point numbers. Ex: 0.567, -0.76, 56.78, +247.60. These numbers are shown in decimal notation , having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point or digits after the decimal point. Ex: 215., .95, -.76 or +.5 A real number or a floating point number can also expressed as in exponential notation. Ex: 2.15E2. The general form of exponential notation is mantissa **e** exponent or mantissa **E** exponent The mantissa is either a real number or an integer. The exponent is an integer with an optional plus or minus sign. Embedded white is not allowed in this notation.

3. Single Character constants

A single character constant contains a any valid character enclosed within a pair of single quote marks. Ex: ‘5’, ‘A’, ‘;’ ‘ ‘. The character constants have integer values associated with them known as ASCII values. For ex: A is having the ASCII value of 65.

4. String constants

A string constant is a sequence of characters enclosed in double quotes. The characters may be alphabets, numbers special characters and blank space. Ex: “Hello”, “2002”, “Wel Come”, “5+3”

5. Backslash character constants or Escape sequence characters.

C supports some special backslash character constants that are used in output functions. For ex: '\n' stands for new line characters. Each one of them represents a single character even though it consists of two characters.

\a	Audible alert or bell	\b	back space
\f	form feed	\n	new line
\r	carriage return	\t	horizontal tab
\v	vertical tab	\'	single quote
\"	double quote	\?	Question mark
\\	backslash	\0	null

Identifiers

An identifier is a sequence of letters, digits and an underscore. Identifiers are used to identify or name program elements such as variables, function names, etc. Identifiers give unique names to various elements of the program. Some identifiers are reserved as special to the C language. They are called keywords.

Variables

A variable is a data name that may be used to store data value. A value or a quantity which may vary during the program execution can be called as a variable. Each variable has a specific memory location in memory unit, where numerical values or characters can be stored. A variable is represented by a symbolic name. Thus variable name refers to the location of the memory in which a particular data can be stored. Variables names are also called as identifiers since they identify the varying quantities. For Ex : sum = a+b. In this equation sum, a and b are the identifiers or variable names representing the numbers stored in the memory locations.

<u>Rules to be followed for constructing the Variable names(identifiers)</u>

- They must begin with a letter and underscore is considered as a letter.
- It must consist of single letter or sequence of letters, digits or underscore character.
- Uppercase and lowercase are significant. For ex: Sum, SUM and sum are three distinct variables.
- Keywords are not allowed in variable names.
- Special characters except the underscore are not allowed.
- White space is also not allowed.
- The variable name must be 8 characters long. But some recent compilers like ANSI C supports 32 characters for the variable names and first 8 characters are significant in most compilers.

Data Types

Data refers to any information which is to be stored in a computer. For example, marks of a student, salary of a person, name of a person etc. These data may be of different types. Computer allocates memory to a variable depending on the data that is to be stored in the variable. So it becomes necessary to define the type of the data which is to be stored in a variable while declaring a variable. The different data types supported by C are as follows:

int Data Type

Integer refers to a whole number with a range of values supported by a particular machine. Generally integers occupy one word of storage i.e 16 bits or 2 bytes. So its value can range from -32768 to +32767.

Declaration: int variable name;

Ex: int qty;

The above declaration will allocate a memory location which can store only integers, both positive and negative. If we try to store a fractional value in this location, the fractional data will be lost.

float Data Type

A floating point number consists of sequence of one or more digits of decimal number system along with embedded decimal point and fractional part if any. Computer allocates 32 bits, i.e. 4 bytes of memory for storing float type of variables. These numbers are stored with 6 digits of precision for fractional part.

Declaration: float variableName; Ex : float amount;

double Data Type

It is similar to the float type. It is used whenever the accuracy required to represent the number is more. In others words variables declared of type double can store floating point numbers with number of significant digits is roughly twice or double than that of float type. It uses 64 bits i.e. 8 bytes of memory giving a precision of 14 decimal digits.

Declaration : double variableName;

char Data Type

A single character can be defined as char data type. These are stored usually as 8 bits i.e 1 byte of memory.

Declaration : char variableName ; Ex: char pass;

String refers to a series of characters. Strings are declared as array of char types. Ex: char name[20]; will reserve a memory location to store upto 20 characters.

Further, applying qualifiers to the above primary data types yield additional data types. A qualifier alters the characteristics of the data type, such as its sign or size. There are two types of qualifiers namely, sign qualifiers and size qualifiers. signed and unsigned are the sign qualifiers short and long are the size qualifiers.

Size and range of data types on a 16 bit Machine

Type	Size (bits)	Range
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int	16	-32768 to 32767
unsigned int	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4e-38 to 3.4e+38
double	64	1.7e-308 to 1.7e+308
long double	80	3.4 e-4932 to 1.1e+4932

Declaring a variable as constant

We may want the value of the certain variable to remain constant during the execution of the program. We can achieve this by declaring the variable with const qualifier at the time of initialization.

Ex: const int tax_rate = 0.30;

The above statement tells the compiler that value of variable must not be modified during the execution of the program. Any attempt change the value will generate a compile time error.

Declaring a variable as volatile

Declaring the variable volatile qualifier tells the compiler explicitly that the variable's value may be changed at any time by some external source and the compiler has to check the value of the variable each time it is encountered.

Ex; volatile int date;

Defining Symbolic Constants

We often use certain unique constants in a program. These constants may appear repeatedly in number of places in a program. Such constants can be defined and its value can be substituted during the preprocessing stage itself.

Operators in C

An operator is a symbol which acts on operands to produce certain result as output. For example in the expression a+b; + is an operator, a and b are operands. The operators are fundamental to any mathematical computations.

