

COMPILER CONSTRUCTION LAB

B.E. VII-Semester

LAB MANUAL

[MASTER MANUAL]

Prepared by

**C. Sireesha, Asst. Prof.
K. Shyam Sundar Reddy, Asst. Prof**



**DEPARTMENT OF INFORMATION TECHNOLOGY
VASAVI COLLEGE OF ENGINEERING
(AUTONOMOUS)**

INDEX

S.NO	DESCRIPTION	Page No.
1.	Institute Vision and Mission	1
2.	Department Vision and Mission	2
3.	PSO's, PEO's	3
4.	PO's	4
5.	List of CO with TL and mapping with PO/ PSO	5
6.	CO/ PO/ PSO Mapping Justification	6
7.	University / OU Syllabus	8
8.	List of Experiments – mappingwith CO/PO/PSO	9
9.	Introduction to the lab	10
	EXPERIMENTS	
1.	Implement lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.)	19
	Implementation of Lexical Analyzer using LEX tool	
	a) LEX program to recognize an integer number	30
	b) LEX program to recognize a float number	30
	c) LEX program to print the type of string	31
	d) LEX Program to delete a String.	32
	e) LEX Program to replace one string with another string	33
2.	f) LEX Program to delete a comment line	34
	g) LEX Program to identify octal or hexadecimal number using LEX	35
	h) LEX program to count number of vowels and consonants	36
	i) LEX program to count the number of words, characters, blank spaces and lines	37
	j) LEX program to covert upper case in to lower case	38
	k) LEX program for reversing a string	39
	l) Lexical Analyzer using LEX tool	41
3.	Implement "first" of a given context free grammar	45
4.	Implement "follow" of a given context free grammar	51
5.	Implement elimination of left recursion and left factoring algorithms for any given grammar and generate predictive parsing table.	54
6.	Write a program for generating derivation sequence for a given terminal string using SLR parsing table.	61
	Use LEX and YACC tool to implement Desktop Calculator.	
	Program to recognize strings using grammar ($a^n b^n, n \geq 0$)	71
7.	Program to recognize nested IF control statements and display the levels of nesting	73
	Program to recognize the grammar ($a^n b^n, n \geq 10$)	76
	Implementation of Desktop Calculator using LEX and YACC	78

8.	Implementation of code generation	81
9.	Implementation of code optimization techniques	86
10.	Major assignment: Intermediate code generation for subset C language.	
Additional Experiments		
11.	Implementation of Recursive Descent Parser for a given context free grammar.	92
12.	Generation of DAG for a given expression	97
13.	Implement symbol table using C Language.	100

INSTITUTE VISION AND MISSION

Vision

Striving for a symbiosis of technological excellence and human values.

Mission

To arm young brains with competitive technology and nurture holistic development of the individuals for a better tomorrow.

Our Quality Policy

Education without quality is like a flower without fragrance. It is our earnest resolve to strive towards imparting high standards of teaching, training and developing human resources.

DEPARTMENT VISION AND MISSION

Vision

To be a centre of excellence in core Information Technology and multidisciplinary learning and research, where students get trained in latest technologies for professional and societal growth.

Mission

To enable the students acquire skills related to latest technologies in IT through practice- oriented teaching and training.

PROGRAM EDUCATIONAL OBJECTIVES (PEO's)

The educational objectives of UG program in Information Technology are:

- **PEO1:** With theoretical and practical knowledge to obtain employment or pursue higher studies and solve problems in Information Technology.
- **PEO2:** With effective written and oral communication skills that will help them to work in diversified and dynamic working environments.
- **PEO3:** With competence to succeed in their professional lives with ethical values.

PROGRAM SPECIFIC OUTCOMES (PSO's)

The B.E in Information Technology will demonstrate:

- **PSO1:** Competency in programming using different programming languages to implement algorithms.
- **PSO2:** Competency in the analysis and design of a software solution using different modelling tools.
- **PSO3:** Competency in Electronic Design and Embedded System Design using different simulation tools.

PROGRAM OUTCOMES (POs)

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **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.
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 the public health and safety, and the cultural, societal, and environmental considerations.
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.
5. **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.
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.
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.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **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.
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.
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.

COURSE OUTCOMES:

Course Name:

S.No	Course Outcomes	BTL
1	Implement simple lexical analyzer	3
2	Generate predictive parsing table for a CFG	6
3	Apply Lex and Yacc tools to develop a scanner & parser	3
4	Implement LR parser	6
5	Implement Intermediate code generation for subset C language	6

MAPPING LEVELS:

Correlation levels 1, 2 or 3 as defined below:

1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High)

CO/PO/PSO MAPPING

CO/ PO /PSO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
CO1	3	3	2	2	2	-	-	-	-	-	-	2	3	3	
CO2	3	3	3	3	2	-	-	-	-	-	-	3	3	2	
CO3	3	3	3	3	3	-	-	-	-	-	-	3	2	3	
CO4	3	3	3	3	2	-	-	-	-	-	-	3	3	2	
CO5	1	3	2	-	-	-	-	-	-	-	-	2	3	2	

CO/PO/PSO JUSTIFICATION

MAPPING	CORRELATION LEVELS	JUSTIFICATION
CO1-PO1	3	The knowledge of Finite automata and Regular expressions will help the students to apply the same to formulate solutions for engineering problems.
CO1-PO2	3	The knowledge of Finite automata and Regular expressions will help the students to apply the same to identify and analyze engineering problems.
CO1-PO3	2	Thorough understanding of Finite automata and Regular expressions will help in the design and development of abstract models for computational problems.
CO1-PO4	2	Implementation of Lexical analyzer will help in conducting detailed investigation of complex engineering problems.
CO1-PO5	2	Students will be able to use modern IT tools such as LEX or FLEX to model complex engineering problems
CO1-PO12	2	Information acquired from the compilation phases provides lifelong learning in the context of Compiler Construction.
CO2-PO1	3	The knowledge of Context-free grammars and Predictive parsers will help the students to apply the same to formulate solutions for engineering problems.
CO2-PO2	3	The knowledge of Context-free grammars and Predictive parsers will help the students to apply the same to identify and analyze engineering problems.
CO2-PO3	3	Thorough understanding of Context-free grammars and Predictive parsers will help in the design and development of abstract models for computational problems.
CO2-PO4	3	Implementation of top-down parsers will help in conducting detailed investigation of complex engineering problems.
CO2-PO5	2	Students will be able to use modern IT tools to model complex engineering problems
CO2-PO12	3	Information acquired from the compilation phases provides lifelong learning in the context of Compiler Construction.
CO3-PO1	3	The knowledge of Regular expression and Context-free grammars will help the students to apply the same to formulate solutions for engineering problems.

CO3-PO2	3	The knowledge of Regular expression and Context-free grammars will help the students to apply the same to identify and analyze engineering problems.
CO3-PO3	3	Thorough understanding of Regular expression and Context-free grammars will help in the design and development of abstract models for computational problems.
CO3-PO4	3	Implementation of Scanners and parsers will help in conducting detailed investigation of complex engineering problems.
CO3-PO5	3	Students will be able to use modern IT tools such as LEX/FLEX/YACC to model complex engineering problems
CO3-PO12	3	Information acquired from the compilation phases provides lifelong learning in the context of Compiler Construction.
CO4-PO1	3	The knowledge of Context-free grammars and LR parsers will help the students to apply the same to formulate solutions for engineering problems.
CO4-PO2	3	The knowledge of Context-free grammars and LR parsers will help the students to apply the same to identify and analyze engineering problems.
CO4-PO3	3	Thorough understanding of Context-free grammars and LR parsers will help in the design and development of abstract models for computational problems.
CO4-PO4	3	Implementation of bottom-up parsers will help in conducting detailed investigation of complex engineering problems.
CO4-PO5	2	Students will be able to use modern IT tools such as YACC/ANTLR to model complex engineering problems
CO4-PO12	3	Information acquired from the compilation phases provides lifelong learning in the context of Compiler Construction.
CO5-PO1	3	The knowledge of intermediate representation will help the students to formulate solutions for engineering problems.
CO5-PO2	3	The knowledge of intermediate representation will help the students to apply the same to identify and analyze engineering problems.
CO5-PO3	2	Thorough understanding of intermediate representation will help in the design and development of abstract models for computational problems.
CO5-PO12	2	Information acquired from the compilation phases provides lifelong learning in the context of Compiler Construction.

VASAVI COLLEGE OF ENGINEERING (AUTONOMOUS)
9-5-81, Ibrahimbagh, Hyderabad-500031, Telangana State
DEPARTMENT OF INFORMATION TECHNOLOGY
COMPILER CONSTRUCTION LAB
(B.E. Semester-VI)

Instruction : 2 Hrs/week	SEE Marks : 50	Course Code : PC621IT
Credits : 1	CIE Marks: 25	Duration of SEE : 3 Hours

Course Objectives	Course Outcomes
The course will enable the students to:	At the end of the course student will be able to:
Learn to implement the different Phases of compiler	<ol style="list-style-type: none"> 1. Implement simple lexical analyzer 2. Generate predictive parsing table for a CFG 3. Apply Lex and Yacc tools to develop a scanner & parser 4. Implement LR parser 5. Implement Intermediate code generation for subset C language

LIST OF EXPERIMENTS

1. Implement lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.)
2. Implementation of Lexical Analyzer using LEX tool
3. Implement "first" of a given context free grammar
4. Implement "follow" of a given context free grammar
5. Implement elimination of left recursion and left factoring algorithms for any given grammar and generate predictive parsing table.
6. Write a program for generating derivation sequence for a given terminal string using SLR parsing table.
7. Use LEX and YACC tool to implement Desktop Calculator.
8. Implementation of code generation
9. Implementation of code optimization techniques
10. Major assignment: Intermediate code generation for subset C language.

Suggested Reading:

1. Aho, RaviSethi, Monica S Lam, Ullman, Compilers-Principle, Techniques and Tools 2nd Edition , Pearson, 2002
2. John R Levine, Tony Mason, Doug Broun, Lex and Yacc, Orielly, 2nd Edition, 2009

List of Experiments with CO / PO / PSO Mapping

S.No	Name of the Experiment	CO mapping	PO/PSO mapping
1	Implement lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.)	CO 1	PO1,2,3,4,5,12 PSO1,2
2	Implementation of Lexical Analyzer using LEX tool	CO 3	PO1,2,3,4,5,12 PSO1,2
3	Implement "first" of a given context free grammar	CO 1	PO1,2,3,4,5,12 PSO1,2
4	Implement "follow" of a given context free grammar	CO 1	PO1,2,3,4,5,12 PSO1,2
5	Implement elimination of left recursion and left factoring algorithms for any given grammar and generate predictive parsing table	CO 2	PO1,2,3,4,5,12 PSO1,2
6	Write a program for generating derivation sequence for a given terminal string using SLR parsing table.	CO 4	PO1,2,3,4,5,12 PSO1,2
7	Use LEX and YACC tool to implement Desktop Calculator.	CO 3	PO1,2,3,4,5,12 PSO1,2
8	Implementation of code generation	CO 5	PO1,2,3,12 PSO1,2
9	Implementation of code optimization techniques	CO 5	PO1,2,3,12 PSO1,2
10	Major assignment: Intermediate code generation for subset C language.	CO 5	PO1,2,3,12 PSO1,2
Additional Experiments:			
1		CO 1	PO1, PO10
2		CO 1	PO1, PO10

INTRODUCTION TO COMPILER CONSTRUCTION LAB

A *computer* is a machine. It requires directions (control) to operate. *Programs* provide this control. *Computer program* is a structured combination of data and instructions that operate a computer. Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called *compilers*.

1.1 Evolution of Computer Languages

Like natural languages (Hindi, English, etc.) programming languages are also notations, with which common man can communicate to the computer.

Machine Languages

- Consists of a sequence of zeros and ones.
- Each kind of CPU has its own machine language
- The only way to directly communicate with the computer
- Machine-dependent (programs written in machine language are not portable)
- Tedious and error-prone for humans, requires highly trained programmers
- Not programmer friendly
- No translation required
- Fast and efficient processing for the computer

Assembly Languages

- Programs written in mnemonics (alphabetic abbreviations)
- Requires a program called assembler to convert to machine code before execution
- Machine-dependent
- Easier for the programmer, still tedious and requires highly trained programmers

High-Level Languages

- More closely resemble the English language
- Requires either an interpreter or a compiler to convert into machine codes before execution
- Machine-independent
- Programmer-friendly
- Need to be translated

- Most high-level languages (COBOL, FORTRAN, BASIC, Pascal, C, etc.) are procedure-oriented languages. The emphasis is on how to accomplish a task
- Visual Basic, C++, Smalltalk, and Java are object-oriented languages. The emphasis is on the objects included in the user interface and the events that occur on those objects
 - BASIC: Beginner's All-purpose Symbolic Instruction Code
 - COBOL: Common Business-Oriented Language

1.2 Importance of Compiler Construction Lab

Every software has to be translated to its equivalent machine instruction form so that it will be executed by the underlying machine. There are many different types of system softwares that help during this translation process. Compiler is one such an important System Software that converts High level language programs to its equivalent machine (low level) language. It is impossible to learn and use machine language in software development for the users. Therefore we use high level computer languages to write programs and then convert such programs into machine understandable form with the help of mediator softwares such as compilers, interpreters, assemblers etc. Thus compiler bridges the gap between the user and the machine, i.e., computer.

It's a very complicated piece of software which took 18 man years to build first compiler .To build this software we must understand the principles, tools, and techniques used in its working. The compiler goes through the following sequence of steps called phases of a compiler.

- 1) Lexical Analysis
- 2) Syntax Analysis
- 3) Semantic Analysis
- 4) Intermediate Code Generation
- 5) Code Optimization
- 6) Code Generation.

In performing its role, compiler also uses the services of two essential components namely, Symbol Table and Error Handler

The objective of the Compiler Construction Laboratory is to understand and implement the principles, techniques, and also available tools used in compiler construction process. This will enable the students to work in the development phase of new computer languages in industry.

1.3 Language Translator

A *translator* is a program that takes input a program written in one programming language and produces as output a program in another language.

There are three types of translators:

1. Compilers
2. Interpreters
3. Assemblers

Source Language \Rightarrow Target Language

Compilers

A *compiler* is a program that reads a program written in one language-the source language-and translates it into an equivalent program in another language-the target language. Usually the source language is a *high level language* like Java, C, FORTRAN, etc., whereas the target language is a low level language such as an assembly language or machine language. The translation process should also report the presence of errors in the source program.

A program that translates from a low level language to a higher level one is a **de-compiler**.

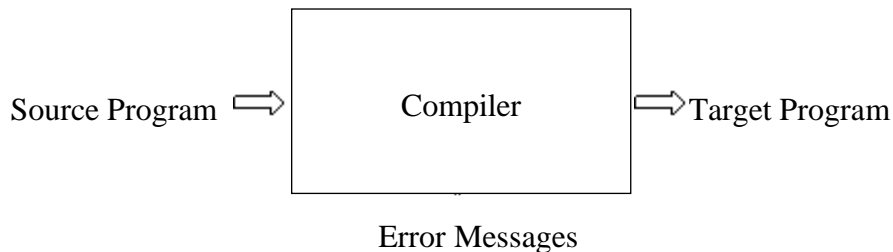


Fig 1.1: A Compiler

If the target program is an assembly language program then an *assembler* is used to convert it into machine language. If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.



Fig 1.2: Running the target program

Compilers are sometimes classified as *single-pass*, *multi-pass*, *load-and-go*, *debugging*, or *optimizing*, depending on how they have been constructed or on what function they are supposed to perform. Despite this apparent complexity, the basic tasks that any compiler must perform are essentially the same.

In addition to a compiler, several other programs may be required to create an executable target program.

Interpreters: An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation (it executes the source program immediately rather than generating object code), an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement. In principle, any programming language can be either interpreted or compiled. Many languages have been implemented using both compilers and interpreters, including [Lisp](#), [Pascal](#), [C](#), [BASIC](#), and [Python](#).

An interpreter translates source code into some efficient intermediate representation (Byte code) and immediately executes this. Speed of execution is slower than compiled code by a factor of 10 or more.

Assemblers: An *assembler* is a program that translates an assembly language program, written in a particular assembly language, into a particular machine language. An assembler is a language translator whose source language is assembly language.

Preprocessors: A *preprocessor* is a separate program that is called by the compiler before actual translation begins. Preprocessors produce input to compilers. They may perform the following functions:

1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text. For example `#include<stdio.h>`, by this statement the header file `stdio.h` can be included and user can make use of the functions defined in this header file.

Linkers: Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the

memory location where these codes will be loaded, making the program instruction to have absolute references.

Loaders: Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

The Structure of a Compiler (or) The Phases of a Compiler

A compiler takes as input a source program and produces as output an equivalent target program. As this process of compilation is highly complex, it is split into a series of sub processes called phases. The entire s/w is structured into various stages (phases). All these stages are operated in sequence. There are mainly two parts of compilation: *analysis* and *synthesis*

Analysis (Machine independent/Language dependent phase): The analysis part is often called the *front end* of the compiler. Analysis part read the source program and understands its structure and meaning. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

Synthesis (Machine dependent/Language independent phase): The synthesis part is often called the *back end* of the compiler. The synthesis part constructs the desired target program from the intermediate representation and the information from the symbol table.

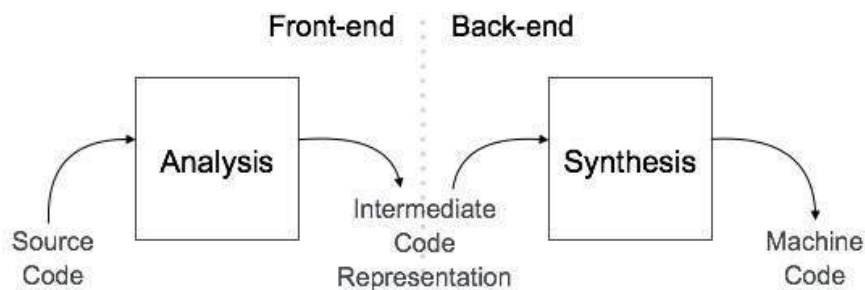


Fig 1.4: Basic structure of a compiler

Compiler operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in fig. 1.5.

Some compilers have a *machine-independent optimization* phase between the front end and the back end. The purpose of this phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program that it would have otherwise produced from an un-optimized intermediate representation.

The first four phases, forms the bulk of the analysis portion of a compiler. Code Optimization is an optional phase, and synthesis part consists of Code Generator phase. Symbol table management and error handling, are shown interacting with the six phases.

Lexical Analysis

The first phase of a compiler is called *lexical analyzer* or *scanner*. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form

<token-name, attribute-value>

that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.

Syntax Analysis

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

Semantic Analysis

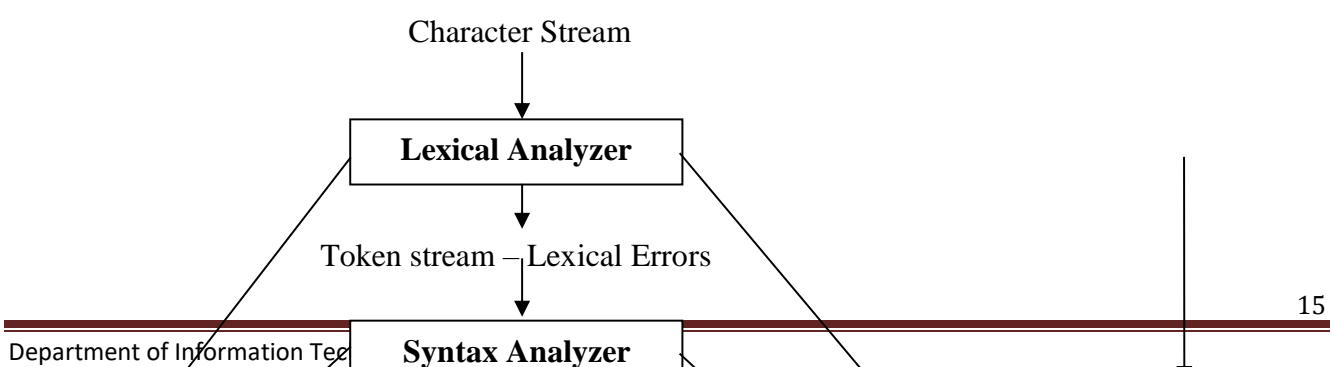
Semantic analysis checks the source program for semantic errors. The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. An important part of semantic analysis is *type checking*, where the compiler checks that each operator has matching operands.

