PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR

DEPARTMENT OF COMPUTER TECHNOLOGY

LAB MANUAL

Compilers
VII Semester

Academic Year: 2022-23

INDEX

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

1. Vision and Mission of the Institute

Institute Vision

To become one of the India's leading Engineering institutes in both education and research. We are committed to provide quality and state-of-the-art technical education to our students, so that they become technologically superior and in turn contribute for creating a great society.

==========

Institute Mission

We commit ourselves to the pursuit of excellence in technical education and promise to uphold the spirit of professionalism to serve the humanity.

==========

2. Vision and Mission of the Department

Department Vision

To excel in creating outstanding academicians and technocrats for valuable contribution to the society and computer engineering field worldwide as per the ever changing needs.

==========

Department Mission

The Mission of the Computer Technology Department is:

- To ensure technical proficiency to meet industrial needs
- To impart ethical and value based education for social cause
- To encourage the students for carrying out research activities and lifelong learning.

==========

3. Program Educational Objectives (PEOs)

PEO 1: To provide students with a sound foundation in the Mathematics, Basic Sciences and Engineering fundamentals necessary to formulate, solve and analyze engineering problems.

PEO 2 : To develop an ability among the students to understand and apply technical know-how so as to design, analyze, and develop novel software systems/products to provide solutions to the real life problems.

PEO 3 : To prepare students for successful careers through comprehensive and precise education/training/skill development using state-of-the-art infrastructure.

PEO 4 : To inculcate social and ethical values to relate engineering issues to a broader social context.

PEO 5 : To encourage students for pursuing higher education and Life long learning.

4. 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.

5. Program Specific Outcomes (PSOs)

1. An ability to analyze a problem and identify its solution by applying knowledge of computing and fundamental concepts appropriate to the discipline.
2. An ability to design and develop a computerized systems using conventional and modern techniques, tools for solving real world engineering problems of varying complexity.
3. An ability to employ the knowledge of program specific domains for professional growth and pursuing higher education to meet the current industrial needs.

6. Pre-Requisites

Programming Languages: C, Java
Software Required:- FLex, YACC
Text Editor:- Text Pad / Notepad

7. Course Objective

Course Objectives:

1. To make students to understandand demonstrate basics of Compilation Process.
2. To make students to understand and implement the concepts of various phases of Lexical Analysis, Syntax Analysis, Syntax Directed Translation Scheme.
3. To make students to demonstrate Code generation and code Optimization techniques

8. Course Outcomes

CO1	To implement Lexical Analysis Phase.
CO2	To design , analyze and apply various parsing techniques on context free grammar.
CO3	To implement Intermediate code generation phase.
CO4	To implement various code optimization techniques .
CO5	To implement Symbol table.
CO6	To implement Code generation phase.

9. List of Practicals with Course Outcomes

Sr. No	Aim	CO
1	To study phases of compiler	CO1
2	To install a LEX tool and Implement lexical analysis phase.	CO1
3	To demonstrate whether input identifier is number or character with LEX tool.	CO1
4	To demonstrate whether the string is in small case letter, uppercase letter or contains mixed letter with a LEX tool.	CO1
5	To implement a program to find FIRST() for any given Production.	CO2
6	To implement a program to find FOLLOW() for any given Production.	CO2
7	To implement a program to check whether the given grammar is LL(1) or not.	CO2
8	To install YACC tool and implement Calculator using YAAC.	CO2
9	To implement Bottom Up Parser.	CO2
10	To implement a program to eliminate left recursion.	CO2
11	To convert given expression into Three Address Code.	CO3
12	To convert given source code into Three Address Code.	CO3
13	To implement Code optimization.	CO4
14	To create symbol table for any given code.	CO5
15	To develop back end of Compiler.	CO6

10. CO-PO Mapping

Course Outcomes	Program Outcomes											
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CT_401.1	3	3		3	3				2			3
CT_401.2	3	3		3	3				2			3
CT_401.3	3	3		3	3				2			3
CT_401.4	3	3		3	3				2			3
CT_401.5	3	3		3	3				2			3
CT_401.6	3	3		3	3				2			3
Cij	3	3		3	3				2			3

CO-PSO Mapping

Subject	Course Outcomes	Program specific outcomes		
		1	2	3
Compiler	CT_401.1	2	2	3
	CT_401.2	2	2	3
	CT_401.3	2	1	3
	CT_401.4	2	1	3
	CT_401.5	2	2	3
	CT_401.6	2	1	3
	Cij	2	1.5	3