Operators can be classified as follows:

- Based on the number of operands the operator acts upon:
    - Unary operators: acts on a single operand. For example: unary minus(-5, -20, etc), address of operator (&a)
    - Binary operators: acts on two operands. Ex: +, -, %, /, *, etc
    - Ternary operator: acts on three operands. The symbol ?: is called ternary operator in C language. Usage: big= a>b?a:b; i.e if a>b, then big=a else big=b.
- Based on the functions
    - Arithmetic operators
    - Relational operators
    - Logical Operators
    - Increment and Decrement operators
    - Assignment operators
    - Bitwise operators
    - Conditional Operators
    - Special operators

Precedence and Associativity of operators:

Each operator in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. The operator at the higher level of precedence is evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left depending on the level. This is known as the associativity property of an operator.

Operator	Description	Level	Associativity
( ) [ ]	Parenthesis Array index	1	L – R
+ - ++ -- ! ~ & sizeof(type)	Unary plus Unary minus Increment Decrement Logical negation One’s Complement Address of type cast conversion	2	R – L
* / %	Multiplication Division Modulus	3	L- R
+ -	Addition Subtraction	4	L – R
<< >>	Left Shift Right Shift	5	L – R
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	6	L – R
= = ! =	is equal to Not equal to	7	L – R
&	Bitwise AND	8	L – R

^	Bitwise XOR	9	L – R
|	Bitwise OR	10	L – R
&&	Logical AND	11	L – R
| |	Logical OR	12	L – R
? :	Conditional Operator	13	R – L
=, +=, -=, *=, /=, %=	Assignment operator Short hand assignement	14	R – L
,	Comma operator	15	R – L

Preprocessor directives:

There are different preprocessor directives. The table below shows the preprocessor directives.

Directive	**Function**
#define	defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included.
#ifdef	Tests for a macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether a macro is not defined
#if	Tests a compile-time condition
#else	Specifies alternatives when #if tests fails

Header files:

C language offers simpler way to simplify the use of library functions to the greatest extent possible. This is done by placing the required library function declarations in special source files, called header files. Most C compilers include several header files, each of which contains declarations that are functionally related. stdio.h is a header file containing declarations for input/ouput routines; math.h contains declarations for certain mathematical functions and so on. The header files also contain other information related to the use of the library functions, such as symbolic constant definitions.

The required header files must be merged with the source program during the compilation process. This is accomplished by placing one or more #include statements at the beginning of the source program. The other header files are:

<ctype.h>	character testing and conversion functions
<stdlib.h>	utility functions such as string conversion routines , memory allocation routines, random number generator etc
<string.h>	String manipulations functions
<time.h>	Time manipulation functions

Input and Output Functions

The C language consists of input-output statements to read the data to be processed as well as output the computed results. C language provides a set of library functions or built in functions, in order to carry out input and output operations. These library functions are available in a header file called stdio.h. So for using these library functions the following preprocessor directive is essential.

The input and output functions in C language can be broadly categorized into two types:

- Unformatted Input Output functions : which provides the facility to read or output data as a characters or sequence of characters. Ex: getch(), getche(), getchar(), gets(), putch(), purchar() and puts().
- Formatted I/O functions : which allow the use format specifiers to specify the type of data to be read or printed. Ex: scanf() and printf() functions.

Introduction to Lex Programming

The unix utility lex parses a file of characters. It uses regular expression matching; typically it is used to 'tokenize' the contents of the file. In that context, it is often used together with the yacc utility. However, there are many other applications possible.

Structure of a lex file

A lex file looks like:

```
        ...definitions...
%%
        ...rules...
%%
        ...code...
```

Here is a simple example:

```
%{
int charcount=0,linecount=0;
%}

%%
. charcount++;
\n {linecount++; charcount++;}
%%

int main() {
yylex();
printf("There were %d characters in %d lines\n", charcount,linecount);
return 0; }
```

In the example you just saw, all three sections are present:

- **definitions** All code between %{ and %} is copied to the beginning of the resulting C file. **rules** A number of combinations of pattern and action: if the action is more than a single
- command it needs to be in braces.
- **code** This can be very elaborate, but the main ingredient is the call to yylex, the lexical analyser. If the code segment is left out, a default main is used which only calls yylex.

Introduction to YACC Programming

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.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma ``,'' is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

These are some points about YACC:

Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

EXPERIMENT-1

AIM:

Introduction: Objective, scope and outcome of the course.

OBJECTIVE:

The laboratory course is intended to make experiments on the basic techniques of compiler construction and tools that can be used to perform syntax-directed translation of a high-level programming language into an executable code. Students will design and implement language processors in C by using tools to automate parts of the implementation process. This will provide deeper insights into the more advanced semantics aspects of programming languages, code generation, machine independent optimizations, dynamic memory allocation, and object orientation.

SCOPE:

The scope of this course is to explore the principle, algorithm and data structure involved in the design and construction of compiler.

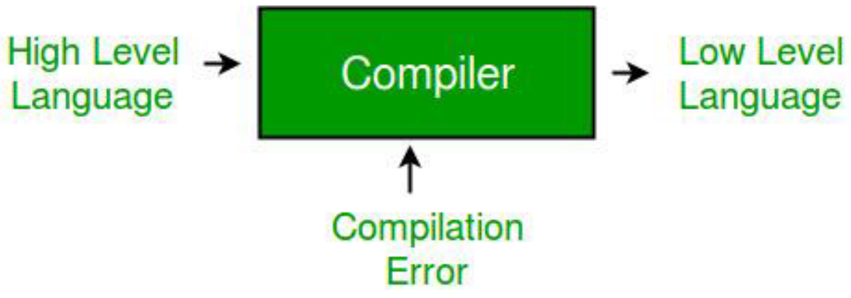
OUTCOMES:

Upon the completion of Compiler Design practical course, the student will be able to:

1. Understand the working of lex and yacc compiler for debugging of programs.
2. Understand and define the role of lexical analyzer, use of regular expression and transition diagrams.
3. Understand and use Context free grammar, and parse tree construction.
4. Learn & use the new tools and technologies used for designing a compiler.
5. Develop program for solving parser problems.
6. Learn how to write programs that execute faster.