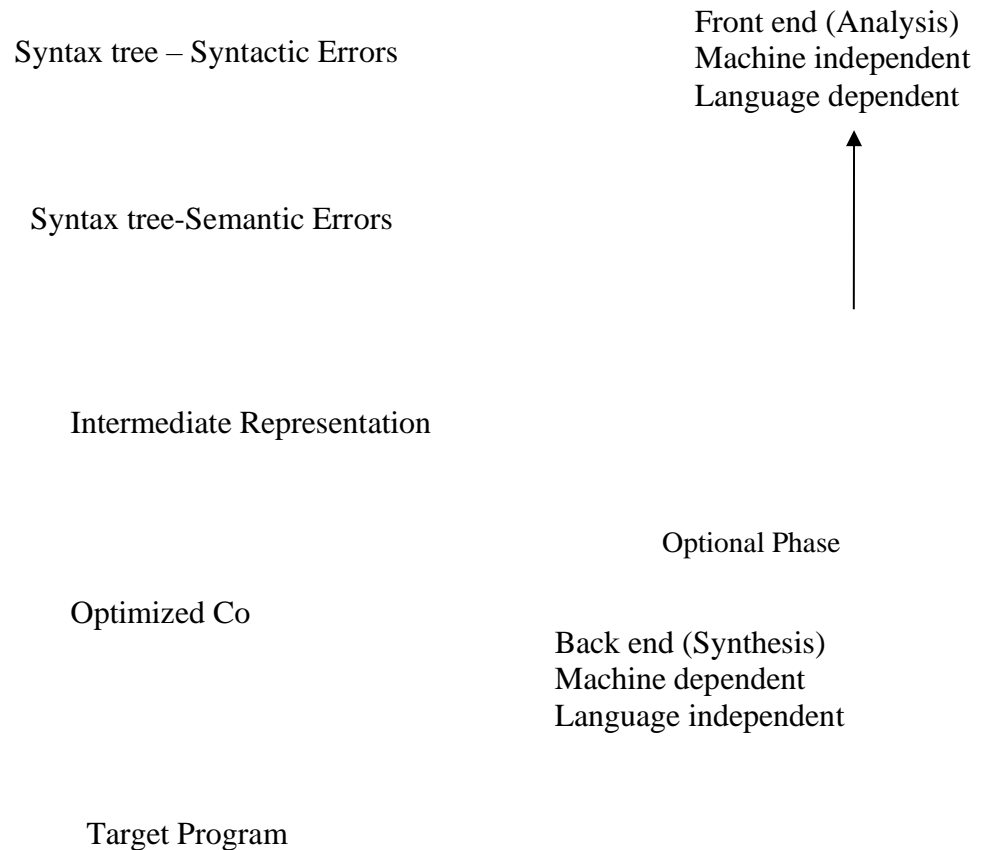
Intermediate Code Generation

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: *it should be easy to produce* and *it should be easy to translate into the target program*.

One popular type of intermediate form called **three-address code**, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

Fig 1.5: Phases of a compiler





Code Optimization

Intermediate code generation process introduces many inefficiencies, such as, extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc. The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. It reduces the code by removing repeated or unwanted instructions from the intermediate code.

Code Generation

Code generation is the final phase of the compiler. The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

Symbol Table Management

An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. These attributes may provide information about the storage allocated for an identifier, its type, its scope, and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned. A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

Literal Table Management

A literal table maintains the details of constants and strings used in the program. It reduces the size of a program in memory by allowing reuse of constants and strings. It also needed by the code generator to construct symbolic addresses for literals.

Error Handler

Error handler is invoked when a fault in the source program is detected. Each phase can encounter errors. The syntax and semantic analysis phases usually handle a large number of errors. A list of errors encountered by various phases is given below.

1. Lexical analyzer: Misspelled tokens (ex: a= 2b)
2. Syntax analyzer: Syntax errors like missing parenthesis, misplaced semicolon, etc.
Ex: (a+b-- Right parenthesis missing
3. Semantic analyzer: Incompatible operands for an operator
4. Code optimizer: Unreachable statements
5. Code generator: Memory restrictions to store a variable.

Passes: A pass refers to the traversal of a compiler through the entire

Phase: A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. **Single-pass Compiler:** A single pass compiler (All the phases are combined into a single group) makes a single pass over the source text, parsing, analyzing, and generating code all at once.

Multi-pass Compiler: A multi pass compiler (More than one phase are combined into a number of groups called multi-pass) makes several passes over the program.

EXP. NO: 1: Standalone Lexical Analyzer Program using C

Aim: To develop a lexical analyzer to recognize a few patterns

Description about Experiment:

The first phase of a compiler is called lexical analyzer or scanner. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form

<token-name, attribute-value>

that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token

Algorithm:

function lexan: integer ; [Returns an integer encoding of token]

var *lexbuf*: array[0 .. 100] of char ;

c : char ;

begin

loop begin

 read a character into *c* ;

if *c* is a blank or a tab **then**

 do nothing

else if *c* is a newline **then**

lineno := *lineno* + 1

else if *c* is a digit **then begin**

 set *tokenval* to the value of this and following digits ;

return NUM

end

else if *c* is a letter **then begin**

```

    place c and successive letters and digits into lexbuf;
    p := lookup ( lexbuf ) ;
    if p = 0 then
        p := insert ( lexbf, ID ) ;
    tokenval := p
    return the token field of table entry p
end
else
    set tokenval to NONE ; /* there is no attribute */
    return integer encoding of character c
end
end

```

Source Code:

```

#include<stdio.h>
#include<ctype.h>
#include<string.h>
int main()
{
    FILE *input,*output;
    int l=1;
    int t=0;
    int j=0;
    int i,flag;
    char ch,str[20];
    char keyword[30][30]={ "int","main","if","else","do","while","struct","goto" };
    input=fopen("input.txt","r");
    output=fopen("output.txt","w");
    fprintf(output,"Line no. \t Token no.\t Token\t Lexeme\n\n");
    while(!feof(input))
    {
        i=0;
        flag=0;

```

```

ch=fgetc(input);
if( ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='<' || ch=='>')
{
    fprintf(output,"%6d\t\t %6d\t\t Operator\t %6c\n",l,t,ch);
    t++;
}
else if( ch==';' || ch=='{' || ch=='}' || ch=='(' || ch==')' || ch=='?' || ch=='@' || ch=='!' || ch=='%')
{
    fprintf(output,"%6d\t\t %6d\t\t Special symbol\t %6c\n",l,t,ch);
    t++;
}
else if(isdigit(ch))
{
    fprintf(output,"%6d\t\t %6d\t\t Digit\t\t %6c\n",l,t,ch);
    t++;
}
else if(isalpha(ch))
{
    str[i]=ch;
    i++;
    ch=fgetc(input);
    while(isalnum(ch)&& ch!=' ')
    {
        str[i]=ch;
        i++;
        ch=fgetc(input);
    }
    str[i]='\0';
    for(j=0;j<30;j++)
    {
        if(strcmp(str,keyword[j])==0)
        {
            flag=1;
            break;

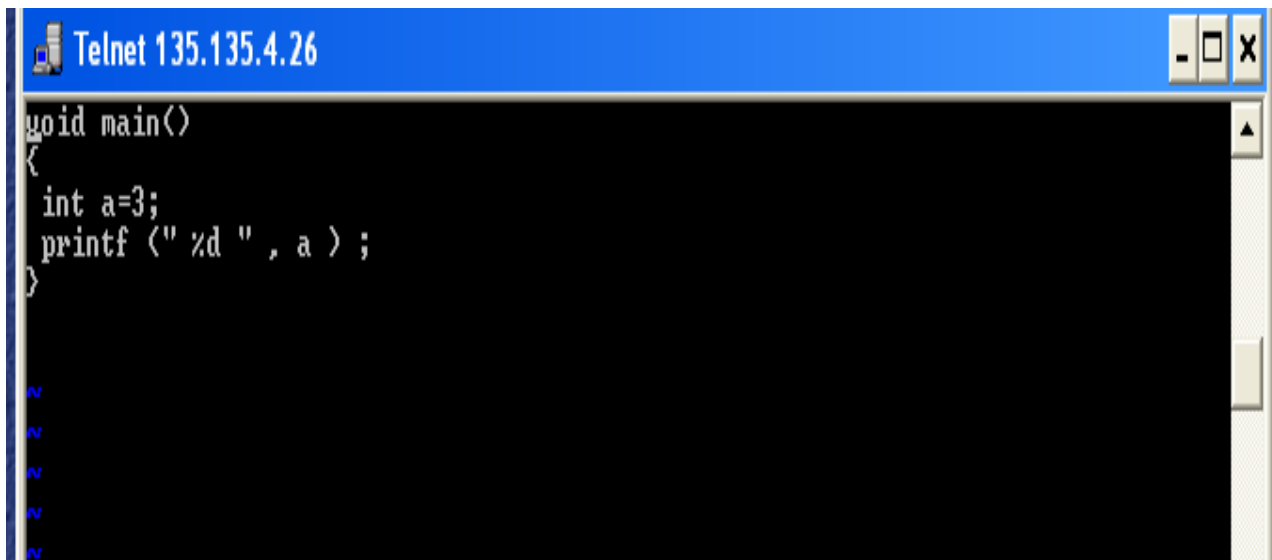
```



```
    }
}
if(flag==1)
{
    fprintf(output,"%6d\t\t %6d\t\t Keyword\t %6s\n",l,t,str);
    t++;
}
else
{
    fprintf(output,"%6d\t\t %6d\t\t Identifier\t %6s\n",l,t,str);
    t++;
}
}
else if(ch=="\n")
{
    l++;
}
}
fclose(input);
fclose(output);
output=fopen("output.txt","r");
while(!feof(output))
{
    ch=fgetc(output);
    printf("%c",ch);
}
fclose(output);
return 0;
}
```

OUTPUT:

Input.txt file

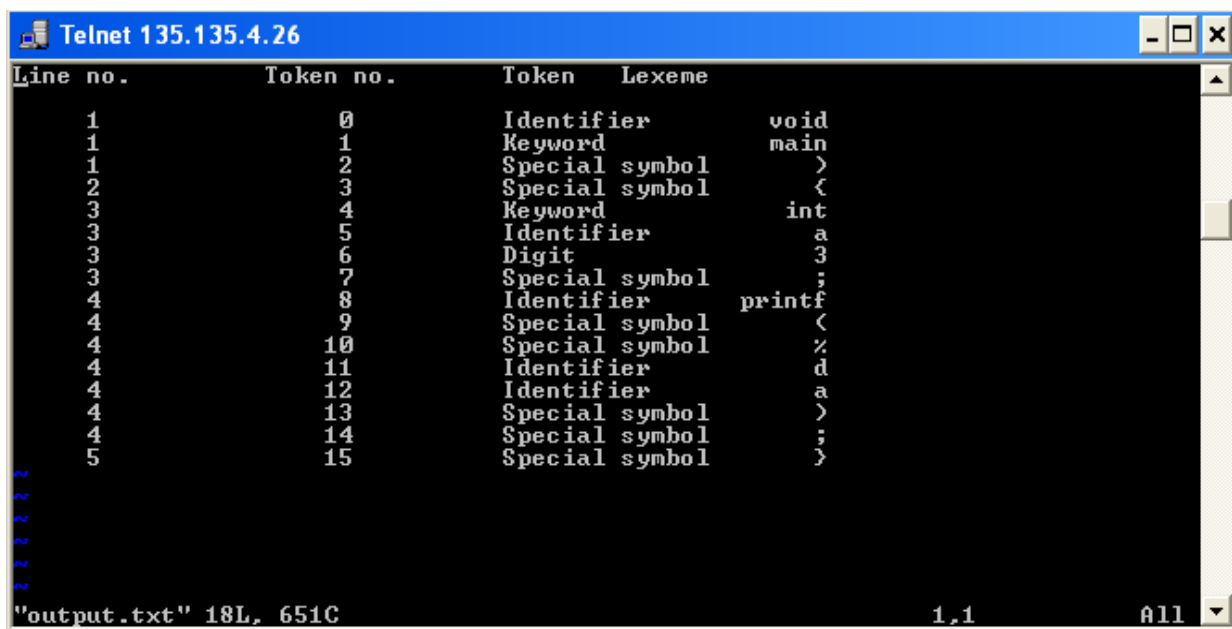


```

void main()
{
    int a=3;
    printf (" %d " , a ) ;
}

```

Output.txt file:



Line no.	Token no.	Token	Lexeme
1	0	Identifier	void
1	1	Keyword	main
1	2	Special symbol	>
2	3	Special symbol	<
3	4	Keyword	int
3	5	Identifier	a
3	6	Digit	3
3	7	Special symbol	;
4	8	Identifier	printf
4	9	Special symbol	<
4	10	Special symbol	%
4	11	Identifier	d
4	12	Identifier	a
4	13	Special symbol	>
4	14	Special symbol	;
5	15	Special symbol	>

"output.txt" 18L, 651C 1,1 All

Result:

The program to develop lexical analyzer using C was executed successfully

Experimental Viva Questions:

1. Describe lexical analysis phase?
2. List out the type of tokens?
3. Find tokens in the below program

```

int main()
{
    int x, y, total;
    x = 10, y = 20;
}

```

```
total = x + y;  
Printf ("Total = %d \n", total);  
}
```

5. What are properties Identifiers in C language?

6. Rules for constructing identifier name in C

INTRODUCTION TO LEX TOOL

LEX: A Lexical Analyzer Generator

There is a wide range of tools for construction of lexical analyzers. The majority of these tools are based on regular expressions. One of the traditional tools of that kind is *Lex* (or in a more recent implementation Flex). The input notation for the *Lex* tool is referred to as the *Lex language* and the tool itself is the *Lex compiler*. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called `lex.yy.c` that simulates this transition diagram. Lex/ Flex are based on the concept that for a regular expression we can always create a DFA.

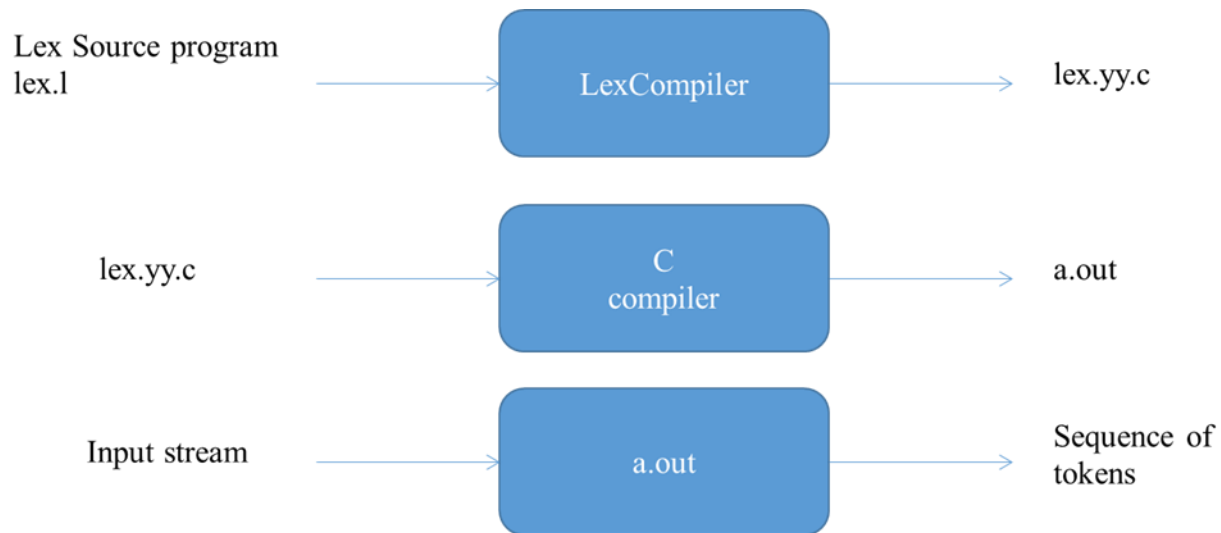


Fig. : Creating a Lexical Analyzer with Lex

The above figure suggests how Lex is used. An input file, which we call `lex.l`, is written in the lex language that describes the lexical analyzer to be generated. The *Lex compiler* transforms `lex.l` to a C program, in a file that is always named `lex.yy.c`. Finally, `lex.yy.c` is run through the C compiler to produce an object program `a.out`. The C compiler output i.e., `a.out` is a working lexical analyzer that can take a stream of input character and produce a stream of tokens.

Structure of Lex Programs

A Lex program has the following form:

```

Declarations
%%
Translation rules
%%
Auxiliary functions

```

A Lex program consists of three parts. The declarations section includes declarations of variables, *manifest constants*, and regular definitions. A manifest constant is an identifier that is declared to represent a constant. For example, the name of a token.

The translation rules each have the form

```
Pattern { Action }
```

For example,

```

Pattern1 { Action1 }
Pattern2 { Action2 }
.
.
.
Patternn { Actionn }

```

Here each pattern P_i is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created. Each fragment describes what action the lexical analyzer should take when Pattern P_i matches a lexeme i .

The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer. When lexical analyzer called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the largest prefix of the input that matches one of the patterns P_i . It then executes the associated action A_i . Typically, A_i will return to the parser, but if it does not (e.g., because P_i describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable *yylval* to pass additional information about the lexeme found, if needed.

The Lex program that recognizes the tokens of Fig. 1.15 is given below.

```
% {
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
% }

/* regular definitions */
delim      [ \t\n]
ws         { delim }+
letter     [A-Za-z]
digit      [0-9]
id         { letter } ( { letter } | { digit } ) *
number     { digit } ( \. { digit } + ) ? ( E [ + - ] ? { digit } + ) ?
%%

{ ws }     { /* no action and no return */ }
if         { return(IF); }
then       { return(THEN); }
else       { return(ELSE); }
{ id }     { yylval = (int) installID(); return(ID); }
{ number } { yylval = (int) installNum(); return(NUMBER); }
“<”       { yylval = LT; return(RELOP); }
“<=”      { yylval = LE; return(RELOP); }
“=”       { yylval = EQ; return(RELOP); }
“<”       { yylval = NE; return(RELOP); }
“>”       { yylval = GT; return(RELOP); }
“>=”      { yylval = GE; return(RELOP); }
%%
```

IntinstallID() { /* funtion to install the lexeme, whose
first character is pointed to by yytext,
and whose length is yyleng, into the

```

symbol table and return a pointer thereto */
}

IntinstallNum() { /* similar to installID, but puts
numerical constants into a separate table */
}

```

Incidentally, the *yy* that appears in *yylval* and *lex.yy.c* refers to the YACC parser generator, which is commonly used in conjunction with *Lex*. In above program we see a pair of special brackets, `%{` and `%}`, anything within these brackets is copied directly to the file *lex.yy.c*, and is not treated as a regular definition. Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces. Thus, for instance, *delim* is defined to be a shorthand for the character class consisting of the blank, the tab, and the newline; then *ws* is defined to be one or more delimiters, by the regular definition `{delem}+`.