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No: 1

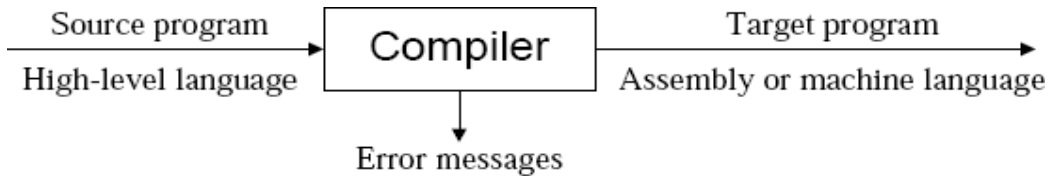
AIM: To study phases of compiler

OBJECTIVE

-To study & understand the concept of compiler.
-To study different phases of compiler

THEORY:

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands. A program for a computer must be build by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error-prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.



Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines. On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

The Translation Process

A compiler performs two major tasks:

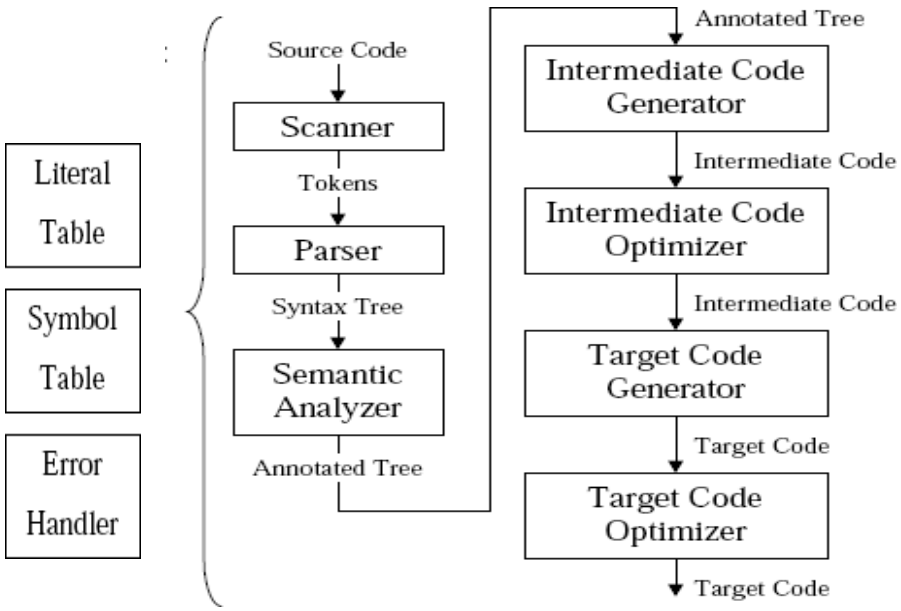
- Analysis of the source program
- Synthesis of the target-language instructions

Phases of a compiler:

- Scanning
- Parsing
- Semantic Analysis
- Intermediate Code Generation
- Intermediate Code Optimizer
- Target Code Generator
- Target Code Optimizer

Three auxiliary components interact with some or all phases:

- Literal Table
- Symbol Table
- Error Handler



Scanner

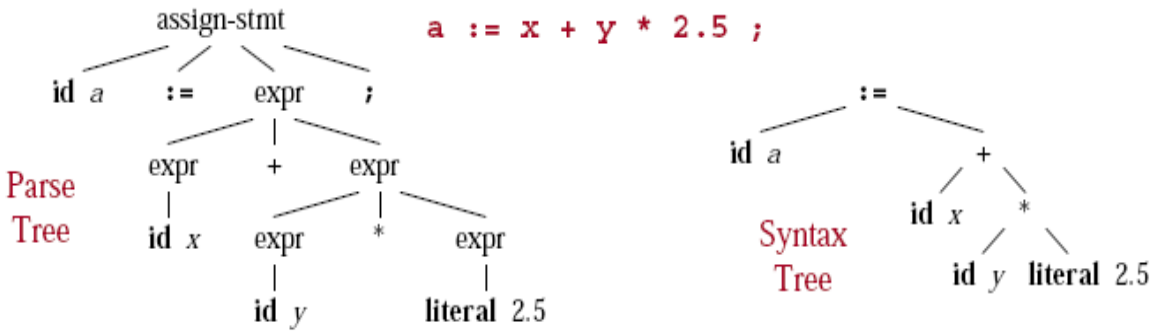
The scanner begins the analysis of the source program by:

- Reading file character by character
- Grouping characters into tokens
- Eliminating unneeded information (comments and white space)
- Entering preliminary information into literal or symbol tables
- Processing compiler directives by setting flags
- Tokens represent basic program entities such as:
- Identifiers, Literals, Reserved Words, Operators, Delimiters, etc.