Introduction of Compiler Design

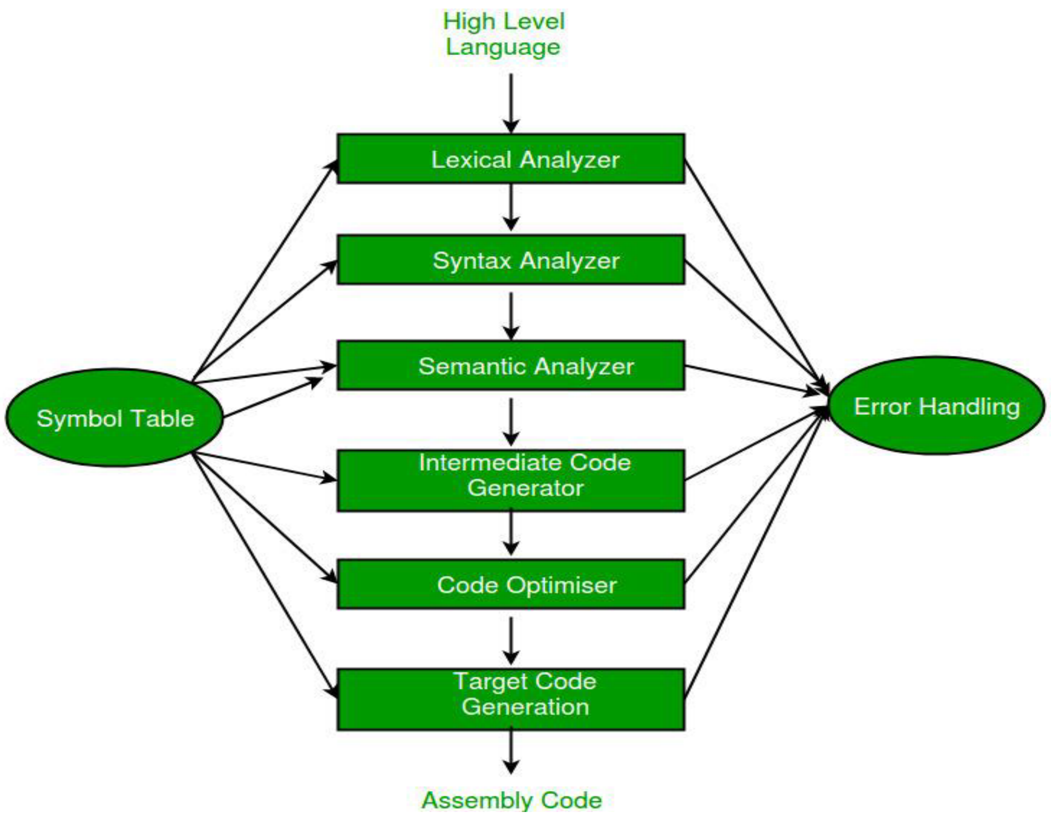
Compiler is a software which converts a program written in high level language (Source Language) to low level language (Object/Target/Machine Language).



- **Cross Compiler** that runs on a machine ‘A’ and produces a code for another machine ‘B’. It is capable of creating code for a platform other than the one on which the compiler is running.

Phases of a Compiler –

There are two major phases of compilation, which in turn have many parts. Each of them take input from the output of the previous level and work in a coordinated way.



Analysis Phase – An semantic tree representation is created from the give source code:

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer

Lexical analyzer divides the program into "tokens", Syntax analyzer recognizes "sentences" in the program using syntax of language and Semantic analyzer checks static semantics of each construct.

Synthesis Phase – It has three parts :

4. Intermediate Code Generator
5. Code Optimizer
6. Code Generator

Intermediate Code Generator generates "abstract" code. Code Optimizer optimizes the abstract code, and final Code Generator translates abstract intermediate code into specific machine instructions.

<u>**EXPERIMENT-2**</u>

AIM:

Program to find whether given string is keyword or not

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

void main()
{
        char a[5][10]={"printf","scanf","if","else","break"}; char
        str[10];
        int i,flag;

        clrscr();

        puts("Enter the string :: ");
        gets(str);

        for(i=0;i<strlen(str);i++)
        {
                if(strcmp(str,a[i])==0)
                {
                        flag=1;
                        break;
                }
                else
                        flag=0;
        }
        if(flag==1)
                puts("Keyword");
        else
                puts("String");

        getch();
}
```

EXPERIMENT-3

AIM:

Count total no. of keywords in a file. [Taking file from user]

PROGRAM:

```
#include<stdio.h>

#include<stdlib.h>
#include<string.h>
#include<ctype.h>
static int count=0;
int isKeyword(char buffer[]){

    char keywords[32][10] =
    {"auto","break","case","char","const","continue","default","do","double","else","enum","extern","fl
    oat","for","goto","if","int","long","register","return","short","signed","sizeof","static","struct","switc
    h","typedef","union","unsigned","void","volatile","while"};
    int i, flag = 0;

    for(i = 0; i < 32; ++i){
        if(strcmp(keywords[i], buffer) == 0){
                flag = 1;
count++;
                break;
        }
    }

    return flag;
}

int main(){
    char ch, buffer[15]   ;
    FILE *fp;
    int i,j=0;

    fp = fopen("KESHAV3.C","r");

    if(fp == NULL){
        printf("error while opening the file\n");
        exit(0);
    }

    while((ch = fgetc(fp)) != EOF){
        if(isalnum(ch)){
                buffer[j++] = ch;
        }
        else if((ch == ' ' || ch == '\n') && (j != 0)){
                        buffer[j] = '\0';
                        j = 0;