Notice that in the definition of *id* and *number*, parenthesis are used as grouping meta symbols and do not stand for themselves. In contrast, *E* in the definition of *number* stands for itself. If we wish to use one of the Lex meta symbols, such as any of the parenthesis, `+`, `*` or `?`, to stand for themselves, we may proceed them with a backslash. For instance, we see `\.` in the definition of *number*, to represent the dot, since that character is a meta symbol representing “any character” as usual in UNIX regular expressions. Regular expressions in LEX are composed of metacharacters (Table 1).

Pattern-matching examples are shown in Table 2.

Metacharacter	Matches
<code>.</code>	any character except newline
<code>\n</code>	newline
<code>*</code>	zero or more copies of the preceding expression
<code>+</code>	one or more copies of the preceding expression
<code>?</code>	zero or one copy of the preceding expression
<code>^</code>	beginning of line
<code>\$</code>	end of line
<code>a b</code>	a or b
<code>(ab) +</code>	one or more copies of ab (grouping)
<code>"a+b"</code>	literal "a+b" (C escapes still work)
<code>[]</code>	character class

Table 1: Pattern Matching Primitives

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbcb abcbcbcb ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\ -z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a b]	one of: a, , b
a b	one of: a, b

Table 2: Pattern Matching Examples

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

Table 3: Lex Predefined Variables

1. Write a LEX program to recognize an integer number

```
%%
[+-]?[0-9]+ {printf("integer is recognized");}
.* {printf("integer is not recognized");}
%%
```

OUT PUT:

2. Write a LEX program to recognize a float number

OUT PUT:

3. Write a LEX program to print the type of string.

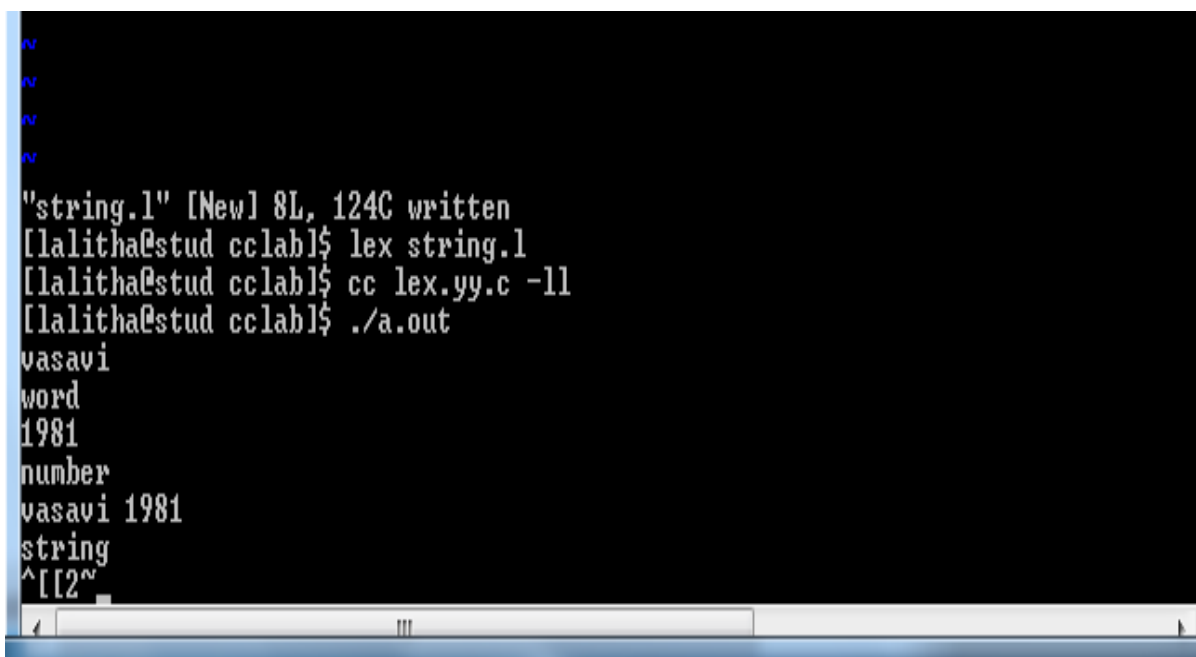
```
% {
#include <stdio.h>

% }

%%

[a-zA-Z]* {printf("word");}
[0-9]* {printf("number");}
[0-9 a-zA-Z]* {printf("string");}

%%
```

OUT PUT:


```
"string.l" [New] 8L, 124C written
[lalitha@stud cclab]$ lex string.l
[lalitha@stud cclab]$ cc lex.yy.c -ll
[lalitha@stud cclab]$ ./a.out
vasavi
word
1981
number
vasavi 1981
string
^[[2~
```

4. LEX Program to delete a String.

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

% }

%%

"ENGINEERING" {printf(" ");}

%%

main()
{
yylex();
return 0;
```

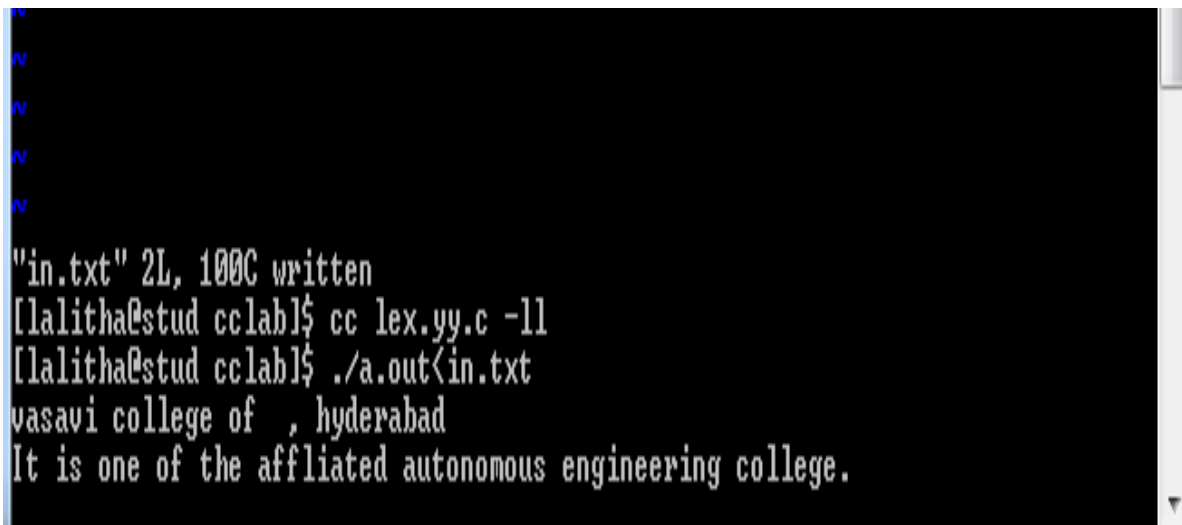
```
}  
yywrap()  
{  
return 0;  
}
```

OUTPUT:

vi in.txt

VASAVI COLLEGE OF ENGINEERING

VASAVI ENGINEERING COLLEGE



```
"in.txt" 2L, 100C written  
[lalitha@stud cclab]$ cc lex.yy.c -ll  
[lalitha@stud cclab]$ ./a.out<in.txt  
vasavi college of , hyderabad  
It is one of the affiliated autonomous engineering college.
```

5. LEX Program to replace one string with another string

```
%{
#include<stdio.h>
%}
%%
"ENGG"  {printf("ENGINEERING");}
"IT"    {printf("INFORMATION TECHNOLOGY");}
"VCE"   {printf("VASAVI COLLEGE OF ENGINEERING");}
%%
main()
{
yylex();
return 0;
}
yywrap()
{
return 0;
}
```

OUT PUT:

vi replace.txt

DEPARTMENT OF IT

VCE

VASAVI COLLEGE OF ENGG



```
"replace.l" 17L, 214C written
[lalitha@stud cclab]$ lex replace.l
[lalitha@stud cclab]$ cc lex.yy.c -ll
[lalitha@stud cclab]$ ./a.out<rep.txt
DEPARTMENT OF INFORMATION TECHNOLOGY
VASAVI COLLEGE OF ENGINEERING
HYDERABAD
```

6. LEX

Program to delete a comment line

```
%{
```

```
#include<stdio.h>

% }

%%

"/*"([a-z A-Z][0-9][ ])*"/" {printf(" ");}

%%

main()
{
yylex();
return 0;
}

yywrap()
{
return 0;
}
```

OUT PUT:

```
lex delcom.l
gcc lex.yy.c -ll
./a.out<pro.txt
```

```
vi pro.txt
/*this is a simple program*/
main()
{
printf("hi");
}
```

OUTPUT:

```

"comment.1" 15L, 131C written
[lalitha@stud cclab]$ lex comment.1
[lalitha@stud cclab]$ cc lex.yy.c -ll
[lalitha@stud cclab]$ ./a.out<pro.txt

main()
{
printf('hi');
}

```

Program to identify octal or hexadecimal number using LEX

```

%{
    /*program to identify octal and hexadecimal numbers*/
%}
Oct [o][0-8]+
Hex [o][x|X][0-9A-F]+
%%
{Hex} printf("this is a hexadecimal number");
{Oct} printf("this is an octal number");
%%
main()
{
yylex();
}
int yywrap()
{return 1;}

```

OUT PUT

```

"octal.1" [New] 18L, 247C written
[lalitha@stud cclab]$ lex octal.1
[lalitha@stud cclab]$ cc lex.yy.c -ll
[lalitha@stud cclab]$ ./a.out
o123
this is an octal number
123x
123x
0123x
0123x
x123
x123
ox123
this is a hexadecimal number

```

8. LEX program to count number of vowels and consonants

```

%{
    int v=0,c=0;
}%
%%
[aeiouAEIOU] v++;
[a-zA-Z] c++;
%%
main()
{
    printf("ENTER INPUT : \n");
    yylex();
    printf("VOWELS=%d\nCONSONANTS=%d\n",v,c);
}

```

Output:

10. LEX program to count the number of words, characters, blank spaces and lines

```

"vowels.l" 27L, 186C written
[lalitha@stud cclab]$ ./a.out
ENTER INPUT :
vasavi

VOWELS=3
CONSONANTS=3
[lalitha@stud cclab]$

```

```

% {
int c=0,w=0,l=0,s=0;
% }
%%

[\n] l++;
[' '\n\t] s++;
[^' '\t\n]+ w++; c+=yyleng;

%%

int main(int argc, char *argv[])
{
if(argc==2)
{
yyin=fopen(argv[1],"r");
yylex();
printf("\nNUMBER OF SPACES = %d",s);
printf("\nCHARACTER=%d",c);
printf("\nLINES=%d",l);
printf("\nWORD=%d\n",w);
}
}

```

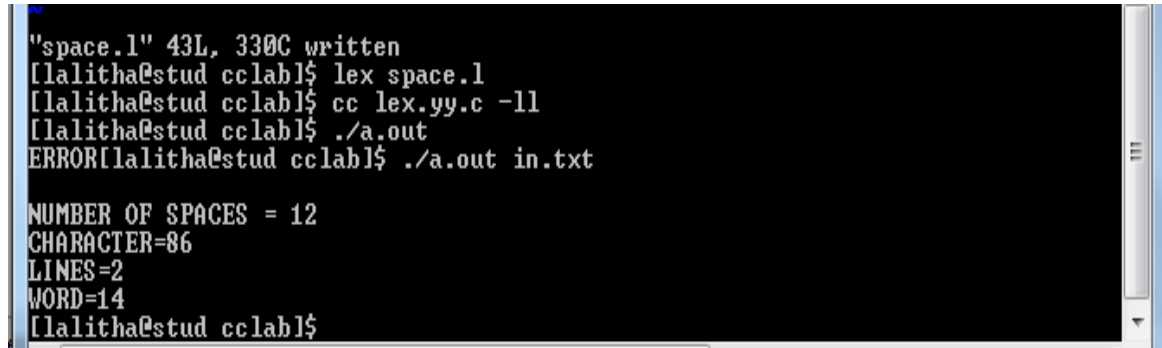


```

else
printf("ERROR");
}

```

OUTPUT:



```

"space.l" 43L, 330C written
[lalitha@stud cclab]$ lex space.l
[lalitha@stud cclab]$ cc lex.yy.c -ll
[lalitha@stud cclab]$ ./a.out
ERROR[lalitha@stud cclab]$ ./a.out in.txt

NUMBER OF SPACES = 12
CHARACTER=86
LINES=2
WORD=14
[lalitha@stud cclab]$

```

11. LEX program to covert upper case in to lower case

```

%{
#include<stdio.h>
#include<ctype.h>
%}
let [a-z][A-Z]
str {let}+
%%

{str} { int i;
        for(i=0;i<yyleng;i++)
        yytext[i]=tolower(yytext[i]);
        printf("lower case\t %s",yytext);
    }
%%

main()
{
printf("enter input");
yylex();

}

```

OUTPUT:


```

Telnet 135.135.4.26
[lalitha@stud cclab]$ ./a.out
enter inputVASAVI
lower case      vasavi

```

12. LEX program for reversing a string

```

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

let [a-z][A-Z]
str {let}+

%%

{str} { int i=0,j=yyleng-1;
        char rev[50];
        for(i=0;i<=yyleng;i++)
        {

            rev[i]=yytext[j];
            j--;
        }
        puts(rev);
}

%%

main()
{
printf("enter input");
yylex();
}

```

OUTPUT:

```
"rev.l" [New] 26L, 327C written
[lalitha@stud cclab]$ lex rev.l
[lalitha@stud cclab]$ cc lex.yy.c -ll
[lalitha@stud cclab]$ ./a.out
enter inputvasavi
ivasav
```

EX.NO:2 Lexical Analyzer Program using LEX TOOL

Aim : Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.

Algorithm :

```

STEP1 : WHILE there is more input
    InputChar := GetChar
    State := Table[0, InputChar]
STEP2 : WHILE State != Blank
    InputChar := GetChar
    State := Table[State, InputChar]
STEP 3: ENDWHILE
STEP 4: Retract
STEP 5 : Accept
STEP 6: Return token = (Class, Value)
STEP7 :ENDWHILE
    
```

SourceCode:

```

% {
int COMMENT=0;
% }
id [a-z][a-z0-9]*
%%
#.*          {printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int|double|char {printf("\n\t%s is a KEYWORD",yytext);}
if|then|endif  {printf("\n\t%s is a KEYWORD",yytext);}
else if |      {printf("\n\t%s is a KEYWORD",yytext);}
while |        {printf("\n\t%s is a KEYWORD",yytext);}
for |          {printf("\n\t%s is a KEYWORD",yytext);}
do |           {printf("\n\t%s is a KEYWORD",yytext);}
break |        {printf("\n\t%s is a KEYWORD",yytext);}
    
```

```

continue |          {printf("\n\t%s is a KEYWORD",yytext);}
void |              {printf("\n\t%s is a KEYWORD",yytext);}
switch |            {printf("\n\t%s is a KEYWORD",yytext);}
case |              {printf("\n\t%s is a KEYWORD",yytext);}
long |              {printf("\n\t%s is a KEYWORD",yytext);}
struct |            {printf("\n\t%s is a KEYWORD",yytext);}
const |             {printf("\n\t%s is a KEYWORD",yytext);}
typedef |           {printf("\n\t%s is a KEYWORD",yytext);}
return |            {printf("\n\t%s is a KEYWORD",yytext);}
else |              {printf("\n\t%s is a KEYWORD",yytext);}
goto              {printf("\n\t%s is a KEYWORD",yytext);}
"/*"              {COMMENT=1;}
"*/"              {COMMENT=0;}
{id}\(            if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
{id}\(\\[[0-9]*\\])? {if(!COMMENT) printf("\n\tidentifier\t%s",yytext);}
\{                {if(!COMMENT) printf("\n BLOCK BEGINS");ECHO; }
\}                {if(!COMMENT)printf("\n BLOCK ends");ECHO; }
\".*\"            {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[+|-]?[0-9]+      {if(!COMMENT)printf("\n\t%s is a NUMBER",yytext);}
\((              {if(!COMMENT)printf("\n\t");ECHO;printf("\t delim
openparanthesis\n");}
\)                {if(!COMMENT)printf("\n\t");ECHO;printf("\t delim closed
paranthesis");}
\;                {if(!COMMENT)printf("\n\t");ECHO;printf("\t delim semicolon");}
\=                {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT
OPERATOR",yytext);}
<|>              {printf("\n\t %s is relational operator",yytext);}
"+|"-"|"*"|"\/"  {printf("\n %s is an operator\n",yytext);}
"\n" ;
%%
main(int argc ,char **argv)
{
if (argc > 1)

```

```
yyin = fopen(argv[1], "r");
else
yyin = stdin;
yylex ();
printf ("\n");
}
int yywrap()
{
return 0;
}
```

Input:

hello.c

```
#include<stdio.h>
int main()
{
int a=5,b=7;
printf("Sum=%d",a+b);
}
```

Output:

```

"input.txt" 9L, 71C written
[sireesha@stud ~]$ lex lex.l
[sireesha@stud ~]$ cc lex.yy.c -ll
[sireesha@stud ~]$ ./a.out input.txt
~
#include<stdio.h> is PREPROCESSOR DIRECTIVE      2,9      All
-- INSERTint is KEYWORD                          1,2      All

FUNCTION
    main(
    )

BLOCK BEGINS
    int is KEYWORD
a IDENTIFIER
    = is an ASSIGNMENT OPERATOR
    5 IS A NUMBER,
b IDENTIFIER
    = is an ASSIGNMENT OPERATOR
    7 IS A NUMBER;

FUNCTION
    printf(
    "Sum=%d" is a STRING,
a IDENTIFIER+
b IDENTIFIER
    );
  
```

Result:

The program to develop lexical analyzer using LEX tool was executed successfully

Experimental Viva Questions:

1. Distinguish compiler and interpreter
2. What are the phases of a compiler
3. What is meant by sentinels
4. What is the purpose of yylex()
5. Give the significance of lex.yy.c

EX. NO: 3 : Implement “first” of a given context free grammar

Aim : To Implement “first” of a given context free grammar.

Description About Experiment:

The construction of a predictive parser is aided by two functions associated with a grammar G . These functions, $FIRST$ and $FOLLOW$, allow us to fill in the entries of a predictive parsing table for G , whenever possible. Sets of tokens yielded by the $FOLLOW$ function can also be used as synchronizing tokens during panic-mode error recovery.

FIRST is applied to the r.h.s. of a production rule, and tells us all the terminal symbols that can start sentences derived from that r.h.s. It is defined as:

For any terminal symbol a , $FIRST(a) = \{a\}$.

Also, $FIRST(\epsilon) = \{\epsilon\}$.

For any non-terminal A with production rules $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$,

$$FIRST(A) = FIRST(\alpha_1) \cup FIRST(\alpha_2) \cup \dots \cup FIRST(\alpha_n)$$

For any r.h.s. of the form: $\beta_1\beta_2\dots\beta_n$ (where each β_i is a terminal or a non-terminal) we have:

$$FIRST(\beta_1) \text{ is in } FIRST(\beta_1\beta_2\dots\beta_n)$$

if β_1 can derive ϵ , then $FIRST(\beta_2)$ is also in $FIRST(\beta_1\beta_2\dots\beta_n)$

if both β_1 and β_2 can derive ϵ , then $FIRST(\beta_3)$ is also in $FIRST(\beta_1\beta_2\dots\beta_n)$

...

if $\beta_1\beta_2\dots\beta_i$ can derive ϵ , then $FIRST(\beta_{i+1})$ is also in $FIRST(\beta_1\beta_2\dots\beta_n)$

ϵ is in $FIRST(\beta_1\beta_2\dots\beta_n)$ only if ϵ is in $FIRST(\beta_i)$, for all $0 \leq i \leq n$

$FIRST$ can be applied to any r.h.s., and returns a set of terminal symbols.

Thus, if X is the current non-terminal (i.e. leftmost in the current sentential form), and a is the next symbol on the input, then we want to look at the r.h.s. of the production rules for X and choose the one whose $FIRST$ set contains a .

Algorithm:

Source code:

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