- Example: **a := x + y * 2.5 ;** is scanned as **a** identifier **y** identifier **:=** assignment operator ***** multiplication operator **x** identifier **2.5** real literal **+** plus operator **;** semicolon

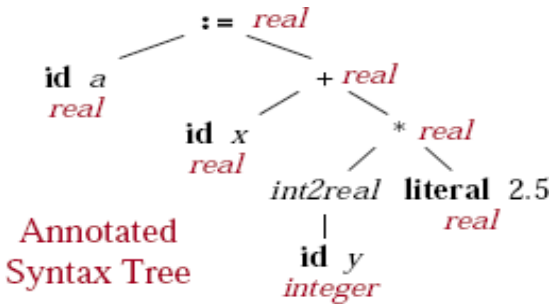
Parser

- Receives tokens from the scanner
- Recognizes the structure of the program as a **parse tree**
- Parse tree is recognized according to a context-free grammar
- Syntax errors are reported if the program is syntactically incorrect
- A parse tree is inefficient to represent the structure of a program
- A **syntax tree** is a more condensed version of the parse tree
- A syntax tree is usually generated as output by the parser



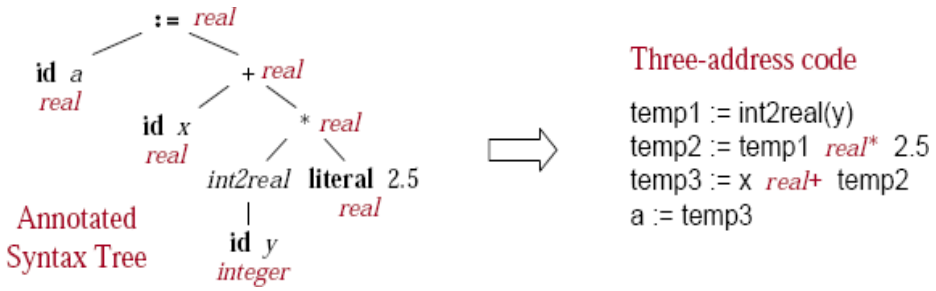
Semantic Analyzer

- The semantics of a program are its **meaning** as opposed to syntax or structure
- The semantics consist of:
- **Runtime semantics** – behavior of program at runtime
- **Static semantics** – checked by the compiler
- Static semantics include:
- Declarations of variables and constants before use
- Calling functions that exist (predefined in a library or defined by the user)
- Passing parameters properly
- Type checking.
- Static semantics are difficult to check by the parser
- The semantic analyzer does the following:
- Checks the static semantics of the language
- Annotates the syntax tree with type information



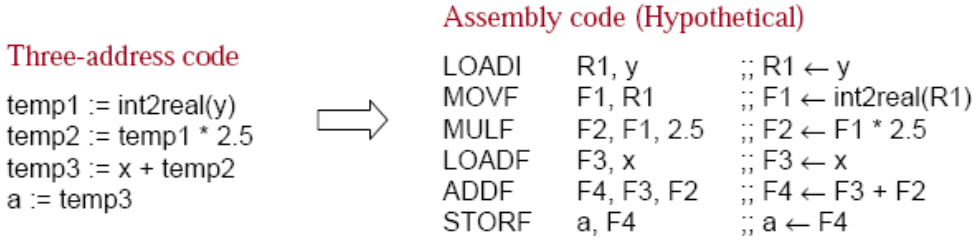
Intermediate Code Generator

- Comes after syntax and semantic analysis
- Separates the compiler front end from its backend
- Intermediate representation should have 2 important properties:
- Should be easy to produce
- Should be easy to translate into the target program
- Intermediate representation can have a variety of forms:
- Three-address code, P-code for an abstract machine, Tree or DAG representation



Code Generator

- Generates code for the target machine, typically:
- Assembly code, or
- Relocatable machine code
- Properties of the target machine become a major factor
- Code generator selects appropriate machine instructions
- Allocates memory locations for variables
- Allocates registers for intermediate computations



Code Improvement

- Code improvement techniques can be applied to:
- Intermediate code – independent of the target machine
- Target code – dependent on the target machine
- Intermediate code improvement include:
- Constant folding
- Elimination of common sub-expressions
- Identification and elimination of unreachable code (called dead code)
- Improving loops
- Improving function calls
- Target code improvement include:
- Allocation and use of registers
- Selection of better (faster) instructions and addressing modes

Conclusion: Thus we have studied and understood the concept of compiler and phases of compiler.