                        if(isKeyword(buffer) == 1)
                                printf("%s is keyword\n", buffer);

        }

    }
    printf("no of keywords= %d", count);
    fclose(fp);

    return 0;
}
```

EXPERIMENT-4

AIM:

Count total no of operators in a file. [Taking file from user] #include<stdio.h>

PROGRAM:

```
#include<stdlib.h>

#include<string.h>
#include<ctype.h>
static int count=0;
int main(){
    char ch, buffer[15], operators[] = "+-*/%=";
    FILE *fp;
    int i;
    clrscr();

    fp = fopen("KESHAV3.C","r");

    if(fp == NULL){
        printf("error while opening the file\n");
        exit(0);
    }

    while((ch = fgetc(fp)) != EOF){
        for(i = 0; i < 6; ++i){
                if(ch == operators[i]) {
                        printf("%c is operator\n", ch);
                        count++;
                        }
        }



    }
    printf("no of operators= %d", count);
    fclose(fp);

    return 0;
}
```

EXPERIMENT-5

AIM:

Count total occurrence of each character in a given file. [Taking file from user]

PROGRAM:

```
#include <stdio.h>

#include <string.h>

#include<conio.h>

int main ()
{

    FILE * fp;
    char string[100];
    int c = 0, count[26] = { 0 }, x;


    fp = fopen ("deepa.txt", "r");
    clrscr();

    while (fscanf (fp, "%s", string) != EOF)

    {       c=0;
            while (string[c] != '\0')
            {

 /** Considering characters from 'a' to 'z' only and ignoring others. */

                    if (string[c] >= 'a' && string[c] <= 'z')
                    {

                            x = string[c] - 'a';

                            count[x]++;

                    }
            c++;

            }
    }


    for (c = 0; c <26 ; c++)
    printf ("%c occurs %d times in the string.\n", c + 'a', count[c]);
    return 0;

}
```

EXPERIMENT-6

AIM:

Write a C program to insert, delete and display the entries in Symbol Table.

PROGRAM:

```
//Implementation of symbol table

#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
void main()
{
 int i=0,j=0,x=0,n;
 void *p,*add[5];
 char ch,srch,b[15],d[15],c;
 printf("Expression terminated by $:");
 while((c=getchar())!='$')
 {
  b[i]=c;
  i++;
 }
 n=i-1;
 printf("Given Expression:");
 i=0;
 while(i<=n)
 {
  printf("%c",b[i]);
  i++;
 }
 printf("\n Symbol Table\n");
 printf("Symbol \t addr \t type");
 while(j<=n)
 {
  c=b[j];
  if(isalpha(toascii(c)))
  {
   p=malloc(c);
   add[x]=p;
   d[x]=c;
   printf("\n%c \t %d \t identifier\n",c,p);
   x++;
   j++;
}
else
{
 ch=c;
 if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
 {
  p=malloc(ch);
  add[x]=p;
  d[x]=ch;
  printf("\n %c \t %d \t operator\n",ch,p);
  x++;
  j++;
 }}}}
```

EXPERIMENT-7

AIM:

Write a LEX program to identify following:

1. Valid mobile number
2. Valid url
3. Valid identifier
4. Valid date (dd/mm/yyyy)
5. Valid time (hh:mm:ss)

PROGRAM:

1.Valid mobile number

```
%{
   /* Definition section */
%}

/* Rule Section */
%%

[1-9][0-9]{9} {printf("\nMobile Number Valid\n");}

.+ {printf("\nMobile Number Invalid\n");}

%%

// driver code
int main()
{
   printf("\nEnter Mobile Number : ");
   yylex();
   printf("\n");
   return 0;
}
int yywrap()
{
}
```

2. Valid url

```
%%

((http)|(ftp))s?:\/\/[a-zA-Z0-9]{2,}(\.[a-z]{2,})+(\/[a-zA-Z0-9+=?]*)* {printf("\nURL Valid\n");}

.+ {printf("\nURL Invalid\n");}

%%



void main() {



        printf("\nEnter URL : ");

        yylex();

        printf("\n");



}
int yywrap()
{
}
```

3. Valid identifier

```
%%

^[a - z A - Z _][a - z A - Z 0 - 9 _] * printf("Valid Identifier");

// regex for invalid identifiers
^[^a - z A - Z _] printf("Invalid Identifier");
.;
%%



void main() {



        printf("\nEnter Identifier: ");

        yylex();

        printf("\n");



}
int yywrap()
{
}
```

4. Valid date (dd/mm/yyyy)

```
%{
    #include<stdio.h>
    int i=0,yr=0,valid=0;
    %}
    %%
    ([0-2][0-9]|[3][0-1])\/((0(1|3|5|7|8))|(10|12))\/([1-2][0-9][0-9][-0-9]) {valid=1;}

    ([0-2][0-9]|30)\/((0(4|6|9))|11)\/([1-2][0-9][0-9][0-9]) {valid=1;}

    ([0-1][0-9]|2[0-8])\/02\/([1-2][0-9][0-9][0-9]) {valid=1;}

    29\/02\/([1-2][0-9][0-9][0-9]) { while(yytext[i]!='/')i++;
    i++;while(yytext[i]!='/')i++;i++;while(i<yyleng)yr=(10*yr)+(yytext[i++]-'0');
    if(yr%4==0||(yr%100==0&&yr%400!=0))valid=1;}

    %%
    void main()
    {
    yyin=fopen("new","r");
    yylex();
    if(valid==1) printf("It is a valid date\n");
    else printf("It is not a valid date\n");
    }
    int yywrap()
    {
    return 1;
    }
```

5.Valid time(hh:mm:ss)