```
#include<ctype.h>
```



```

void FIRST(char[],char );
void addToResultSet(char[],char);
int numOfProductions;
char productionSet[10][10];
main()
{
    int i;
    char choice;
    char c;
    char result[20];
    printf("How many number of productions ? :");
    scanf(" %d",&numOfProductions);
    for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T
    {
        printf("Enter productions Number %d : ",i+1);
        scanf(" %s",productionSet[i]);
    }
    do
    {
        printf("\n Find the FIRST of :");
        scanf(" %c",&c);
        FIRST(result,c); //Compute FIRST; Get Answer in 'result' array
        printf("\n FIRST(%c)= { ",c);
        for(i=0;result[i]!='\0';i++)
            printf(" %c ",result[i]);    //Display result
        printf("}\n");
        printf("press 'y' to continue : ");
        scanf(" %c",&choice);
    }
    while(choice=='y' || choice == 'Y');
}
void FIRST(char* Result,char c)
{

```

```

int i,j,k;
char subResult[20];
int foundEpsilon;
subResult[0]='\0';
Result[0]='\0';
if(!(isupper(c)))
{
    addToResultSet(Result,c);
    return ;
}
for(i=0;i<numOfProductions;i++)
{
    if(productionSet[i][0]==c)
    {
        if(productionSet[i][2]=='$') addToResultSet(Result,$');
        else
        {
            j=2;
            while(productionSet[i][j]!='\0')
            {
                foundEpsilon=0;
                FIRST(subResult,productionSet[i][j]);
                for(k=0;subResult[k]!='\0';k++)
                    addToResultSet(Result,subResult[k]);
                for(k=0;subResult[k]!='\0';k++)
                    if(subResult[k]=='$')
                    {
                        foundEpsilon=1;
                        break;
                    }
                if(!foundEpsilon)
                    break;
            }
            j++;
        }
    }
}

```

```
        }  
    }  
}  
}  
return ;  
}  
void addToResultSet(char Result[],char val)  
{  
    int k;  
    for(k=0 ;Result[k]!='\0';k++)  
        if(Result[k]==val)  
            return;  
    Result[k]=val;  
    Result[k+1]='\0';  
}
```

Output:

```

How many number of productions ? :8
Enter productions Number 1 : E=TD
Enter productions Number 2 : D=+TD
Enter productions Number 3 : D=$
Enter productions Number 4 : T=FS
Enter productions Number 5 : S=*FS
Enter productions Number 6 : S=$
Enter productions Number 7 : F=(E)
Enter productions Number 8 : F=a

Find the FIRST of :E
FIRST(E)= { ( a )
press 'y' to continue : Y

Find the FIRST of :D
FIRST(D)= { + $ }
press 'y' to continue : Y

Find the FIRST of :S
FIRST(S)= { * $ }
press 'y' to continue : Y

Find the FIRST of :a
FIRST(a)= { a }
press 'y' to continue :

```

Result:

The program to implement predictive parser was executed successfully

Experimental Viva Questions:

1. What is the output of syntax analysis phase? List different types of parsers
2. What are the rules for FIRST?
3. Find first terms and follow terms for the given grammar?

$$E \rightarrow TE'$$

$$E' \rightarrow +TE'$$

$$E' \rightarrow \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT'$$

$$T' \rightarrow \varepsilon$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

4. Define ambiguous grammar
5. Define context free language. When will you say that two CFGs are equal?

6. Compare parse tree and syntax tree
7. What are the limitations of context free grammar
8. Define handle and explain handle pruning with an example
9. What do you mean by Recursive Descent Parsing?
10. List the issues in top down parsing.

EX.NO: 4 : Implement follow set of a given context free grammar

Aim: To implement follow set of a give context free grammar

Description about experiment:

Algorithm:

Source Code:

```
#include<stdio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],followResult[10];
void follow(char c);
void first(char c);
void addToResult(char);
int main()
{
    int i;
    int choice;
    char c,ch;
    printf("Enter the no.of productions: ");
    scanf("%d", &n);
    printf(" Enter %d productions\nProduction with multiple terms should be give as separate productions \n",
n);
    for(i=0;i<n;i++)
        scanf("%s%c",a[i],&ch);
        // gets(a[i]);
    do
    {
        m=0;
        printf("Find FOLLOW of -->");
        scanf(" %c",&c);
        follow(c);
    }
```

```

printf("FOLLOW(%c) = { ",c);
for(i=0;i<m;i++)
    printf("%c ",followResult[i]);
printf(" }\n");
printf("Do you want to continue(Press 1 to continue....)?");
scanf("%d%c",&choice,&ch);
}
while(choice==1);
}
void follow(char c)
{
    if(a[0][0]==c)addToResult('$');
    for(i=0;i<n;i++)
    {
        for(j=2;j<strlen(a[i]);j++)
        {
            if(a[i][j]==c)
            {
                if(a[i][j+1]!='\0')first(a[i][j+1]);
                if(a[i][j+1]!='\0'&&c!=a[i][0])
                    follow(a[i][0]);
            }
        }
    }
}
void first(char c)
{
    int k;
    if(!(isupper(c)))
        //f[m++]=c;
        addToResult(c);
    for(k=0;k<n;k++)
    {
        if(a[k][0]==c)
        {
            if(a[k][2]=='$') follow(a[i][0]);
            else if(islower(a[k][2]))
                //f[m++]=a[k][2];

```

```

        addToResult(a[k][2]);
    else first(a[k][2]);
    }
}

void addToResult(char c)
{
    int i;
    for( i=0;i<=m;i++)
        if(followResult[i]==c)
            return;
    followResult[m++]=c;
}

```

Output:

```

Enter the no.of productions: 8
Enter 8 productions
Production with multiple terms should be give as separate productions
E=TD
D=+TD
D=$
T=FS
$=*FS
S=$
F=<E>
F=a
Find FOLLOW of -->E
FOLLOW(E) = { $ }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->D
FOLLOW(D) = { }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->T
FOLLOW(T) = { + $ }
Do you want to continue(Press 1 to continue....)?S
Find FOLLOW of -->FOLLOW(S) = { $ }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->F
FOLLOW(F) = { * + $ }
Do you want to continue(Press 1 to continue....)?

```

Result: Implement of follow set of a give context free grammar executed successfully.

EX.NO: 5 : Implementation of LL(1) parser

Aim : To construct a LL(1) parser for an expression.

Algorithm :

Step1 : If $A \rightarrow \alpha$ is a production choice, and there is a derivation $\alpha \Rightarrow *$

$a\beta$, where a is a token, then we add $A \rightarrow \alpha$ to the table entry $M[A, a]$.
 Step 2 : If $A \rightarrow \alpha$ is a production choice, and there are derivations $\alpha \Rightarrow^* \epsilon$
 and $S\$ \Rightarrow^* \beta A a y$, where S is the start symbol and a is a token (or $\$$),
 then we add $A \rightarrow \alpha$ to the table entry $M[A, a]$

Source code:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<stdlib.h>
char first1(char);
void follow1(char);
void null(char);
void parsingtable(void);
int n,j,i,id,c,k;
static int t=0,nt=0;
char prod[15][10], first[15][30], follow[15][30], x,px=-1,terminals[10],nonterminals[10],check,temp;
void main()
{
    char ch;
    //clrscr();
    printf("total number of productions\n");
    scanf("%d",&n);
    printf("\n\nenter production in this format LEFT-->RIGHT\n");
    for(i=0;i<n;i++)
    {
        printf("\nenter production%d \n", i+1);
        scanf("%s",&prod[i]);
    }
    for(i=0;i<n;i++)
    {
        for(j=4;j<strlen(prod[i]);j++)
        {
            check=prod[i][j];

            if(!isupper(check))
            {
                terminals[t++]=check;
            }
        }
    }
    for(k=0;k<strlen(terminals);k++)
    {
        for(i=k+1;i<strlen(terminals);i++)
        {
            if(terminals[k]==terminals[i])
                terminals[i]=terminals[i+1];
        }
    }

    //calculate FIRST
    for(i=0;i<n;i++)
    {
        ch=prod[i][4];
```

```

    j=i;
    x=first1(ch);
    // printf("%c\t",x);
    first[i][0]=x;
}
printf("\n");
//Calculate FOLLOW
for(id=0;id<n;id++)
{
    follow1(prod[id][0]);
    if(id==0)
        strcat(follow[0],"$");
}
//if NULL then change FOLLOW
for(id=0;id<n;id++)
{
    x=prod[id][strlen(prod[id])-1];
    if(isupper(x)&&x!=px)

        {
            px=x;
            null(x);
        }
}
//copying FOLLOW of Non terminals
for(c=0;c<n;c++)
{
    if(prod[c][0]==prod[c-1][strlen(prod[c-1])-1])
        strcpy(follow[c],follow[c-1]);
}
//printing FIRST after clubbing and removing duplicates
for(j=0;j<n-1;j++)
{
    if(prod[j][0]==prod[j+1][0])
    {
        strcat(first[j],first[j+1]);
        strcpy(first[j+1],first[j]);
    }
}
for(j=n-1;j>0;j--)
{
    for(c=j-1;c>=0;c--)
        if(first[j][0]==first[c][0])
            strcpy(first[c],first[j]);
}
for(c=0;c<n;c++)
{
    if(prod[c][0]==prod[c+1][0])
    {
        nonterminals[nt++]=prod[c][0];
        printf("FIRST(%c)= { %s }\n",prod[c][0],first[c]);

        c++;
    }
}

```

```

else
    {
        nonterminals[nt++]=prod[c][0];
        printf("FIRST(%c)= { %s }\n",prod[c][0],first[c]);
    }
}
//Similar printing of FOLLOW
for(c=0;c<n;c++)
{
    if(prod[c][0]==prod[c+1][0])
    {
        printf("FOLLOW(%c)= { %s }\n",prod[c][0],follow[c]);
        c++;
    }
    else
    {
        printf("FOLLOW(%c)= { %s }\n",prod[c][0],follow[c]);
    }
}
    parsingtable();
}
char first1(char ch)
{
    if(isupper(ch)&& j<n)
    {
        for(c=j+1;c<n;c++)
        {
            if(prod[c][0]==ch)
            {
                first1(prod[c][4]);
                break;
            }
        }
    }
    else
        return ch;
}
void follow1(char ch)
{
    for(i=0;i<n;i++)

    {
        for(j=4;j<strlen(prod[i]);j++)

        {
            if(ch==prod[i][j])

```

```

    {
    if(prod[i][j+1]!='\0')

        {
        if(isupper(prod[i][j+1]))

            {
            for(c=0;c<n;c++)
            if(prod[c][0]==prod[i][j+1])

                {
                strcpy(follow[id],first[c]);
                break;
                }
            }
            else
            follow[id][0]=prod[i][j+1];
        }
        else

            {
            strcpy(follow[id],follow[i]);
            }
        }
    }
}
void null(char ch)

{
int q=id,k;
q++;
while(q<n)

{
if(prod[q][4]=='#'&&prod[q][0]==ch)

{
for(k=0;k<n;k++)

{
if(prod[k][0]==prod[id][strlen(prod[id])-2])

{

```

```

        strcat(follow[k],follow[id]);
        break;
    }
}
}
q++;
}
}

#include<firfol.c>
#include<stdlib.h>
//#include<stdio.h>
void parsingtable(void)
{
    int i,j,h,m,r;
    static char table[10][10][10]={ " " };
    *table[0][0]='X';
    for(i=1,j=1;i<=strlen(nonterminals),j<=strlen(terminals);i++,j++)
    {

        *table[i][0]=nonterminals[i-1];
        *table[0][j]=terminals[j-1];
    }
    h=strlen(terminals);
    *table[0][h+1]='$';

    for(i=0;i<=n;i++)
    {
        //check=prod[i][0];
        for(j=1;j<=strlen(nonterminals);j++)
        if(prod[i][0]==*table[j][0])
            r=j;
        for(k=0;k<=strlen(first[i]);k++)
        {

            if(prod[i][4]!='#')

            {

                for(j=1;j<=strlen(terminals)+1;j++)
                {
                    if(isupper(prod[i][4]))
                    {
                        if(first[i][k]==*table[0][j])
                        strcpy(table[r][j],prod[i]);
                    }
                    else
                    {
                        if(first[i][k]==*table[0][j] && prod[i][4]==first[i][k])
                        strcpy(table[r][j],prod[i]);
                    }
                }

            }

        }
        else

```

```

        {
        for(m=0;m<=strlen(follow[i]);m++)
        {
            for(j=1;j<=strlen(terminals)+1;j++)
            { if(follow[i][m]==*table[0][j])
              strcpy(table[r][j],prod[i]);
            }
        }
    }

}

printf("-----");
for( i=0;i<=strlen(nonterminals);i++)
{
    printf("\n\n");
    for(j=0;j<=strlen(terminals)+1;j++)
    {

        printf("%s\t",table[i][j]);

    }

}
printf("\n-----");
}

```

OUTPUT:

```

C:\Windows\system32\cmd.exe
total number of productions
8

enter production in this format LEFT-->RIGHT
enter production1
E-->TA
enter production2
A-->+TA
enter production3
A-->#
enter production4
F-->FB
enter production5
B-->*FB
enter production6
B-->#
enter production7
F--><E>
enter production8
F-->i

FIRST<E>= < >
FIRST<A>= < + # >
FIRST<F>= < < i >
FIRST<B>= < * # >
FIRST<F>= < < i >
FOLLOW<E>= < > $ >
FOLLOW<A>= < > $ >
FOLLOW<F>= < * * * * >
FOLLOW<B>= < * * * * >
FOLLOW<F>= < * >

+      #      *      <      >      i
A-->+TA
A-->#
B-->*FB      F-->FB      F-->FB
Press any key to continue . . .

```

Result:

The program to implement predictive parser was executed successfully

Experimental Viva Questions:

1. What are the rules of first and follows
2. Define left recursion and left factoring
3. List out top down parsing techniques
4. What is the advantage of predictive parser
5. How error handling can be done in syntax analysis

EX.NO: 7 : Implementation of SLR parser

Aim : To construct a SLR parser for an expression.

Algorithm :

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G .

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w ; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36.

□

```

let  $a$  be the first symbol of  $w\$$ ;
while(1) { /* repeat forever */
    let  $s$  be the state on top of the stack;
    if ( ACTION[ $s, a$ ] = shift  $t$  ) {
        push  $t$  onto the stack;
        let  $a$  be the next input symbol;
    } else if ( ACTION[ $s, a$ ] = reduce  $A \rightarrow \beta$  ) {
        pop  $|\beta|$  symbols off the stack;
        let state  $t$  now be on top of the stack;
        push GOTO[ $t, A$ ] onto the stack;
        output the production  $A \rightarrow \beta$ ;
    } else if ( ACTION[ $s, a$ ] = accept ) break; /* parsing is done */
    else call error-recovery routine;
}

```

Source code

```

#include<stdio.h>
#include<string.h>
int action[12][6][2]={
    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
    {{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},
    {{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},
    {{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},
    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
    {{-1,-1},{101,6},{101,6},{-1,-1},{101,6},{101,6}},
    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
    {{-1,-1},{100,6},{-1,-1},{-1,-1},{100,11},{-1,-1}},
    {{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},
    {{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},
    {{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}}
};

```



```
}  
void display1(char p[], int m) // Displays the present Input string  
{  
    int l;  
    printf("\t");  
    for(l=m;p[l]!='\0';l++)  
        printf("%c",p[l]);  
    printf("\n");  
}  
  
void error()  
{  
    printf("\nSyntax Error\n");  
}  
  
void reduce(int p)  
{  
    int len, k, ad;  
    char src, *dest;  
    switch(p)  
    {  
        case 1: dest = "E+T";  
                src='E';  
                break;  
        case 2: dest = "T";  
                src='E';  
                break;  
        case 3: dest = "T*F";  
                src='T';  
                break;  
        case 4: dest = "F";  
                src='T';  
                break;  
        case 5: dest = "(E)";  
                src='F';  
                break;
```

```
        case 6: dest ="i";
                src='F';
                break;
        default: dest="\0";
                src='\0';
    }
    for(k=0; k<strlen(dest); k++)
    {
        pop();
        popb();
    }
    pushb(src);
    switch(src)
    {
        case 'E': ad=0;
                break;

        case 'T': ad=1;
                break;

        case 'F': ad=2;
                break;

        default: ad=-1;
                break;
    }
    push(gotot[TOS()][ad]);
}
int main()
{
    int j, st, ic;
    char ip[20]="\0",an;
    clrscr();
    printf("Enter any string:\n");
    gets(ip);
    push(0);
```

```

printf("STACK\tSYMBOLS\tINPUT\n");
display();
printf("\t%s\n",ip);
for(j=0; ip[j]!='\0';)
{
    st=TOS();
    an=ip[j];
    if(an>='a' && an<='z')
        ic=0;
    else if(an=='+')
        ic=1;
    else if(an=='*')
        ic=2;
    else if(an=='(')
        ic=3;
    else if(an==')')
        ic=4;
    else if(an=='$')
        ic=5;
    else
    {
        error();
        break;
    }
    if(action[st][ic][0]==100)
    {
        pushb(an);
        push(action[st][ic][1]);
        display();
        j++;
        display1(ip,j);
    }

    if(action[st][ic][0]==101)
    {
        reduce(action[st][ic][1]);
    }
}

```

```

        display();
        display1(ip,j);
    }

    if(action[st][ic][0]==102)
    {
        printf("Given String is accepted");
        break;
    }
    if(action[st][ic][0]==-1)
    {
        error();
        break;
    }
}
getch();
return 0;
}

```

OUTPUT:

```

Enter any string:
i+i*i$
STACK  SYMBOLS  INPUT
0      i+i*i$
05     i      +i*i$
03     F      +i*i$
02     T      +i*i$
01     E      +i*i$
016    E+     i*i$
0165   E+i    *i$
0163   E+F    *i$
0169   E+T    *i$
01697  E+T*   i$
016975 E+T*i  $
0169710 E+T*F $
0169   E+T    $
01     E      $
Given String is accepted_

```

Result:

The program to develop SLR parser executed successfully

Experimental Viva Questions:

1. What are the techniques available in bottom up parsing
2. Among SLR,CLR and LALR which is efficient
3. What is operator precedence grammar
4. What are the error recovery techniques in LR parse

INTRODUCTION TO YACC TOOL

Introduction

The UNIX utility YACC (Yet Another Compiler Compiler) parses a stream of token, typically generated by LEX, according to a user-specified grammar. YACC reads the grammar and generate C code for a parser .Grammars written in Backus Naur Form (BNF). BNF grammar used to express Context-Free Languages.

Ex: To parse an expression, do reverse operation (reducing the expression)

This known as bottom-up or shift-reduce parsing.

Structure of a yacc file

A YACC file looks much like a LEX file:

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

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 analyzer. If the code segment is left out, a default main is used which only calls yylex.

3 Definitions section

There are three things that can go in the definitions section:

C code: Any code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

Definitions: The definitions section of a LEX file was concerned with characters; in YACC this is tokens. These token definitions are written to a .h file when YACC compiles this file.

Associativity Rules: These handle associativity and priority of operators.

LEX - YACC interaction:

Conceptually, LEX parses a file of characters and outputs a stream of tokens; YACC accepts a stream of tokens and parses it, performing actions as appropriate. In practice, they are more tightly coupled.

If your LEX program is supplying a tokenizer, the YACC program will repeatedly call the yylex routine. The LEX rules will probably function by calling return every time they have parsed a token. We will now see the way LEX returns information in such a way that YACC can use it for parsing.

The shared header file of return codes

If LEX is to return tokens that YACC will process, they have to agree on what tokens there are.

This is done as follows.

- The YACC file will have token definitions

%token NUMBER in the definitions section.

- When the YACC file is translated with YACC -d, a header file y.tab.h is created that has definitions like #define NUMBER 258. This file can then be included in both the LEX and YACC program.
- The LEX file can then call return NUMBER, and the YACC program can match on this token.

The return codes that are defined from %TOKEN definitions typically start at around 258, so that single characters can simply be returned as their integer value:

```
/* in the LEX program */
[0-9]+ {return NUMBER}
[-+*/] {return *yytext}
/* in the YACC program */
sum: TERMS '+' TERM
```

Return values

In the above, very sketchy example, LEX only returned the information that there was a number, not the actual number. For this we need a further mechanism. In addition to specifying the return code, the LEX parse can return a symbol that is put on top of the stack, so that YACC can access it. This symbol is returned in the variable yylval. By default, this is defined as an int, so the lex program would have

```
extern int llval;
%%
[0-9]+ {llval=atoi(yytext); return NUMBER;}
```

If more than just integers need to be returned, the specifications in the YACC code become more complicated. Suppose we want to return double values, and integer indices in a table.

The following three actions are needed.

1. The possible return values need to be stated:

%union {int ival; double dval;}

2. These types need to be connected to the possible return tokens:

%token <ival> INDEX
%token <dval> NUMBER

3. The types of non-terminals need to be given:

%type <dval> expr


```
%type <dval> mulex
```

```
%type <dval> term
```

The generated .h file will now have

```
#define INDEX 258
```

```
#define NUMBER 259
```

```
typedef union {int ival; double dval;} YYSTYPE;
```

```
extern YYSTYPE yylval;
```

Rules section

The rules section contains the grammar of the language you want to parse. This looks like

```
name1: THING something OTHERTHING {action} | other something THING {other action}
```

```
name2: .....
```

This is the general form of context-free grammars, with a set of actions associated with each matching right-hand side. It is a good convention to keep non-terminals (names that can be expanded further) in lower case and terminals (the symbols that are finally matched) in upper case.

The terminal symbols get matched with return codes from the lex tokenizer. They are typically defines coming from %token definitions in the yacc program or character values;

User code section

The minimal main program is

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

Extensions to more ambitious programs should be self-evident.

In addition to the main program, the code section will usually also contain subroutines, to be used either in the yacc or the lex program

1. Program to recognize strings using grammar ($a^n b^n, n \geq 0$)

Lex Part- (string.l)

```
%{
```

```
    #include "y.tab.h"
```

```
%}
```

```
%%
[aA] { return A; }
[bB] { return B; }
\n { return NL; }
. { return yytext[0]; }
```

Yacc Part (String.y)

```
%token A B NL

%%

stmt : s NL { printf("Valid String\n"); exit(0); }

      ;

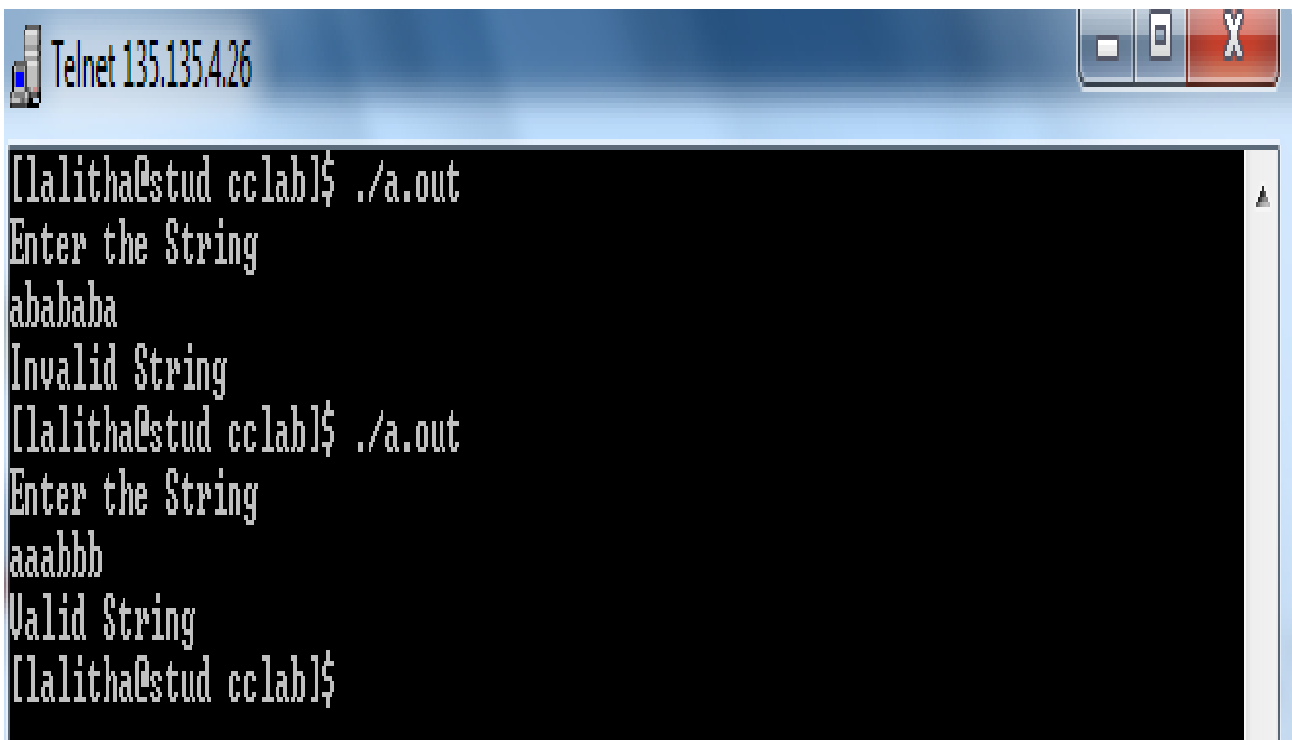
s : A s B
  |
  ;

%%

int yyerror(char *msg)
{
    printf("Invalid String\n");
    exit(0);
}

main ()
{
    printf("Enter the String\n");  yyparse();
}
```

OUTPUT



```

Telnet 135.135.4.26

[lalitha@stud cclab]$ ./a.out
Enter the String
abababa
Invalid String
[lalitha@stud cclab]$ ./a.out
Enter the String
aaabbbb
Valid String
[lalitha@stud cclab]$

```

2. Program to recognize nested IF control statements and display the levels of nesting.

Lex Part

```

% {
    #include "y.tab.h"

% }

%%

"if" { return IF; }

[sS][0-9]* {return S;}

"<" ">" "=" "!=" "<=" ">=" { return RELOP; }

[0-9]+ { return NUMBER; }

[a-zA-Z][a-zA-Z0-9_]* { return ID; }

\n { ; }

. { return yytext[0]; }

%%

```


Yacc Part

```
%token IF RELOP S NUMBER ID

%{
    int count=0;
%}

%%

stmt : if_stmt { printf("No of nested if statements=%d\n",count); exit(0);}
    ;

if_stmt : IF '(' cond ')' if_stmt {count++;}
        | S;
    ;

cond : x RELOP x
    ;

x : ID
  | NUMBER
  ;

%%

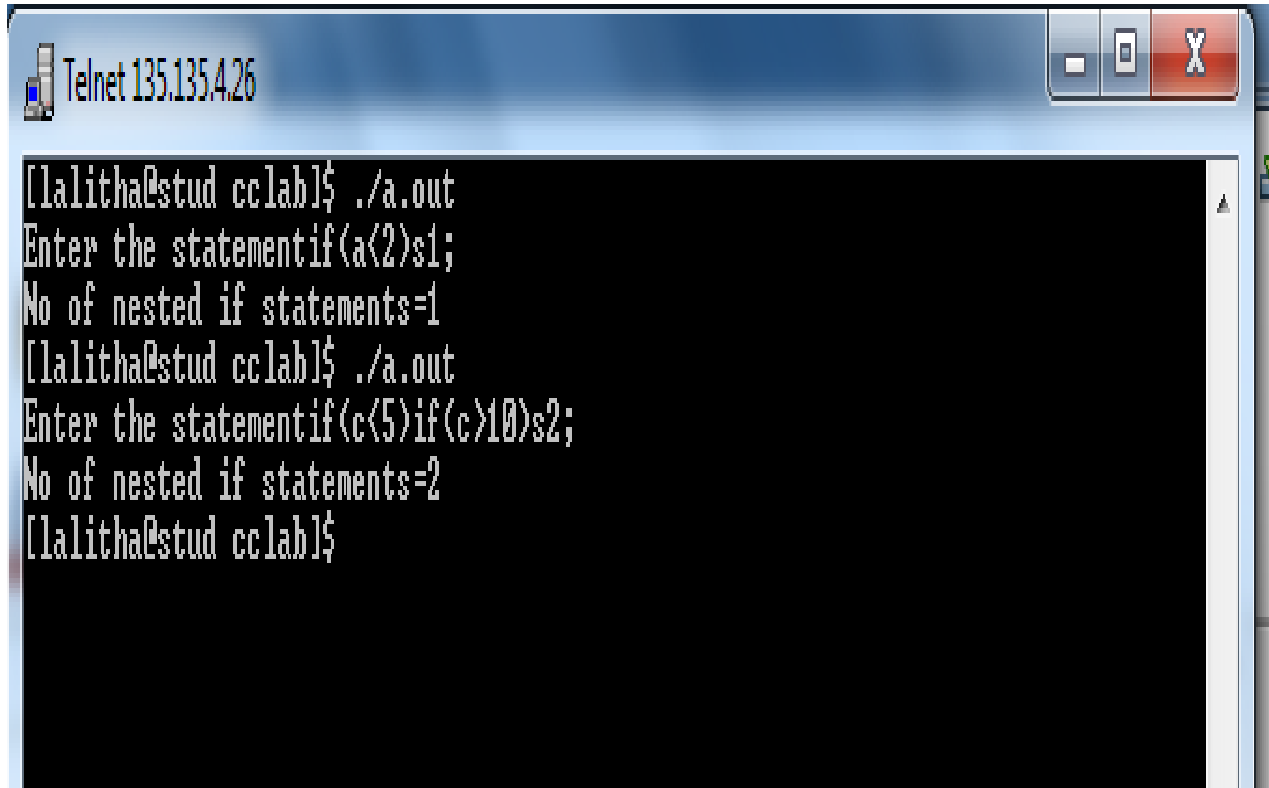
int yyerror(char *msg)
{
    printf("Invalid Expression\n");
    exit(0);
}

main ()
{
    printf("Enter the statement");

    yyparse();
}
```

}

OUTPUT:



```
Telnet 135.135.4.26

[lalitha@stud cclab]$ ./a.out
Enter the statement if(a<2)s1;
No of nested if statements=1
[lalitha@stud cclab]$ ./a.out
Enter the statement if(c<5)if(c>10)s2;
No of nested if statements=2
[lalitha@stud cclab]$
```

3. Program to recognize the grammar ($a^n b$, $n \geq 10$)

Lex Part

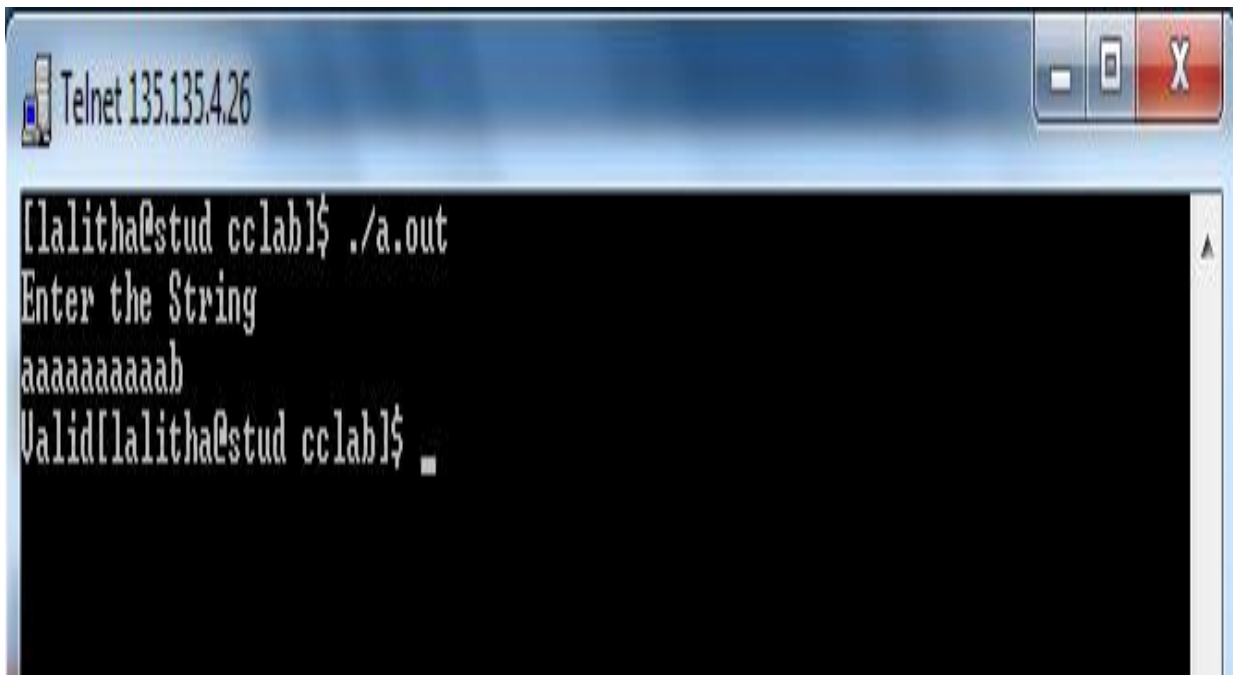
```
% {
    #include "y.tab.h"
% }
```

```
%%
[aA] { return A; }
[bB] { return B; }
\n { return NL ;}
. { return yytext[0]; }
%%
```

Yacc Part

```
%token A B NL
%%
stmt : A A A A A A A A A s B NL
      {
          Printf("Valid"); exit(0);
      }
      ;
s : s A
  |
  ;
int yyerror(char *msg)
{
    printf("Invalid String\n");
    exit(0);
}
main ()
{
    printf("Enter the String\n");
    yyparse();
}
```

OUTPUT:

A screenshot of a Telnet window titled 'Telnet 135.135.4.26'. The window shows a terminal session where a user runs './a.out' at the prompt '[lalitha@stud cclab]\$'. The program prompts 'Enter the String' and the user enters 'aaaaaaaaab'. The program then outputs 'Valid' and returns to the shell prompt '[lalitha@stud cclab]\$'.

Result:

The program to develop syntax analyzer for recognizing grammar was implemented successfully

EX. NO: 7 : Implementation of Desktop Calculator using LEX and YACC

Aim: To implement a program for CALCULATOR using LEX and YACC by considering the following grammar $E \rightarrow E + E | E * E | (E) | \text{DIGIT}$

Source Code:

//calculator.l

```
%{
    #include<stdio.h>
    #include "y.tab.h"
}%

%%

[0-9]+ {yylval.dval = atoi( yytext ); return DIGIT;}
"\n"|. return yytext[0];
%%
```

//calculator.y

```
%{
/*E->E+E|E*E|(E)|DIGIT*/
}%

%union
{
    int dval;
}

%token <dval> DIGIT
%type <dval> expr
%type <dval> expr1

%%

line    :    expr '\n'                {printf("%d\n", $1);}
        ;
```

```

expr  :    expr '+' expr1          {$$ = $1 + $3 ;}
      |    expr '-' expr1         {$$ = $1 - $3 ;}
      |    expr '*' expr1         {$$ = $1 * $3 ;}
      |    expr '/' expr1         {$$ = $1 / $3 ;}
      |    expr1
      ;

```

```

expr1 :    '('expr')'             {$$=$2;}
      |    DIGIT
      ;

```

%%

```
int main()
```

```
{
    yyparse ();
}
```

```
yyerror(char *s)
```

```
{
    printf("%s",s);
}
```

Output:

```
$ lex calculator.l
```

```
$ yacc -d calculator.y
```

```
$ gcc lex.yy.c y.tab.c -ll
```

```
$ ./a.out
```

```
1+2
```

```
3
```

```
$ ./a.out
```

```
4-2
```

```
2
```

```
$ ./a.out
```

```
8+2
```

16

\$./a.out

7/3

Experimental Viva Questions:

1. Define YACC
2. What is the role of a parser
3. What y.tab.c contains
4. What is the purpose of %union,%type
5. How the precedence and associativity is represented in YACC program

EX. NO: 8: Implementation of simple code generator

Aim: To write a C program to implement Simple Code Generator.

Algorithm:

Input: Set of three address code sequence.

Output: Assembly code sequence for three address codes (opd1=opd2, op, opd3).

Method:

- 1- Start
- 2- Get address code sequence.
- 3- Determine current location of 3 using address (for 1st operand).
- 4- If current location not already exist generate move (B,O).
- 5- Update address of A(for 2nd operand).
- 6- If current value of B and () is null,exist.
- 7- If they generate operator () A,3 ADPR.
- 8- Store the move instruction in memory
- 9- Stop.