```
%{
#include<stdio.h>
int i=0,yr=0,valid=0;
%}
%%
([0-2][0-9]:[0-6][0-9]\:[0-6][0-9]) {printf("%s It is a valid time\n",yytext);}


%%
void main()
{
yyin=fopen("new","r");
yylex();
}
int yywrap()
{
return 1;
}
```

EXPERIMENT-8

AIM:

Write a lex program to count blank spaces,words,lines in a given file.

PROGRAM:

```
%{
    #include<stdio.h>
    int lines=0, words=0,s_letters=0,c_letters=0, num=0, spl_char=0,total=0;
    %}
    %%
    \n { lines++; words++;}
    [\t ' '] words++;
    [A-Z] c_letters++;
    [a-z] s_letters++;
    [0-9] num++;
    . spl_char++;
    %%
    void main(void)
    {
    FILE *fp;
    char f[50];
    printf("enterfile name \n");
    scanf("%s",f);
    yyin= fopen(f,"r");
    yylex();
    total=s_letters+c_letters+num+spl_char;
    printf(" This File contains ...");
    printf("\n\t%d lines", lines);
    printf("\n\t%d words",words);
    printf("\n\t%d small letters", s_letters);
    printf("\n\t%d capital letters",c_letters);
    printf("\n\t%d digits", num);
    printf("\n\t%d special characters",spl_char);
    printf("\n\tIn total %d characters.\n",total);
    }
     int yywrap()
    {
    return(1);
    }
```

EXPERIMENT-9

AIM:

Write a lex program to count the no. of vowels and consonants in a C file.

PROGRAM:

```
%{

#include<stdio.h>
int vcount=0,ccount=0;
%}
%%
[a|i|e|o|u|E|A|I|O|U] {vcount++;}
[a-z A-Z (^a|i|e|o|u|E|A|I|O|U) ] {ccount++;}
%%
int main()
{
FILE *fp;
char f[50];
printf("enterfile name \n");
scanf("%s",f);
yyin= fopen(f,"r");
yylex();
printf("No. of Vowels :%d\n",vcount);
printf("No. of Consonants :%d\n",ccount);
        return 0;
}
int yywrap()
{
}
```

EXPERIMENT-10

AIM:

Write a YACC program to recognize strings aaab,abbb using a^nb^n, where b>=0.

PROGRAM:

Gm.l

```
%{
#include "y.tab.h"
%}
%%
"a"|"A" {return A;}
"b"|"B" {return B;}
[ \t] {;}
\n {return 0;}
. {return yytext[0];}
%%
int yywrap()
 {
  return 1;
 }
```

Gm.y

```
%{
#include<stdio.h>
%}
%token A B
%%
stmt: S
    ;
S: A S B
  |
 ;
%%
void main()
{
printf("enter \n");
yyparse();
printf("valid");
exit(0);
}
void yyerror()
{
printf("invalid ");
exit(0);
}
```

EXPERIMENT-11

AIM:

Write a YACC program to evaluate an arithmetic expression involving operators +,-,* and /.

PROGRAM:

Expr.l

```
%{
#include "y.tab.h"
 extern int yylval;
%}
%%
[0-9]+ {yylval=atoi(yytext);
           return number;}
[\t] {;}
[\n] {return 0;}
. {return yytext[0];}
%%
int yywrap()
 {
  return 1;
 }
```

Expr.y

```
%{
#include<stdio.h>
int res=0;
%}
%token number
%left '+' '-'
%left '*' '/'
%%
stmt:expr {res=$$;}
;
expr:expr '+' expr {$$=$1+$3;}
    |expr '-' expr {$$=$1-$3;}
    |expr '*' expr {$$=$1*$3;}
    |expr '/' expr {if($3==0)
             exit(0);
             else $$=$1/$3;}
    |number
    ;
%%
void main()
{
printf(" enter expr\n");
yyparse();
printf("valid=%d",res);
exit(0);
}

void yyerror()
{
printf("invalid\n");
exit(0);
}
```

<u>EXPERIMENT-12</u>

AIM:

Write a YACC program to check validity of a strings abcd,aabbcd using grammar a^nb^nc^md^m, where n , m>0

PROGRAM:

Grammer.y

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

%}
%token A B C D NEWLINE
%%
stmt: S NEWLINE {  printf("valid\n");
          return 1;
        }
    ;
S: X Y
  ;

X: A X B
   |
   ;
Y: C Y D
   |
   ;
%%
extern FILE *yyin;
void main()
{
printf("enter \n");
do
{
yyparse();
}
while(!feof(yyin));

}
int yyerror(char* str)
{
```

```
printf("invalid ");
return 1;
}
```

Grammer.l

```
%{
#include"y.tab.h"
%}
%%
a |
A {return A;}
c |
C {return C;}
b |
B {return B;}
d |
D {return D;}
[ \t] {;}
"\n" {return NEWLINE;}
. {return yytext[0];}
%%
int yywrap()
 {
  return 1;
 }
```

EXPERIMENT-13

AIM:

Write a C program to find first of any grammar.