Source Code:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
#include<graphics.h>
typedef struct
{
    char var[10];
    int alive;
}regist;

regist preg[10];

void substring(char exp[],int st,int end)
```

```

{
int i,j=0;
char dup[10]="";
for(i=st;i<end;i++)
dup[j++]=exp[i];
dup[j]='\0';
strcpy(exp,dup);
}
int getregister(char var[])
{
int i;
for(i=0;i<10;i++)
{
if(preg[i].alive==0)
{
strcpy(preg[i].var,var);
break;
}
}
return(i);
}
void getvar(char exp[],char v[])
{
int i,j=0;
char var[10]="";
for(i=0;exp[i]!='\0';i++)
if(isalpha(exp[i]))
var[j++]=exp[i];
else
break;
strcpy(v,var);
}
void main()

```

```

{
    char basic[10][10],var[10][10],fstr[10],op;
    int i,j,k,reg,vc,flag=0;
    clrscr();
    printf("\nEnter the Three Address Code:\n");
    for(i=0;;i++)
    {
        gets(basic[i]);
        if(strcmp(basic[i],"exit")==0)
            break;
    }
    printf("\nThe Equivalent Assembly Code is:\n");
    for(j=0;j<i;j++)
    {
        getvar(basic[j],var[vc++]);
        strcpy(fstr,var[vc-1]);
        substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
        getvar(basic[j],var[vc++]);
        reg=getregister(var[vc-1]);
        if(preg[reg].alive==0)
        {
            printf("\nMov R%d,%s",reg,var[vc-1]);
            preg[reg].alive=1;
        }
        op=basic[j][strlen(var[vc-1])];
        substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
        getvar(basic[j],var[vc++]);
        switch(op)
        {
            case '+': printf("\nAdd"); break;
            case '-': printf("\nSub"); break;
            case '*': printf("\nMul"); break;
            case '/': printf("\nDiv"); break;

```

```
    }
    flag=1;
    for(k=0;k<=reg;k++)
    {
        if(strcmp(preg[k].var,var[vc-1])==0)
        {
            printf("R%d, R%d",k,reg);
            preg[k].alive=0;
            flag=0;
            break;
        }
    }
    if(flag)
    {
        printf(" %s,R%d",var[vc-1],reg);
        printf("\nMov %s,R%d",fstr,reg);
    }
    strcpy(preg[reg].var,var[vc-3]);
    getch();
}
```

OUTPUT:

```
Enter the Three Address Code:
c=a+b
d=e-f
g=h*j
exit

The Equivalent Assembly Code is:

Mov R0,a
Add b,R0
Mov c,R0
Mov R1,e
Sub f,R1
Mov d,R1
Mov R2,h
Mul j,R2
Mov g,R2
```

Result:

The program for the generation of assembly language code was executed successfully for the given three address code

Experimental Viva Questions:

1. List the three kinds of intermediate code generation
2. How can you generate three address code
3. Define DAG
4. Difference between triples,quadruples,indirect triples
5. What is short circuit code or jumping code

EXP. NO: 09: Code Optimization Techniques

Aim: To write a C program to implement Code Optimization Techniques.

Description About Experiment:

Goals of code optimization:

Remove redundant code without changing the meaning of program.

Objective:

1. Reduce execution speed
2. Reduce code size

1.Common sub-expression elimination:

Remove many occurrences of an expression by its value (constraint: the value should not change across various occurrences).

Original code	Transformed code
a = (b + c)*m;	T1 = b + c; a = T1*m;
x = b + c;	x = T1;
y = (b + c) * z;	y = T1 * z;

2.Dead code elimination:

Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Algorithm:

Input: Set of 'L' values with corresponding 'R' values.

Output: Intermediate code & Optimized code after eliminating common expressions.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct op
{
char l;
char r[20];
}
op[10],pr[10];
void main()
{
int a,i,k,j,n,z=0,m,q;
char *p,*l;
char temp,t;
char *tem;
clrscr();
printf("Enter the Number of Values:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("left: ");
op[i].l=getche();
printf("\tright: ");
scanf("%s",op[i].r);
}
printf("Intermediate Code\n") ;
for(i=0;i<n;i++)
{
```

```

printf("%c=",op[i].l);
printf("%s\n",op[i].r);
}
for(i=0;i<n-1;i++)
{
    temp=op[i].l;
    for(j=0;j<n;j++)
    {
        p=strchr(op[j].r,temp);
        if(p)
        {
            pr[z].l=op[i].l;
            strcpy(pr[z].r,op[i].r);
            z++;
        }
    }
}
pr[z].l=op[n-1].l;
strcpy(pr[z].r,op[n-1].r);
z++;
printf("\nAfter Dead Code Elimination\n");
for(k=0;k<z;k++) {
printf("%c\t=",pr[k].l);
printf("%s\n",pr[k].r);
}
for(m=0;m<z;m++)
{
    tem=pr[m].r;
    for(j=m+1;j<z;j++)
    {
        p=strstr(tem,pr[j].r);
        if(p)
        {

```

```

        t=pr[j].l;
        pr[j].l=pr[m].l;
        for(i=0;i<z;i++)
        {
            l=strchr(pr[i].r,t) ;
            if(l)
            {
                a=l-pr[i].r;
                printf("pos: %d",a);
                pr[i].r[a]=pr[m].l;
            }
        }
    }
}

printf("Eliminate Common Expression\n");
for(i=0;i<z;i++)
{
    printf("%c\t=",pr[i].l);
    printf("%s\n",pr[i].r);
}
for(i=0;i<z;i++)
{
    for(j=i+1;j<z;j++)
    {
        q=strcmp(pr[i].r,pr[j].r);
        if((pr[i].l==pr[j].l)&&!q)
        {
            pr[i].l='\0';
            strcpy(pr[i].r,'\0');
        }
    }
}

```

```

printf("Optimized Code\n");
for(i=0;i<z;i++)
{
if(pr[i].l!='\0')
{
printf("%c=",pr[i].l);
printf("%s\n",pr[i].r);
}
}
getch();
}

```

OUTPUT:

```

left: a right: 9
left: b right: c+d
left: e right: c+d
left: f right: B_e
left: r right: f
Intermediate Code
a=9
b=c+d
e=c+d
f=B_e
r=f

```

```

After Dead Code Elimination
e      =c+d
f      =B_e
r      =f
Eliminate Common Expression
e      =c+d
f      =B_e
r      =f

```

```

Optimized Code
e=c+d
f=B_e
r=f

```

Result:

The program to implement optimization techniques was executed successfully

Experimental Viva Questions:

1. What is the purpose of optimization
2. Give the classification of optimization techniques

3. What is dead code elimination
4. Give the example of common sub expression elimination
5. What are the characteristics of peephole optimization techniques

EXP. NO: 10: Major assignment: Intermediate code generation for subset C language.

Intermediate Code Generation:

Intermediate Code Generation phase is the glue between frontend and backend of the compiler design stages. The final goal of a compiler is to get programs written in a high-level language to run on a computer. This means that, eventually, the program will have to be expressed as machine code which can run on the computer. Many compilers use a medium-level language as a stepping-stone between the high-level language and the very low-level machine code. Such stepping-stone languages are called intermediate code. It provides lower abstraction from source level and maintain some high level information.

Intermediate Code can be represented in many different formats depending whether it is language-specific (e.g. Bytecode for Java) or language-independent (three-address-code). We have used three-address-code to make it independent.

Most common independent intermediate representations are:

1. Postfix notation
2. Three Address Code
3. Syntax tree

Three Address Code

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

General representation: $x = y \text{ op } z$

An address can be a name, constant or temporary.

- Assignments $x = y \text{ op } z$; $x = \text{op } y$.
- Copy $x = y$.
- Unconditional jump $\text{goto } L$.
- Conditional jumps $\text{if } x \text{ relop } y \text{ goto } L$.
- Parameters $\text{param } x$.
- Function call $y = \text{call } p$

Yacc Script

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine. Lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

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

The structure of our Yacc script is given below; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%

Rules section
%%
```

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

In the rules section, each grammar rule defines a symbol in terms of:

1. Other symbols
2. Tokens (or terminal symbols) which come from the lexer.

Each rule can have an associated action, which is executed *after* all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compiletime.

Code:**Updated Lexer Code:**

```
% {
    #include<stdio.h>
    #include<string.h>
    #include<stdlib.h>
    #include"y.tab.h"

    #define ANSI_COLOR_RED          "\x1b[31m"
    #define ANSI_COLOR_GREEN        "\x1b[32m"
    #define ANSI_COLOR_YELLOW       "\x1b[33m"
    #define ANSI_COLOR_BLUE         "\x1b[34m"
    #define ANSI_COLOR_MAGENTA      "\x1b[35m"
    #define ANSI_COLOR_CYAN         "\x1b[36m"
    #define ANSI_COLOR_RESET        "\x1b[0m"

    struct symboltable
    {
        char name[100];
        char class[100]; char
        type[100]; char value[100];
        int nestval; int lineno;
        int length;
        int params_count;
    } ST[1001];

    struct constanttable
    {
        Char
        name[100]; char
        type[100]; int
        length;
    } CT[1001];

    int currnest = 0;
    extern int
```

```

int hash(char *str)
{
    int value = 0;
    for(int i = 0 ; i < strlen(str) ; i++)
    {
        value = 10 * value + (str[i] - 'A');
        value = value % 1001;
        while(value < 0)
            value = value + 1001;
    }
    return value;
}

int lookupST(char *str)
{
    int value = hash(str);
    if(ST[value].length == 0)
    {
        return 0;
    }
    else
    if(strcmp(ST[value].name, str) == 0)
    {
        return value;
    }
    else
    {
        for(int i = value + 1 ; i != value ; i = (i + 1) % 1001)
        {
            if(strcmp(ST[i].name, str) == 0)
            {
                return i;
            }
        }
        return 0;
    }
}

int lookupCT(char *str)
{

```

```

    Int value=hash(str);
    if(CT[value].length == 0)
        return 0;
    else
        if(strcmp(CT[value].name,str)==0)
            return 1;
        else
        {
            for(int i=value + 1; i!=value;i= (i+1)%1001)
            {
                if(strcmp(CT[i].name,str)==0)
                {
                    return 1;
                }
            }
            return 0;
        }
    }
}

void insertSTline(char *str1, int line)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            ST[i].lineno = line;
        }
    }
}

void insertST(char *str1, char *str2)
{
    if(lookupST(str1))
    {
        if(strcmp(ST[lookupST(str1)].class,"Identifier")==0&&strcmp(str2,"
Array Identifier")==0)
        {
            printf("Error use of array\n"); exit(0);
        }
        return;
    }
}

```

```

else
{
    int value=hash(str1);
    if(ST[value].length == 0)
    {
        strcpy(ST[value].name,str1);
        strcpy(ST[value].class,str2);
        ST[value].length=strlen(str1);
        ST[value].nestval= 9999;
        ST[value].params_count=-1;
        insertSTline(str1,yylineno;
        return;
    }

    int pos = 0;

    for (int i= value + 1;i!=value;i=(i+1)%1001)
    {
        if(ST[i].length==0)
        {
            pos = i;
            break;
        }
    }

    strcpy(ST[pos].name,str1);
    strcpy(ST[pos].class,str2);
    ST[pos].length = strlen(str1);
    ST[pos].nestval = 9999;
    ST[pos].params_count = -1;
}
}

void insertSTtype(char *str1,char *str2)
{
    for(int i=0;i<1001;i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            strcpy(ST[i].type,str2);
        }
    }
}

```

```
}
```

```
void insertSTvalue(char *str1, char *str2)
{
    for(int i= 0; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0 && ST[i].nestval == currnest)
        {
            strcpy(ST[i].value,str2);
        }
    }
}

void insertSTnest(char *s, int nest)
{
    if(lookupST(s)&&ST[lookupST(s)].nestval != 9999)
    {
        int pos = 0;
        int value = hash(s);
        for (int i= value+1; i!=value;i= (i+1)%1001)
        {
            if(ST[i].length==0)
            {
                pos = i;
                break;
            }
        }

        strcpy(ST[pos].name,s);
        strcpy(ST[pos].class,"Identifier");
        ST[pos].length=strlen(s);
        ST[pos].nestval=nest;
        ST[pos].params_count=-1;
        ST[pos].lineno= yylineno;
    }
    else
    {
        for(int i= 0;i<1001;i++)
        {
            if(strcmp(ST[i].name,s)==0 )
            {
```

```

        ST[i].nestval = nest;
    }
}
}

void insertSTparamscount(char *s, int count1)
{
    for(int i=0;i<1001;i++)
    {
        if(strcmp(ST[i].name,s)==0 )
        {
            ST[i].params_count = count1;
        }
    }
}

int getSTparamscount(char *s)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,s)==0 )
        {
            return ST[i].params_count;
        }
    }
    return -1;
}

void insertSTF(char *s)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,s)==0 )
        {
            strcpy(ST[i].class,"Function");
            return;
        }
    }
}

```

```

void insertCT(char *str1, char *str2)
{
    if(lookupCT(str1)
        ) return;
    else
    {
        int value=hash(str1);
        if(CT[value].length == 0)
        {
            strcpy(CT[value].name,str1);
            strcpy(CT[value].type,str2);
            CT[value].length=strlen(str1);
            return;
        }

        int pos = 0;

        for (int i=value+1;i!=value;i=(i+1)%1001)
        {
            if(CT[i].length == 0)
            {
                pos = i;
                break;
            }
        }

        strcpy(CT[pos].name,str1);
        strcpy(CT[pos].type,str2);
        CT[pos].length = strlen(str1);
    }
}

void deletedata (int nesting)
{
    for(int i=0;i<1001;i++)
    {
        if(ST[i].nestval == nesting)
        {
            ST[i].nestval = 99999;
        }
    }
}

```

```
}

```

```
int checkscope(char *s)
{
    int flag = 0;
    for(int i=0;i<1000;i++)
    {
        if(strcmp(ST[i].name,s)==0)
        {
            if(ST[i].nestval > currnest)
            {
                flag = 1;
            }
            else
            {
                flag = 0;
                break;
            }
        }
    }

    if(!flag)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

```

```
}

```

```
int check_id_is_func(char *s)
{
    for(int i=0;i<1000;i++)
    {
        if(strcmp(ST[i].name,s)==0)
        {
            if(strcmp(ST[i].class,"Function")==0)
            return 1;
        }
    }
}

```



```

    return 0;
}

int checkarray(char *s)
{
    for(int i=0;i<1000;i++)
    {
        if(strcmp(ST[i].name,s)==0)
        {
            if(strcmp(ST[i].class,"Array Identifier")==0)
            {
                return 0;
            }
        }
    }
    return 1;
}

int duplicate(char *s)
{
    for(int i=0;i < 1000;i++)
    {
        if(strcmp(ST[i].name,s)==0)
        {
            if(ST[i].nestval == currnest)
            {
                return 1;
            }
        }
    }

    return 0;
}

int check_duplicate(char* str)
{
    for(int i=0; i<1001; i++)
    {
        if(strcmp(ST[i].name, str) == 0 &&
            strcmp(ST[i].class, "Function") ==0)
        {

```



```

return SIGNED; }
"sizeof" { insertST(yytext, "Keyword"); return SIZEOF; }
"struct" { strcpy(curtype, yytext);
           insertST(yytext, "Keyword"); return STRUCT; }
"unsigned" { insertST(yytext, "Keyword"); return UNSIGNED; }
"void" { strcpy(curtype, yytext); insertST(yytext, "Keyword"); return VOID; }
"break" { insertST(yytext, "Keyword"); return BREAK; }

{ "++" return increment_operator; }
"--" { return decrement_operator; }
{ "<<" return leftshift_operator; }
">>" { return rightshift_operator; }
"<=" { return less_than_assignment_operator; }
"<" { return less_than_operator; }
">=" { return greater_than_assignment_operator; }
">" { return greater_than_operator; }
"==" { return equality_operator; }
"!=" { return inequality_operator; }
"&&" { return AND_operator; }
"||" { return OR_operator; }
"^" { return caret_operator; }
"*=" { return multiplication_assignment_operator; }
"/=" { return division_assignment_operator; }
"%=" { return modulo_assignment_operator; }
"+=" { return addition_assignment_operator; }
"-=" { return subtraction_assignment_operator; }
"<<=" { return leftshift_assignment_operator; }
">>=" { return rightshift_assignment_operator; }
"&=" { return AND_assignment_operator; }
"^=" { return XOR_assignment_operator; }
"|=" { return OR_assignment_operator; }
"&" { return amp_operator; }
"!" { return exclamation_operator; }
"~" { return tilde_operator; }
"-" { return subtract_operator; }
"+" { return add_operator; }
"*" { return multiplication_operator; }
"/" { return division_operator; }
%" { return modulo_operator; }

```

```

"|" { return pipe_operator; }
\= { return assignment_operator; }

\[^\n]*\[;|,|\)] { strcpy(curval,yytext);
insertCT(yytext,"StringConstant");return string_constant; }
\[A-Z|a-z]\[;|,|\)|:] { strcpy(curval,yytext);
insertCT(yytext,"CharacterConstant"); return character_constant; }
[a-zA-Z]([a-zA-Z|0-9])*\[ { strcpy(curid,yytext);
insertST(yytext, "Array Identifier"); return array_identifier; }
[1-9][0-9]*|0|[;|,|" "\\\|<|>|=\\!\\|&\\+\\-\\*\\|\\%|~\\|\\]|:|\\n\\t\\^]
{ strcpy(curval,yytext); insertCT(yytext, "Number Constant"); yy1val =
atoi(yytext); return integer_constant; }
([0-9]*).([0-9]+)/[;|,|" "\\\|<|>|=\\!\\|&\\+\\-\\*\\|\\%|~\\|\\]|:|\\n\\t\\^]
{ strcpy(curval,yytext);insertCT(yytext,"FloatingConstant");
return float_constant; }
[A-Za-z_][A-Za-z_0-9]* { strcpy(curid,yytext);insertST(curid,"Identifier");
return identifier; }

(.*?) {
    if(yytext[0]=='#')
    {
        printf("Error in Pre-Processor directive at line no.
%d\n",y
yline
);
    }

    else if(yytext[0]=='/')
    {
        printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);
    }
    else if(yytext[0]=='"')
    {
        printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);
    }
    else
    {
        printf("ERROR at line no. %d\n",yylineno);
    }

    printf("%s\n", yytext);
    return 0;
}

```

% %

Parser Code:

```

%{
    #include
    <stdio.h>
    #include
    <string.h>
    #include
    <stdlib.h>

    void
    yyerror(char* s);
    int yylex();
    void ins(); void insV();
    int flag=0;
    #define ANSI_COLOR_RED "\x1b[31m"
    #define ANSI_COLOR_GREEN "\x1b[32m"
    #define ANSI_COLOR_CYAN "\x1b[36m"
    #define ANSI_COLOR_RESET "\x1b[0m"
    extern char curid[20];
    extern char curtype[20];
    extern char curval[20];
    extern int currnest; void
    deletedata (int ); int
    checkscope(char*);
    int check_id_is_func(char *);
    void insertST(char*, char*);
    void insertSTnest(char*, int);
    void insertSTparamscount(char*, int);
    int getSTparamscount(char*);
    int check_duplicate(char*);
    int check_declaration(char*, char *);
    int check_params(char*);
    int duplicate(char *s);
    int checkarray(char*);
    char currfunctype[100];
    char currfunc[100];
    char currfunccall[100];
    void insertSTF(char*);
    char gettype(char*,int);
    char getfirst(char*);
    void push(char *s);
    void codegen();

```



```

void codeassign();
char* itoa(int num, char* str, int base); void reverse(char str[], int length);
void swap(char*,char*);
void label1();
void label2();
void label3();
void label4();
void label5();
void label6();
void genunary();
void codegencon();
void funcgen();
void funcgenend();
void arggen();
void callgen();
int params_count=0;
int call_params_count=0;
int top=0,count=0,ltop=0,lno=0; char temp[3] = "t";
% }

```

```

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
%token ENDIF
%expect 1
%token identifier array_identifier func_identifier
%token integer_constant string_constant float_constant character_constant
%nonassoc ELSE
%right leftshift_assignment_operator rightshift_assignment_operator
%right XOR_assignment_operator OR_assignment_operator
%right AND_assignment_operator modulo_assignment_operator
%right multiplication_assignment_operator division_assignment_operator
%right addition_assignment_operator subtraction_assignment_operator
%right assignment_operator
%left OR_operator
%left AND_operator
%left pipe_operator
%left caret_operator
%left amp_operator

```



```

%left equality_operator inequality_operator
%leftless_than_assignment_operatorless_than_operator
greater_than_assignment_operator greater_than_operator
%left leftshift_operator rightshift_operator
%left add_operator subtract_operator
%left multiplication_operator division_operator modulo_operator

%right SIZEOF
%right tilde_operator exclamation_operator
%left increment_operator decrement_operator