PROGRAM:

```
#include<stdio.h>

#include<ctype.h>
void FIRST(char );
int count,n=0;
char prodn[10][10], first[10];

void main()
{
int i,choice;
char c,ch;
printf("How many productions ? :");
scanf("%d",&count);
printf("Enter %d productions epsilon= $ :\n\n",count);
for(i=0;i<count;i++)
scanf("%s%c",prodn[i],&ch);
do
{
n=0;
printf("Element :");
scanf("%c",&c);
FIRST(c);
printf("\n FIRST(%c)= { ",c);
for(i=0;i<n;i++)
printf("%c ",first[i]);
printf("}\n");

printf("press 1 to continue : ");
scanf("%d%c",&choice,&ch);
}
while(choice==1);
}

void FIRST(char c)
{
int j;
if(!(isupper(c)))first[n++]=c;
for(j=0;j<count;j++)
{
if(prodn[j][0]==c)
{
if(prodn[j][2]=='$') first[n++]='$';
else if(islower(prodn[j][2]))first[n++]=prodn[j][2];
else FIRST(prodn[j][2]);
}
}
}
```

Multiple Choice Questions

Q1. What is a compiler?
A. A compiler does a conversion line by line as the program is run
B. A compiler converts the whole of a higher level program code into machine code in one step
C. A compiler is a general purpose language providing very efficient execution
D. All of the Above
Ans.: B

Q2. What are the stages in the compilation process?
A. Feasibility study, system design, and testing
B. Implementation and documentation
C. Analysis Phase, Synthesis Phase
D. None of These
Ans.: C

Q3. What is the definition of an interpreter?
A. An interpreter does the conversion line by line as the program is run
B. An interpreter is a representation of the system being designed
C. An interpreter is a general purpose language providing very efficient execution
D. All of the Above
Ans.: A

Q4. Symbol table can be used for
A. Checking type compatibility
B. Storage allocation
C. Suppressing duplication of error massages
D. All of the Above
Ans.: D

Q5. A basic block can be analyzed by
A. A DAG
B. A graph which may involve the cycles
C. Flow Graph
D. None of These
Ans.: A

Q6. Assembly language
A. is usually the primary user interface
B. requires fixed format commands
C. is a mnemonic form of machine language
D. is a quite different from the SCL interpreter
Ans.: C

Q7. Every symbolic references to a memory operand has to be assembled as
A. (offset, index base)
B. (segment base, offset)
C. (index base, offset)
D. offset
Ans.: B

Q8. Type checking is normally done during
A. lexical analysis
B. syntax analysis
C. syntax directed translation
D. code optimization
Ans.: C

Q9. Which translator program converts assembly language program to object program
A. Assembler
B. Compiler
C. Microprocessor
D. Linker
Ans.: A

Q10. A compiler for a high level language that runs on one machine and produces code for a different machine is called
A. Optimizing compiler
B. One pass compiler
C. Cross compiler
D. Multipass Compiler
Ans.: C

Q11. An intermediate code form is
A. Postfix notation
B. Syntax trees
C. Three address code
D. All of these
Ans.: D

Q12. Which one of the following is not a syntax error
A. Semantic
B. Lexical
C. Arithmetic
D. Logical
Ans.: A

Q13. Semantic errors can be detected
A. at compile time only
B. at run time only
C. both at compile and run time
D. none of these
Ans.: C

Q14. The action of passing the source program into the proper syntactic classes is known as
A. syntax analysis
B. lexical analysis
C. interpretation analysis
D. general syntax analysis
Ans.: B

Q15. Undeclared name is …………… error
A. syntax
B. lexical
C. semantic
D. not an error
Ans. C

Q16. Intermediate code generator is used between
A. symbol table and code generator
B. symbol table and code optimizer
C. code optimizer and code generator
D. semantic analyzer and code optimizer
Ans.: D

Q17. Grouping of characters into tokens is done by the
A. scanner
B. parser
C. code generator
D. code optimizer
Ans.: A

Q18. Lexical analysis phase uses
A. regular grammar
B. context free grammar
C. context sensitive grammar
D. none of the above
Ans.: A

Q19. 'Divide by 0' is a
A. lexical error
B. syntactic error
C. semantic error
D. internal error
Ans. C

Q20. In which of the following has attribute values at each node
A. Associated parse trees
B. Postfix parse tree
C. Annotated parse tree
D. Prefix parse tree
Ans.: C

Q21. CFG can be recognized by a
A. Push-down automata
B. 2-way linear bounded automata
C. Both (a) and (b)
D. None of these
Ans. c

Q22. The ambiguous grammar can have
a. Only one parse tree
b. More than one parse tree
c. Parse trees with l-values
d. None of the above
Ans. B

Q23. Which of the following suffices to convert an arbitrary CFG to an LL(1) grammar
a. Removing left recursion alone
b. Factoring the grammar alone
c. Removing left recursion and factoring the grammar
d. None of the above
Ans. C

Q24. The 'k' in LR(k) cannot be
a. 0
b. 1
c. 2
d. None of these

Ans. d

Q25. Non backtracking form of the top-down parser are called

a. Recursive-descent parsers
b. Predictive parsers
c. Shift reduce parsers
d. None of these

Ans. b

Q26. Parsing table in the LR parsing contains

a. action, goto
b. state, action
c. input, action
d. state, goto

Ans. a

Q27. The condensed form of the parse tree is called

a. L-attributed
b. Inherited attributed
c. DAG
d. Syntax tree

Ans. d

Q28. The post fix form of A-B/(C*D$E) is

a. ABCDE$*/-
b. AB/C*DE$
c. ABCDE$-/*
d. ABCDE/-*$

Ans. a

Q29. Shift-reduce parsers are

a. top-down parsers
b. bottom-up parsers
c. both (a) and (b)
d. none of these

Ans. b

Q30. In some programming languages, an identifier is permitted to be a letter followed by any number of letters or digits. If L and D denote the sets of letter and digits respectively, which expression defines an identifier?

a. $(LUD)^+$
b. $L(LUD)^*$
c. $(LD)^*$
d. $L(LD)^*$

Note: Here U stands for Union

Ans. b

Q31. Backtracking is possible in

a. LR parsing
b. Predictive parsing
c. Recursive descent parsing
d. None of the above

Ans. c

Q32. Consider the following grammar

a. expr op expr
b. expr->id
c. op-> + | *

Which of the following is true?

a. op and expr are start symbols
b. op and id are terminals
c. expr is start symbol and op is terminal
d. none of these

Ans. c

Q33. To compute FOLLOW(A) for any grammar symbol A

a. We must compute FIRST of some grammar symbols
b. No need of computing FIRST of some symbol
c. May compute FIRST of some symbol
d. None of these

Ans. a

34. Which language is generated by the given grammar S-> 0S1 | 01

a. 0011
b. 00^*11
c. 001^*1
d. 00^*1^*1

Ans. d

Q35. The space consuming but easy parsing is

a. LALR
b. SLR
c. LR
d. Predictive parser

Ans. a

Q36. A top down parser generates

a. left-most derivation
b. right-most derivation
c. right-most derivation in reverse
d. left-most derivation in reverse

Ans. a

Q37. Synthesized attribute can easily be simulated by an

a. LL grammar
b. Ambiguous grammar
c. LR grammar
d. None of the above

Ans. c

Q38. Choose the false statement

a. LL(k) grammar has to be a CFG
b. LL(k) grammar has to be unambiguous
c. There are LL(k) grammars that are not Context Free
d. LL(k) grammars cannot have left recursive non-terminals

Ans. c

Q39. If a grammar is unambiguous then it is surely be

a. regular
b. LL(1)
c. Both (a) and (b)
d. Cannot say

Ans. d

Q40. Predictive parsing is a special case of

a. top down parsing
b. bottom up parsing
c. recursive descent parsing
d. none of the above

Ans. c

Q41. The prefix form of (A+B)*(C-D) is

a. +-AB*C-D
b. *+-ABCD
c. *+AB-CD
d. *AB+CD

Ans. c

Q42. Which of the following is true for the flow of control among procedures during execution of program?

a. Control flows randomly
b. Control flows line by line without jumping
c. Control flows sequentially
d. None of these

Ans. c

Q43. In a syntax directed translation scheme, if the value of an attribute of a node is a function of the values of the attributes of its children, then it is called a

a. Synthesized attribute
b. Inherited attribute
c. Canonical attribute
d. None of the above

Ans. a

Q44. Which of the following is not an intermediate code form?

a. Postfix notation
b. Syntax trees
c. Three address code
d. Quadruples

Ans. d

Q45. Three address codes can be implemented by

a. indirect triples
b. direct triples
c. both (a) and (b)
d. none of the above

Ans. a

Q46. In a bottom up evaluation of a syntax directed definition, inherited attributes can be

a. always be evaluated
b. be evaluated only if the definition is L-attributed
c. be evaluated only if the definition has synthesized attributes
d. none of the above

Ans. c

Q47. Three address code involves

a. at the most 3 address
b. exactly 3 address
c. no unary operators
d. none of the above

Ans. a

Q48. Inherited attribute is a natural choice in

a. keeping track of variable declaration
b. checking of the correct use of L-values and R-values
c. both (a) and (b)
d. none of the above

Ans. c

Practical Exam Sample Paper: Compiler Design Lab

1. Write a program to develop a lexical analyzer to recognize pattern **identifier** in C
2. Write a program to develop a lexical analyzer to recognize pattern **constants, comments** in C
3. Write a program to develop a lexical analyzer to recognize pattern **constants, operators** in C
4. Write a program to implement recursive descent parser….
5. Write a program to convert an infix notation into prefix notation
6. Write a program to convert an infix notation into postfix notation
7. Write a program to calculate the value of a postfix notation
8. Explain all the phases of compiler
9. Write a program to find out whether a given expression is valid or not
10. Write a program to perform following operations on a link list: Creation, Insertion and Display
11. Write a program to perform following operations on a link list: Creation, Insertion and Deletion
12. Generate Lexical Analyzer using LEX
13. Write a program to recognize a valid arithmetic expression that uses operators +,-,*,/ using YACC
14. Write a program to recognize a valid variable which starts with a letter followed by any number of letters or digits using YACC
15. Write a program to recognize the grammar $\mathbf{a^n}$ where n>=10
16. Explain the concept of LEX and YACC
17. Write a program to find the Macro Statements in a given C file (Use C file as input file)
18. Write a program to find number of white space characters in a C file.
19. Write a program which will take two input strings. Find all the possible sub common strings from the small String.

Text and Reference Books

- Bennett, Jeremy Peter. Introduction to compiling techniques: a first course using ANSI C, LEX and YACC. McGraw-Hill, Inc., 1996.
- Levine, John R., et al. Lex & yacc. " O'Reilly Media, Inc.", 1992.
- Aho, Alfred V., Jeffrey D. Ullman, and R. Sethi. "Principles of Compiler Construction." (1977).
- Louden, Kenneth C. "Compiler construction." Cengage Learning (1997).

Swami Keshvanand Institute of Technology, Management & Gramothan

Vision and Mission of Institute

Vision: “To promote higher learning in technology and industrial research to make our country a global player.”

Mission: “To promote quality education, training and research in the field of engineering by establishing effective interface with industry and to encourage the faculty to undertake industry sponsored projects for the students.”

Vision of CSE Department

Vision of CSE department is to:

V1: Produce quality computer engineers trained in latest tools and technologies.

V2: Be a leading department in the region and country by imparting in-depth knowledge to the students in emerging technologies in computer science & engineering.

Mission of CSE Department

Mission of CSE department is to:

Deliver resources in IT enable domain through:

M1: Effective Industry interaction and project-based learning.

M2: Motivating our students for employability, entrepreneurship, research and higher education.

M3: Providing excellent engineering skills in a state-of-the art infrastructure.

Program Educational Objectives of CSE department

The graduates of CSE program will be:

PEO1: Prepared to be employed in IT industries and be engaged in learning, understanding, and applying new ideas.

PEO2: Prepared to be responsible professionals in their domain of interest.

PEO3: Able to apply their technical knowledge as practicing professionals or engaged in higher education.