%start program
%%
Program : declaration_list;
declaration_list: declaration D
D: declaration_list| ;
declaration : variable_declaration
            | function_declaration
variable_declaration: type_specifier variable_declaration_list ';'
variable_declaration_list: variable_declaration_list ',' variable_declaration_identifier |
variable_declaration_identifier;
variable_declaration_identifier: identifier
{ if(duplicate(curid)){ printf("Duplicate\n"); exit(0); } insertSTnest(curid,currnest)
; ins(); } vdi | array_identifier
{ if(duplicate(curid)){ printf("Duplicate\n"); exit(0); } insertSTnest(curid,currnest)
; ins(); } vdi;

```

vdi : identifier_array_type | assignment_operator simple_expression ;

identifier_array_type
 : '[' initialization_params
 | ;

initilization_params
 : integer_constant ']' initialization {if(\$\$ < 1) {printf("Wrong
 array size\n"); exit(0);} }
 | ']' string_initilization;

initilization
 : string_initilization
 | array_initialization
 | ;

type_specifier
 : INT | CHAR | FLOAT | DOUBLE
 | LONG long_grammar
 | SHORT short_grammar
 | UNSIGNED unsigned_grammar
 | SIGNED signed_grammar
 | VOID ;

unsigned_grammar: INT | LONG long_grammar | SHORT short_grammar | ;

signed_grammar: INT | LONG long_grammar | SHORT short_grammar | ;

long_grammar : INT | ;

short_grammar: INT | ;

function_declaration:function_declaration_type

function_declaration_param_statement;

function_declaration_type: type_specifier identifier'(
 {strcpy(currfunctype, curtype);
 strcpy(currfunc,curid);
 check_duplicate(curid);
 insertSTF(curid); ins();
 };

```

function_declaration_param_statement: {params_count=0;}params')'
                                     {funcgen();} statement {funcgenend();};
params: parameters_list { insertSTparamscount(currfunc, params_count); }| {
insertSTparamscount(currfunc, params_count); };
parameters_list: type_specifier { check_params(curtype);}
parameters_identifier_list;
parameters_identifier_list: param_identifier
parameters_identifier_list_breakup;
parameters_identifier_list_breakup: ',' parameters_list;
param_identifier: identifier { ins();insertSTnest(curid,1);params_count++; }
param_identifier_breakup;
param_identifier_breakup: '[' ']'| ;
statement : expression_statment | compound_statement
           | conditional_statements | iterative_statements
           | return_statement | break_statement
           | variable_declaration;
compound_statement: { currnest++;} '{ 'statment_list' }'
                  { deletedata(currnest);currnest--;}
statment_list: statement statment_list| ;
expression_statment: expression';'|';' ;

```

```

conditional_statements: IF '(' simple_expression ')'
    { label1(); if ($3 != 1)
      { printf("Condition checking is not of type int\n");
        exit(0); } }
statement { label2(); } conditional_statements_breakup;
conditional_statements_breakup : ELSE statement { label3(); }
    | { label3(); };
iterative_statements: WHILE '(' { label4(); } simple_expression ')'
    { label1(); if ($4 != 1)
      { printf("Condition checking is not
of type int\n"); exit(0); } } statement { label5(); }
    | FOR '(' expression ';' { label4(); } simple_expression ';'
    { label1(); if ($6 != 1) { printf("Condition checking is not of type
int\n"); exit(0); } } expression ')' statement { label5(); } { label4(); }
DO statement WHILE '(' simple_expression ')' { label1(); label5();
if ($6 != 1) { printf("Condition checking is not of type int\n"); exit(0); } } ';';
return_statement: RETURN ';' { if (strcmp(currfunctype, "void"))
    { printf("Returning void of a non-void function\n"); exit(0); } }
    | RETURN expression ';' { if (!strcmp(currfunctype, "void"))
    {
        yyerror("Function is void"); }
    if ((currfunctype[0] == 'i' ||
currfunctype[0] == 'c') && $2 != 1)
    {
        printf("Expression doesn't match
return type
of function\n"); exit(0); }
    };

break_statement
    : BREAK ';' ;

string_initialization
    : assignment_operator string_constant { insV(); } ;

array_initialization

```

```
      : assignment_operator '{' array_int_declarations '}';
```

```
array_int_declarations
```

```
      : integer_constant array_int_declarations_breakup;
```

```
array_int_declarations_breakup
```

```
      : ',' array_int_declarations
      | ;
```

```
expression
```

```
      : mutable assignment_operator {push("=");} expression
```

```
$4==1)      {                                     if($1==1 &&
```

```
      {
      $$=1;
      }
```

```
else
  { $$=-1;
```

```
printf("Type mismatch\n"); exit(0);}

```

```
codeassign();
```

```
}
```

```
| mutable addition_assignment_operator {push("+=");}expression {
                                     if($1==1 &&
```

```
$4==1
```

```
)
```

```
$$=1
```

```
;
```

```
else
```

```
{ $$=-1;
```

```
printf("Type mismatch\n"); exit(0);}

```

```
codeassign();
```

```
}
```

```
| mutable subtraction_assignment_operator {push("-=");} expression
                                     {
```

```
$4==1      if($1==1 &&
```

```
)
```

```
$$=1
```

```
;
```

```
else
```

```
{ $$=-1;
```

```
printf("Type mismatch\n"); exit(0);}
```

```
codeassign();
```

```

        | mutable multiplication_assignment_operator    }
expression                                             {push("*=");}
{
```



```

$4==1)
                                if($1==1 &&
                                $$=1
                                ;
                                else
                                {$$=-1;

printf("Type      mismatch\n");

exit(0);} codeassign();

                                }
                                | mutable division_assignment_operator {push("/=");}expression {
                                if($1==1 &&
$4==1
                                )
                                $$=1
                                ;
                                else
                                {$$=-1;

printf("Type mismatch\n"); exit(0);}

                                }
                                | mutable modulo_assignment_operator {push("%=");}expression {
                                if($1==1 &&
$3==1
                                )
                                $$=1
                                ;
                                else
                                {$$=-1;

printf("Type mismatch\n"); exit(0);}

codeassign();

                                }
                                | mutable increment_operator

{ push("++");if($1 == 1) $$=1; else $$=-1; genunary();}
                                | mutable decrement_operator
{push("--");if($1 == 1) $$=1; else $$=-1;}
                                | simple_expression { if($1 == 1) $$=1; else $$=-1;} ;

simple_expression
                                : simple_expression OR_operator and_expression {push("||");} { if($1
== 1 && $3==1) $$=1; else $$=-1; codegen();}

```

```
| and_expression { if($1 == 1) $$=1; else $$=-1; } ;
```

```
and_expression
```

```
: and_expression AND_operator { push("&&"); }
```

```
unary_relation_expression
```

```
{ if($1 == 1 && $3==1) $$=1; else $$=-1; codegen(); }
```

```
| unary_relation_expression { if($1 == 1) $$=1; else $$=-1; } ;
```

unary_relation_expression

```

      : exclamation_operator {push("!");} unary_relation_expression
    { if($2==1) $$=1; else $$=-1; codegen();}
    | regular_expression { if($1 == 1) $$=1; else $$=-1;} ;

```

regular_expression

```

      : regular_expression relational_operators sum_expression { if($1 ==
1 && $3==1) $$=1; else $$=-1; codegen();}
    | sum_expression { if($1 == 1) $$=1; else $$=-1;} ;

```

relational_operators

```

      :      greaterthan_assignment_operator      {push(">=");}      |
lessthan_assignment_operator {push("<=");} | greaterthan_operator {push(">");}|
lessthan_operator      {push("<");}|      equality_operator      {push("==");}|
inequality_operator {push("!=");} ;

```

sum_expression

```

      : sum_expression sum_operators term { if($1 == 1 && $3==1) $$=1; else
$$=-1; codegen();}
    | term { if($1 == 1) $$=1; else $$=-1;} ;

```

sum_operators

```

      : add_operator {push("+");}
    | subtract_operator {push("-");} ;

```

term

```

      : term MULOP factor { if($1 == 1 && $3==1) $$=1; else $$=-1;
codegen();}
    | factor { if($1 == 1) $$=1; else $$=-1;} ;
}

```

MULOP

```

: multiplication_operator {push("*");}| division_operator
{push("/");} | modulo_operator {push("%");} ;

```

factor

```

      : immutable { if($1 == 1) $$=1; else $$=-1;}
    | mutable { if($1 == 1) $$=1; else $$=-1;} ;

```

```
mutabl  
e      : identifier {  
        push(curid);  
        if(check_id_is_func(curi  
d))  
        { printf("Function name used as Identifier\n");
```

```

exit(8);}

if(!checkscope(curid))
{printf("%s\n",curid);printf("Undeclared\n");exit(0);
} if(!checkarray(curid))
{printf("%s\n",curid);printf("Array ID has no

subscript\n");exit(0) if(gettype(curid,0)=='i' || gettype(curid,1)=='c')
;}
$$ = 1;
else
$$ = -1;
}
| array_identifier
{ if(!checkscope(curid)){printf("%s\n",curid);printf("Undeclared\n");ex
it(0);}} '[' expression ']'
{ if(gettype(curid,0)=='i' || gettype(curid,1)=='
'c')

$$ = 1;
else
$$ = -1;
};

immutable
: '(' expression ')' { if($2==1) $$=1; else $$=-1; }
| call { if($1==1) $$=-1; else $$=1; }
| constant { if($1==1) $$=1; else $$=-1; };

cal
1 : identifier '('{

if(!check_declaration(curid, "Function"))
{ printf("Function not declared");
exit(0);} insertSTF(curid);
strcpy(currfunccall,curid);
if(gettype(curid,0)=='i' || gettype(curid,1)=='c')
{
$$ = 1;
}
else
$$ = -1;
call_params_count=0
;
}

```

```
a      tf"))  
r      {  
g  
u  
m  
e  
n  
t  
s  
,  
)  
,  
{  
i  
f  
(  
s  
t  
r  
c  
m  
p  
(  
c  
u  
r  
r  
f  
u  
n  
c  
c  
a  
l  
l  
,  
"  
p  
r  
i  
n
```

```

if(getSTparamscount(currfuncall)!=call_params_count)
{
    yyerror("Number of arguments in function call
    doesn't match number of parameters");
    exit(8);
}
}
callgen();
};

arguments
    : arguments_list | ;

arguments_list
    : arguments_list ',' exp { call_params_count++; }
    | exp { call_params_count++; };

exp : identifier {arggen(1);} | integer_constant {arggen(2);} |
string_constant
{arggen(3);} | float_constant {arggen(4);} | character_constant {arggen(5);}
;

constant
    : integer_constant { insV(); codegencon(); $$=1; }
    | string_constant { insV(); codegencon(); $$=-1; }
    | float_constant { insV(); codegencon(); }
    | character_constant { insV(); codegencon(); $$=1; };

%%

extern FILE *yyin;
extern int yylineno;
extern char *yytext;
void insertSTtype(char *,char
*); void insertSTvalue(char *,
char *); void incertCT(char *,
char *);
void printST();
void printCT();

```

```
struct stack
{
    char  value[100];
    int  labelvalue;
}s[100],label[100];
```



```
void push(char *x)
{
    strcpy(s[++top].value,x);
}

void swap(char *x, char *y)
{
    char temp = *x;
    *x = *y;
    *y = temp;
}

void reverse(char str[], int length)
{
    int start = 0;
    int end = length - 1;
    while (start < end)
    {
        swap((str+start), (str+end));
        start++;
        end--;
    }
}

char* itoa(int num, char* str, int base)
{
    int i = 0;
    int isNegative = 0;

    if (num == 0)
    {
        str[i++] = '0';
        str[i] = '\0';
        return str;
    }
    if (num < 0 && base == 10)
    {
        isNegative = 1; num = -num;
    }

    while (num != 0)
```

```
{  
    int rem = num % base;
```

```

        str[i++] = (rem > 9)? (rem-10) + 'a' : rem +
        '0'; num = num/base;
    }
    if (isNegative)
        str[i++] = '-';
    str[i] = '\0';

    reverse(str, i);
    return str;
}

void codegen()
{
    strcpy(temp,"t");
    char buffer[100];
    itoa(count,buffer,10);
    strcat(temp,buffer);
    printf("%s = %s %s %s\n",temp,s[top-2].value,s[top-1].value,s[top].value);
    top = top - 2;
    strcpy(s[top].value,temp);
    count++;
}

void codegencon()
{
    strcpy(temp,"t");
    char buffer[100];
    itoa(count,buffer,10);
    strcat(temp,buffer);
    printf("%s = %s\n",temp,curval);
    push(temp);
    count++;
}

int isunary(char *s)
{
    if(strcmp(s, "--")==0 || strcmp(s, "++")==0)
    {
        return 1;
    }
}

```

```
    return 0;  
}
```

```

void genunary()
{
    char    temp1[100],    temp2[100],
    temp3[100];    strcpy(temp1,
s[top].value);    strcpy(temp2,    s[top-
1].value);

    if(isunary(temp1))
    {
        strcpy(temp3,
temp1);
        strcpy(temp1,
temp2);
        strcpy(temp2,
temp3);
    }
    strcpy(temp,
"t");    char
buffer[100];
    itoa(count, buffer, 10);
    strcat(temp,    buffer);
    count++;

    if(strcmp(temp2,"--")==0)
    {
        printf("%s = %s - 1\n", temp, temp1);
        printf("%s = %s\n", temp1, temp);
    }

    if(strcmp(temp2,"++")==0)
    {
        printf("%s = %s + 1\n", temp, temp1);
        printf("%s = %s\n", temp1, temp);
    }

    top = top - 2;
}

void codeassign()
{
    printf("%s    =    %s\n",s[top-
2].value,s[top].value); top = top - 2;
}

```

```
}
```

```
void label1()
```

```
{
```

```
    strcpy(temp,"L");
```

```
    char  buffer[100];
```

```
    itoa(lno,buffer,10
```

```
);
```

```

    strcat(temp,buffer);
    printf("IFnot%s GoTo %s\n",s[top].value,temp);
    label[++ltop].labelvalue = lno++;
}

```

```

void label2()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(lno,buffer,10)
    ;
    strcat(temp,buffer
);
    printf("GoTo      %s\n",temp);
    strcpy(temp,"L");
    itoa(label[ltop].labelvalue,buffer,1
0);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    ltop--;
    label[++ltop].labelvalue=lno
    ++;
}

```

```

void label3()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(label[ltop].labelvalue,buffer,1
0);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    ltop--;

}

```

```

void label4()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(lno,buffer,10);
    strcat(temp,buffer);
    printf("%s:\n",tem
p);
}

```

```

    label[++ltop].labelvalue = lno++;
}

```

```

void label5()

```

```

strcpy(temp,"L");    char
buffer[100];
    itoa(label[ltop-1].labelvalue,buffer,10);
    strcat(temp,buffer);
    printf("GoTo      %s:\n",temp);
    strcpy(temp,"L");
    itoa(label[ltop].labelvalue,buffer,1
0);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    ltop = ltop - 2;

```

```

}

```

```

void funcgen()

```

```

{
    printf("func begin %s\n",currfunc);
}

```

```

void funcgenend()

```

```

{
    printf("func end\n\n");
}

```

```

void arggen(int i)

```

```

{
    if(i==1)
    {
        printf("refparam %s\n", curid);
    }
    else
    {
        printf("refparam %s\n", curval);
    }
}

```

```

void callgen()

```



```

{
    printf("refparam result\n");
    push("result");
    printf("call %s, %d\n",currfunccall,call_params_count);
}

int main(int argc , char **argv)
{
    yyin = fopen(argv[1], "r");
    yyparse();

    if(flag == 0)
    {
        printf(ANSI_COLOR_GREEN "Status: Parsing Complete - Valid"
ANSI_COLOR_RESET "\n");
        printf("%30s" ANSI_COLOR_CYAN "SYMBOL TABLE" ANSI_COLOR_RESET
"\n", " "); printf("%30s %s\n", " ", " ");
        printST();

        printf("\n\n%30s" ANSI_COLOR_CYAN "CONSTANT TABLE"
" "); ANSI_COLOR_RESET "\n",

        printf("%30s %s\n", " ", " ----- ");
    }
    printCT();
}

```

```
}
```

```
void yyerror(char *s)
```

```
{  
    printf(ANSI_COLOR_RED "%d %s %s\n", yylineno, s,  
    yytext); flag=1;  
    printf(ANSI_COLOR_RED "Status: Parsing Failed - Invalid\n"  
    ANSI_COLOR_RESET); exit(7);  
}
```

```
void ins()
```

```
{  
    insertSTtype(curid,curtype);  
}
```

```
void insV()
```

```
{  
    insertSTvalue(curid,curval);  
}
```

```
int yywrap()
```

```
{  
    return 1;  
}
```

Explanation:

The lex code is detecting the tokens from the source code and returning the corresponding token to the parser. In phase 1 we were just printing the token and now we are returning the token so that parser uses it for further computation. We are using the symbol table and constant table of the previous phase only. We added functions like insertSTnest(),insertSTparamscount(),checkscope(),deletedata(),duplicate() etc., in order to check the semantics. In the production rules of the grammar semantic actions are written and these are performed by the functions listed above. Along with semantic actions SDT also included function to generate the 3 address code.

Declaration Section

In this section we have included all the necessary header files,function declaration and flag that was needed in the code.

Between declaration and rules section we have listed all the tokens which are returned by the lexer according to the precedence order. We also declared the operators here according to their associativity and precedence. This ensures the grammar we are giving to the parser is unambiguous as LALR(1) parser cannot work with ambiguous grammar.

Rules Section

In this section production rules for entire C language is written. The grammar productions does the syntax analysis of the source code. When a complete statement with proper syntax is matched by the parser. Along with rules semantic actions associated with the rules are also written and corresponding functions are called to do the necessary actions. Then code generation function was also associated with the production so that we can get the desired 3 address code for the given input program.

C-Program Section

In this section the parser links the extern functions, variables declared in the lexer, external files generated by the lexer etc. The main function takes the input source code file and prints the final symbol table and the 3 address code. Apart from these several functions are also written that generates the 3 address code.

Test Cases:***Without Errors:*****Test Case 1***//Nested if else condition*

```
#include <stdio.h>
void main()
{
    int a,b,c,d;
    if (a<3)
    {
        if(c<d)
        {
            a = 98;
        }
        else
        {
```

Test Case 2

```
#include <stdio.h>
void main()
{
    int a,b,c,d;
    while(a < 10)
    {
        if(a<3)
```

```
        a = 98;
    }
    else
    {
        a = d * b + c;
    }
}
else
{
    a++;
}
```

Test Case 3:

```
#include <stdio.h>

int myfunc(int a,int b)
{
    return a+b;
}

void main()
{
    int a,b,i;

    while(a<3)
    {
        a = a+b;
    }
}
```

With Errors:

Test Case 1:

```
// Undeclared function  
  
#include<stdio.h>  
  
void main()  
{
```

Test Case 2:

```
// Function of type void but still returning #include<stdio.h>  
  
void myfunc(int a)  
{  
    return a;  
}  
  
void main()
```

Test Case 3:

```
// Wrong number of arguments for  
the function #include<stdio.h>  
  
int myfunc(int a)
```

```
myfunc(i,n);
```

Test Case 4:

```
//Invalid condition
checking #include<stdio.h>

void main()
{
    int x,i;
    if("str"
    )
    {
        x=1;
    }
}
```

Test Case 5:

```
// Array of size
0
#include<stdio.h>

void main()
{
    int a[0];
}
```

Implementation:

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom regex.

These were:

- A. The Regex for Identifiers
- B. Multiline comments should be supported
- C. Literals
- D. Error Handling for Incomplete String
- E. Error Handling for Nested Comments

The parser code requires exhaustive token recognition and because of this reason, we utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules.

The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type, value and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error. Along with this semantic actions were also added to each production rule to check if the structure created has some meaning or not. Then we added the function to generate the 3 address code with production so that we can generate the desired intermediate code. In order to generate 3 address code we made use of explicit stack. Whenever we came across an operator, operand or constant we pushed it to stack. Whenever reduction occurred (Since LALR(1) parser is bottom up parser it evaluates SDT when reduction occurs) `codegen()` function generated the 3 address code by creating a new temporary variable and by making use of the entries in the stack, after that it popped those entries from the stack and pushed the temporary variable to the stack so that it gets used in further computation. Similarly functions like labels were used to assign appropriate labels while using conditional statements or iterative statements. All the functions used are described below :

1. `codegen()` : This function is called whenever a reduction of an expression takes place. It creates the temporary variable and displays the desired 3 address code i.e $x = y \text{ op } z$.
2. `codegencon()` : This function is especially written for reductions of expression involving constants since its 3 address code is $x \text{ op } z$.
3. `isunary()` : This function checks if the operator is an unary operator like '++'. If so it returns true else false.
4. `genunary()` : This function is specifically designed to generate 3 address code for unary operations. It makes use of `isunary` function mentioned above. E.g. if $a = i++$ then it converts into $t0 = i + 1, a = t0$.
5. `codeassign()` : This function is specifically designed for assignment operator. It assigns the final temp variable value (after all the evaluation) to the desired variable.
6. `label1()` : It is used while evaluating conditions of loops or if statement. If the condition is not satisfied then it states where to jump to i.e. on which label the control should go.
7. `label2()` : It is used when the statement block pertaining to if statement is over. It tells where the control flow should go once that block is over i.e. it jumps the else statement block.
8. `label3()` : it is used after the whole if else construct is over. It gives label that tells where to jump after the if block is executed.
9. `label4()` : it is used to give labels to starting of loops.

10. `label5()` : it is used after the statement block of the loop. It indicates the label to jump to and also generates the label where the control should go once the loop is terminated.
11. `funcgen()` : it indicates beginning of a function.
12. `funcgenend()` : it indicates the ending of a function.
13. `arggen()` : it displays all the reference parameters that are used in a function call.
14. `callgen()` : it calls the function i.e. displays the appropriate function call according to 3 address code.
15. `Itoa()` : it worked as utility function since we had to name temporary variables and labels it was used to convert int to string and used several functions like reverse , swap to do it.

Results:

We were able to successfully parse the tokens recognized by the flex script for C. The output displays the set of identifiers and constants present in the program with their types, values and line of declaration. Also nesting values changes dynamically as the program ends its made infinite. The parser generates error messages in case of any syntactical errors in the test program or any semantic error. Also we are displaying the 3 address code generated by our yacc script.

Valid Test Cases:**Test Case 1: Operator, Delimiters, Assignments, Nested Conditional Statements**

Output:

```

func begin main
t0 = 3
t1 = a < t0
IF not t1 GoTo L0
t2 = c < d
IF not t2 GoTo L1
t3 = 98
a = t3
GoTo L2
L1:
t4 = d * b
t5 = t4 + c
a = t5
L2:
GoTo L3
L0:
t6 = a + 1
a = t6
L3:
func end

```

Status: Parsing Complete - Valid

SYMBOL TABLE

SYMBOL	CLASS	TYPE	VALUE	LINE NO	NESTING	PARAMS COUNT
a	Identifier	int	3	4	99999	-1
b	Identifier	int		4	99999	-1
c	Identifier	int		4	99999	-1
d	Identifier	int		4	99999	-1
if	Keyword			5	9999	-1
int	Keyword			4	9999	-1
main	Function	void		2	9999	0
else	Keyword			11	9999	-1
void	Keyword			2	9999	-1

CONSTANT TABLE

NAME	TYPE
98	Number Constant
3	Number Constant

Fig. 1

Status: PASS**Test Case 2: LoopStatements****Output:**

```

func begin main
L0:
t0 = 10
t1 = a < t0
IF not t1 GoTo L1
t2 = 3
t3 = a < t2
IF not t3 GoTo L2
t4 = c < d
IF not t4 GoTo L3
t5 = 98
a = t5
GoTo L4
L3:
t6 = d * b
t7 = t6 + c
a = t7
L4:
GoTo L5
L2:
t8 = a + 1
a = t8
L5:
GoTo L0:
L1:
t9 = b + c
a = t9
func end

```

Status: Parsing Complete - Valid

SYMBOL TABLE

SYMBOL	CLASS	TYPE	VALUE	LINE NO	NESTING	PARAMS COUNT
a	Identifier	int	10	4	99999	-1
b	Identifier	int		4	99999	-1
c	Identifier	int		4	99999	-1
d	Identifier	int		4	99999	-1
if	Keyword			7	9999	-1
int	Keyword			4	9999	-1
main	Function	void		2	9999	0
else	Keyword			13	9999	-1
while	Keyword			5	9999	-1
void	Keyword			2	9999	-1

CONSTANT TABLE

NAME	TYPE
10	Number Constant
98	Number Constant
3	Number Constant

Fig. 2

Status : PASS

Test Case 3: Function Call and Loop Constructs

Output:

```
func begin myfunc
t0 = a + b
func end

func begin main
L0:
t1 = 3
t2 = a < t1
IF not t2 GoTo L1
t3 = a + b
a = t3
t4 = 0
i = t4
L2:
t5 = i < b
IF not t5 GoTo L3
t6 = i + 1
i = t6
t7 = b + 1
b = t7
refparam a
refparam b
refparam result
call myfunc, 2
GoTo L2:
L3:
t8 = a + 1
a = t8
GoTo L0:
L1:
func end
```

Status: Parsing Complete - Valid

SYMBOL TABLE							
SYMBOL	CLASS	TYPE	VALUE	LINE NO	NESTING	PARAMS	COUNT
a	Identifier	int		3	99999	-1	
b	Identifier	int		3	99999	-1	
a	Identifier	int	3	10	99999	-1	
b	Identifier	int		10	99999	-1	
i	Identifier	int		10	99999	-1	
for	Keyword			15	9999	-1	
return	Keyword			5	9999	-1	
int	Keyword			3	9999	-1	
main	Function	void		8	9999	0	
myfunc	Function	int		3	9999	2	
while	Keyword			12	9999	-1	
void	Keyword			8	9999	-1	

Fig. 3.

Status : PASS

Invalid Test Cases

Test Case 1: Function not declared Output:

```
===== Running TestCase 2 =====
Function not declared
```

Fig.4.

Status : PASS**Test Case 2: Function of type void Output:**

```
===== Running TestCase 3 =====
6 Function is void ;
Status: Parsing Failed - Invalid
```

Fig. 5

Status : PASS**Test Case 3: Unmatched number of arguments Output:**

```
===== Running TestCase 4 =====
12 Number of arguments in function call doesn't match number of parameters )
Status: Parsing Failed - Invalid
```

Fig. 6

Status : PASS**Test Case 4: Type mismatch Output:**

```
===== Running TestCase 18 =====
Condition checking is not of type int
```

Fig. 7

Status : PASS**Test Case 5: Wrong Array Size Output:**

```
===== Running TestCase 6 =====
Wrong array size
```

Fig. 8

Status : PASS

EX.NO: 11 : Implementation of Recursive Descent Parser for a given context free grammar.

Aim: To implement Recursive Descent Parser for a given CFG

Description about experiment:

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing. This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Algorithm:

- The parsing program consists of a set of procedures, one for each non-terminal.
- Process begins with the procedure for start symbol.
- Start symbol is placed at the root node and on encountering each non-terminal, the procedure concerned is called to expand the non-terminal with its corresponding production.
- Procedure is called recursively until all non-terminals are expanded.
- Successful completion occurs when the scan over entire input string is done. ie., all terminals in the sentence are derived by parse tree.

```
void A()
{
    choose an A-production, A ----> X1 X2 X3... Xk;
    for (i = 1 to k)
        if (Xi is a non-terminal)
            call procedure Xi ();
        else if (Xi equals the current input symbol a)
            advance the input to the next symbol;
        else
            error;
}
```

Source Code:

Given CFG:

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/i$

Eliminate left recursion first, resulting grammar is:

$E \rightarrow TE'$

$E' \rightarrow +TE'/E$

$T \rightarrow FT'$

$T' \rightarrow *FT'/E$

$F \rightarrow (E)/i$

Source Code:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void E(), E1(), T(), T1(), F();
int ip=0;
static char s[10];
int main()
{
char k;
int l;
ip=0;
printf("Enter the string:");
scanf("%s",s);
printf("The string is: %s",s);
E();
if(s[ip]=='$' && strlen(s)>1)
    printf("\nString is accepted.\nLength of the string is %d\n",strlen(s)-1);
else
```



```

    printf("String is not accepted\n");
return 0;
}
void E()
{
    T();
    E1();
return;
}
void E1()
{
    if(s[ip]=='+')
    {
        ip++;
        T();
        E1();
    }
return;
}
void T()
{
    F();
    T1();
return;
}
void T1()
{
    if(s[ip]=='*')
    {
        ip++;
        F();
        T1();
    }
}

```

```

    }
return;
}
void F()
{
    if(s[ip]=='(')
    {
        ip++;
        E();
        if(s[ip]==')')
        {
            ip++;
        }

    else
    {
        printf("String not accepted\n");
        exit(1);
    }

}
else if(s[ip]=='i')
{
    ip++;
}
else
{
    printf("\nString is not accepted");
    exit(1);
}
return;
}

```

Outputs:

>a.exe

Enter the string:i+\$

The string is: i+\$

String is not accepted

>a.exe

Enter the string:(i+\$

The string is: (i+\$

String is not accepted

>a.exe

Enter the string:i+i\$

The string is: i+i\$

String is accepted.

Length of the string is 3

Result:

The program to develop recursive descent parser for a given CFG executed successfully

EXP. NO: 12: Generation of DAG for a given expression

AIM:

To write a C program to construct of DAG(Directed Acyclic Graph)

INTRODUCTION:

The code optimization is required to produce an efficient target code. These are two important issues that used to be considered while applying the techniques for code optimization.

They are:

The semantics equivalences of the source program must not be changed.

The improvement over the program efficiency must be achieved without changing the algorithm.

ALGORITHM:

1. Start the program
2. Include all the header files
3. Check for postfix expression and construct the in order DAG representation
4. Print the output
5. Stop the program

PROGRAM: (TO CONSTRUCT OF DAG(DIRECTED ACYCLIC GRAPH))

```
#include<stdio.h>
main()
{
    struct da
    {
        int ptr,left,right;
        char label;
    }dag[25];
    int ptr,l,j,change,n=0,i=0,state=1,x,y,k;
    char store,*input1,input[25],var;
    for(i=0;i<25;i++)
        dag[i].ptr=NULL;
    dag[i].left=NULL;
    dag[i].right=NULL;
    dag[i].label=NULL;
}
printf("\n\nENTER THE EXPRESSION\n\n");
scanf("%s",input1);
/*EX:((a*b-c))+((b-c)*d)) like this give with paranthesis.limit is 25 char ucan change that*/
for(i=0;i<25;i++)
    input[i]=NULL;
l=strlen(input1);
a:
for(i=0;input1[i]!='\0';i++);
```

```

for(j=i;input1[j]!='(';j--);
for(x=j+1;x<i;x++)
if(isalpha(input1[x]))
input[n++]=input1[x];
else
if(input1[x]!='0')
store=input1[x];
input[n++]=store;
for(x=j;x<=i;x++)
input1[x]='0';
if(input1[0]!='0')goto a;
for(i=0;i<n;i++)
{
dag[i].label=input[i];
dag[i].ptr=i;
if(!isalpha(input[i])&&!isdigit(input[i]))
{
dag[i].right=i-1;
ptr=i;
var=input[i-1];
if(isalpha(var))
ptr=ptr-2;
else
{
ptr=i-1;
b:
if(!isalpha(var)&&!isdigit(var))
{
ptr=dag[ptr].left;
var=input[ptr];
goto b;
}
else
ptr=ptr-1;
}
dag[i].left=ptr;
}
}
printf("\n SYNTAX TREE FOR GIVEN EXPRESSION\n\n");
printf("\n\n PTR \t\t LEFT PTR \t\t RIGHT PTR \t\t LABEL \n\n");
for(i=0;i<n;i++)/* draw the syntax tree for the following output with pointer value*/
printf("\n%d\t%d\t%d\t%d\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i].label);
getch();
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{

```

```

if((dag[i].label==dag[j].label&&dag[i].left==dag[j].left)&&dag[i].right==dag[j].right)
{
for(k=0;k<n;k++)
{
if(dag[k].left==dag[j].ptr)dag[k].left=dag[i].ptr;
if(dag[k].right==dag[j].ptr)dag[k].right=dag[i].ptr;
}
dag[j].ptr=dag[i].ptr;
}
}
}
printf("\n DAG FOR GIVEN EXPRESSION\n\n");
printf("\n\n PTR \t LEFT PTR \t RIGHT PTR \t LABEL \n\n");
for(i=0;i<n;i++)/*draw DAG for the following output with pointer value*/
printf("\n %d
\t\t%d\t\t%d\t\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i].label);
}
OUTPUT:

```

ENTER THE EXPRESSION
 <<a*(b-c)>>+<<(b-c)*d>>
 SYNTAX TREE FOR GIVEN EXPRESSION

PTR		LEFT PTR	RIGHT PTR	LABEL
0	0	0		b
1	0	0		c
2	0	1		-
3	0	0		a
4	2	3		*
5	0	0		b
6	0	0		c
7	5	6		-
8	0	0		d
9	7	8		*
10	4	9		+

Result:

The program for the generation of DAG was executed successfully for the given expression

EXP. NO: 13: Implement symbol table using C Language.

AIM:

To implement symbol table using C Language.

THEORY:

A Symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source

Possible entries in a symbol table:

Name : a string Attribute:

1. Reserved word

2. Variable name

3. Type Name

4. Procedure name

5. Constant name

Data type

Scope information: where it can be used. Storage allocation

INTRODUCTION:

A Symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source

Possible entries in a symbol table:

Name : a string Attribute:

1. Reserved word
2. Variable name
3. Type Name
4. Procedure name
5. Constant name

Data type

Scope information: where it can be used. Storage allocation

ALGORITHM:

1. Open the file pointer for the input file.
2. Read each string and separate the token.
3. For identifiers use 'id','k' for keywords and 's' for symbols.
4. Write each string in the above form to an output file.
5. For each identifier read make an entry in symbol table.
6. For each symbol mention its type, value, relocatable location and size.
7. If there is an assignment expression updates the table.

PROGRAM:

```
#include<conio.h>
#include<string.h>
#include<ctype.h>
#include<process.h>
struct identifier
{
char name[10],type[10],value[10];
int size;
};
void main()
{
FILE *f;
struct identifier id[10];
```



```

char ch, str[19], num[2]="0";
int nid=0, k, flag=0, lastdata, l, oper1=0, idflag=0, s[10]={1,2,4,8}, loc=1000, operand1, operand2, des;
char key[10][10]={"char", "int", "float", "double"};
char iden[10][10], operation;
clrscr();
f=fopen("lexical.txt", "r");
if(f==NULL)

exit(0);
ch=fgetc(f);
while(ch!='{')
ch=fgetc(f);
while(ch==EOF)
{
switch(ch)
{
case '{':
printf("\n OB");
break;
case '}':
printf("\n CB");
break;
case ',':
printf("\n PM1");
break;
case ';':
printf("\n PM2");
break;
case ':':
printf("\n PM3");
break;
case '"':
printf("\n PM4");
break;

```

```

case ' ':
printf(" ");
break;
case '(':
printf("\n OPA");
break;
case ')':
printf("\n CPA");
break;
case '+':
printf("\n OP1");
break;

```

```

case '-':
printf("\n OP2");
break;
case '*':
printf("\n OP3");
break;
case '/':
printf("\n OP4");
break;
case '>':
printf("\n OP5");
break;
case '<':
printf("\n OP6");
break;
case '!':
printf("\n OP7");
break;
case '&':
printf("\n OP8");
break;

```

```

case '=':
printf("\nOP9");
break;
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
case '0':
printf("%c",ch);
break;
default:
if(isalpha(ch))

{
if(idflag!=1)
{
k=0;
while(ch=="")&& &&(ch!=';') (ch!=',') &&(ch!='=') &&(ch!='+') &&(ch!='-') &&(ch!='*')
&&(ch!='/')
{
str[k++]=ch;
ch=fgetc(f);
flag=1;
idflag=1;
}
Str[k++]=ch;
ch=fgetc(f);
flag=1;
idflag=1;

```

```

}
str[k]='\0';
if(strcmp(str,"char")==0)
{
lastdata=0;
printf("\n key1");
}
else if(strcmp(str,"int")==0)
{
lastdata=1;
printf("\n key2");
}
else if(strcmp(str,"float")==0)
{
lastdata=1;
printf("\n key3");
}

else if(strcmp(str,"double")==0)
{
lastdata=3;
printf("\n key4");
}
else
{
idflag=0;
for(i=0;i<nid;i++)
{
if(strcmp(str,iden[i])==0)
break;
}
if(i==nid+1)
printf("Found invalid identifier");
else

```

```

{
printf("%s",id[i].name);
if(ch==';')
{
strcpy(str,id[i].value);
operand2=0;
for(j=0;j<strlen(str);j++)
{
operand2=opearnd2*10+(str[j]-48);
}
if(operation=='+')
operand1=operand1+operand2;
if(operation=='-')
operand1=operand1-operand2;
if(operation=='*')
operand1=operand1*operand2;
if(operation=='/')
operand1=operand1/operand2;
j=0;
while(operand>0)
{
str[j]=(opearnd1%10)+48;
opearnd1=operand1/10;
j++;
}
str[j]='\0';
strrev(str);

strcpy(id[des].value,str);
}
if(ch=='+'||ch=='-'||ch=='/'||ch=='*')
{
operation=ch;
if(oper1==0)

```

```

{
strcpy(str,id[i].value);
operand1=0;
for(j=0;j<strlen(str);j++)
{
operand1=opearnd1*10+(str[j]-48);
}
oper1=1;
}
}
if(ch=='=')
{
ch=fgetc(f);
if(isdigit(ch))
{
k=0;
printf("OP9");
while((ch!=';')&&(ch!='.')&&(ch!=' '))
{
flag=1;
str[k++]=ch;
ch=fgetc(f);
}
str[k]='\0';
strcpy(id[i].value,str);
printf("%s",id[i].value);
}
else
{
des=I;
}}}}
else

{

```

```

flag=1;
k=0;
while((ch!=';')&&(ch!='')&&(ch!=' '))
{
str[k++]=ch;
ch=fgetc(f);
flag=1;
}
str[k]='\0';
strcpy(iden[nid],str);
strcpy(str,"id");
strcat(str,num);
num[0]++;
strcpy(id[nid].name,str);
id[nid].size=s[lastdata];
printf("%s",str);
if(ch==';')
{
idflag=0;
lastdata=4;
}
k=0;
if(ch=='=')
{
printf("OP9");
ch=fgetc(f);
if(isdigit(ch))
{
while((ch!=';')&&(ch!='')&&(ch!=' '))
{
Str[k++]=ch;
ch=fgetc(f);
}
str[k]='\0';

```

```

printf("%s",str);
strcpy(id[nid].value,str);
}
}
nid++;
}
}
break;
}
if(flag!=1)
ch=fgetc(f);
flag=0;
}
printf("\n \t SYMBOL TABLE \n\n NAME  TYPE  SIZE  LOC  VALUE \n");
for(i=0;i<nid;i++)
{
printf("%s\t%s\t%d",iden[i],id[i].type,id[i].size);
printf("\t%d",loc);
loc=loc+id[i].size;
printf("\t%s\n",id[i].value);
}
}

```

OUTPUT:

Lexical.txt:

Main()

```

{
Float x,y,z;
X=10;y=5;z=x-y;
}

```

OB

key3 id0 PM1 id1 PM1 id2 PM2

id0 OP9 10 PM2

id1 OP9 5 PM2

id2 id0 OP2 id1 PM2

CB

SYMBOL TABLE

NAME	TYPE	SIZE	LOC	VALUE
x	float	4	1000	10
y	float	4	1004	5
z	float	4	1008	5

RESULT:

Thus the symbol table program was executed and verified successfully.