PEO4: Able to work efficiently as an individual and in a professional team environment.

Program Outcomes of CSE Department

PO 1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization for the solution of complex engineering problems.

PO 2: Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO 3: 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 public health and safety, and cultural, societal, and environmental considerations.

PO 4: 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.

PO 5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO 6: 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.

PO 7: 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.

PO 8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9: Individual and teamwork: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10: Communication: Communicate effectively on complex engineering activities with the engineering community and with the 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.

PO 11: 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.

PO 12: 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 of CSE Department

PSO1: Core Engineering Skills: Exhibit fundamental concepts of Data Structures, Databases, Operating Systems, Computer Network, Theory of Computation, Advanced Programming and Software Engineering.

PSO2: Standard Software Engineering practices: Demonstrate an ability to design, develop, test, debug, deploy, analyze, troubleshoot, maintain, manage and secure a software.

PSO3: Future Endeavors: Recognize the need to have knowledge of higher education institutions/ organizations/ companies related to computer science & engineering.

Swami Keshvanand Institute of Technology, Management &Gramothan, Jaipur

Programme: B.Tech. (Computer Science & Engineering)
Semester: V
Course Name (Course Code): Compiler Design Lab (5CS4-22)

Course Outcomes

After completion of this course, students will be able to –

	Course Outcome
5CS4-22.1	***Understand*** Lexical analyzer and ***use*** this knowledge to implement its various sub-functions for any regular language.
5CS4-22.2	***Create*** Symbol table and implement its operations in C language.
5CS4-22.3	***Define*** ex tool and ***determine*** various tokens using it.
5CS4-22.4	**Examine** string for the given regular expression and **evaluate** an expressions using YACC Tool.
5CS4-22.5	***Understand*** the concept of Context Free Grammar and ***break down*** the given grammar for calculating First.

Name of Faculty: Mayank Jain Deepa Modi

(Signature) [illegible] [illegible]

Verified by Course Coordinator

Signature [illegible]
(Name: GIRISH SHARMA

Verified by Verification and Validation Committee, DPAQIC

Signature [illegible]
(Name: Girish Sharma

SwamiKeshvanandInstituteof Technology,

Management&Gramothan,Jaipur

COURSE: **Compiler Design Lab** (5CS4-22)

C O	Outcomes	Bloom's Level	PO Indicators	PSO Indicators
Upon successful completion of this course, students should be able to:				
5CS4-21.1	***Understand*** Lexical analyzer and ***use*** this knowledge to implement its various sub-functions for any regular language.	2,3	PO - 1.2.1, 1.4.1,2.1.1, 2.1.2, 2.1.3,2.2.4,2.4.3,2.4.4,3.1.1, 3.2.1,3.2.3, 3.4.2,12.1.1, 12.1.2 ,12.2.1, 12.2.2, 12.3.1, 12.3.2	PSO(1.1.1,1.1.3)
5CS4-21.2	***Create*** Symbol table and implement its operations in C language.	6	PO - 1.3.1, 1.4.1,2.1.1, 2.1.2, 2.1.3,2.2.1, 2.2.2,2.2.4,2.4.3,2.4.4,3.1.1, 3.2.1,3.2.3, 3.4.2,12.1.1, 12.1.2 ,12.2.1, 12.2.2, 12.3.1, 12.3.2	PSO(1.1.1,1.1.3)
5CS4-21.3	***Define*** ex tool and ***determine*** various tokens using it.	1,3	PO - 1.3.1, 1.4.1,2.1, 2.1.2, 2.1.3,2.2.4,2.4.3,2.4.4,3.1.1, 3.2.1,3.2.3, 3.4.2, 5.1.1,5.1.2,5.2.1,5.3.1,5.3.2,12.1, 12.1.2 ,12.2.1, 12.2.2, 12.3.1, 12.3.2	PSO(1.1.3)
5CS4-21.4	**Examine** string for the given regular expression and **evaluate** an expressions using YACC Tool.	3,5	PO - 1.3.1, 1.4.1,2.1.1, 2.1.2, 2.1.3,2.2.4,2.4.3,2.4.4,3.1.1, 3.2.1,3.2.3, 3.4.2. 5.1.1, 5.1.2, 5.2.1,5.3.1,5.3.2,12.1.1, 12.1.2 ,12.2.1, 12.2.2, 12.3.1, 12.3.2	PSO(1.1.3)
5CS4-21.5	***Understand*** the concept of Context Free Grammar and ***break down*** the given grammar for calculating First.	4	PO - 1.3.1, 1.4.1,2.1.1, 2.1.2, 2.1.3,2.2.1, 2.2.2,2.2.4,2.4.3,2.4.4,3.1.1,3.2.1,3.2.3, 3.4.2,12.1.1, 12.1.2 ,12.2.1, 12.2.2, 12.3.1, 12.3.2	PSO(1.1.1,1.1.3)

Swami Keshvanand Institute of Technology, Management & Gramothan, Jaipur

CO-PO/PSO Mapping: Formulation and Justification

CO-PO/PSO Mapping

Programme: B.Tech. (Computer Science & Engineering)
Semester: V
Course Name (Course Code): Compiler Design Lab (5CS4-22)

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO1 0	PO1 1	PO 12	PSO 1	PSO 2	PSO 3
CO1	2	2	1									3	3		
CO2	2	2	1									3	3		
CO3	2	2	1		3							3	2		
CO4	2	2	1		3							3	2		
CO5	2	2	1									3	3		

[illegible]
(Signature) Mayank Jain

Deepa Modi
(Signature) [illegible]

Verified by Course Coordinator

Signature [illegible]
(Name: Girish Sharma

Verified by Verification and Validation Committee, DPAQIC

Signature [illegible]
(Name Girish Sharma)