Viva Voce Questions:

1. **What Is A Compiler?**

Answer : 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. The compiler reports to its user the presence of errors in the source program.

2. **What Are The Two Parts Of A Compilation? Explain Briefly.**

Answer :

Analysis and Synthesis are the two parts of compilation.

- The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- The synthesis part constructs the desired target program from the intermediate representation.

3. **Define Symbol Table.**

Answer : Symbol table is a data structure used by the compiler to keep track of semantics of the variables. It stores information about scope and binding information about names.

4. **List The Various Phases Of A Compiler ?**

Answer : The following are the various phases of a compiler:

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate code generator
- Code optimizer
- Code generator

5. **List The Phases That Constitute The Front End Of A Compiler.**

Answer : The front end consists of those phases or parts of phases that depend primarily on the source language and are largely independent of the target machine. These include

- Lexical and Syntactic analysis
- The creation of symbol table
- Semantic analysis
- Generation of intermediate code

A certain amount of code optimization can be done by the front end as well. Also includes error handling that goes along with each of these phases.

6. **Mention The Back-end Phases Of A Compiler.**

Answer : The back end of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language. These include

- Code optimization
- Code generation, along with error handling and symbol- table operations.

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No: 2

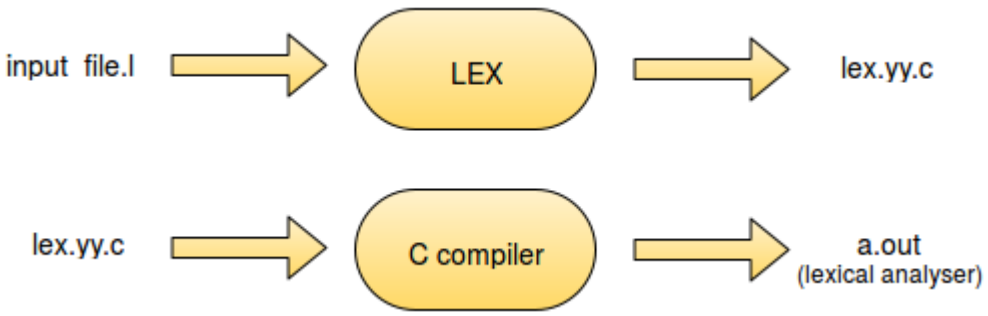
AIM: To install LEX tool and Implement lexical analysis phase.

OBJECTIVE:

A. To install FLEX and execute a sample program.
B. To demonstrate the concept of how to separates the tokens into lexical analysis phase.

THEORY:

A. **FLEX (fast lexical analyzer generator)** is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.
Bison produces parser from the input file provided by the user. The function **yylex()** is automatically generated by the flex when it is provided with a **.l file** and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.



Installing Flex on Ubuntu:

sudo apt-get update

sudo apt-get install flex

Step 1: An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.
Step 2: The C compiler compile lex.yy.c file into an executable file called a.out.
Step 3: The output file a.out take a stream of input characters and produce a stream of tokens.

Program Structure:

In the input file, there are 3 sections:

1. Definition Section: The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in **"%{ %}"** brackets. Anything written in this brackets is copied directly to the file **lex.yy.c**

Syntax:

```
%{

   // Definitions

%}
```

2. Rules Section: The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in **"%% %%"**.

Syntax:

```
%%

pattern  action

%%
```

3. User Code Section: This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Basic Program Structure:

```
%{

// Definitions

%}


%%

Rules

%%

User code section
```

How to run the program:

To run the program, it should be first saved with the extension **.l or .lex**. Run the below commands on terminal in order to run the program file.

Step 1: lex filename.l or lex filename.lex depending on the extension file is saved with
Step 2: gcc lex.yy.c
Step 3: ./a.out
Step 4: Provide the input to program in case it is required

COMPUTING ENVRONMENT:

Platform: ubuntu

Tool: FLEX

Sample Example: Count the number of characters in a string

Write the program in text editer and save the file with .l extention.

For writing program:

Algorithm:

a. Definition Section has one variable which can be accessed inside yylex() and main()
b. Rule Section has three rules, first rule matches with capital letters, second rule matches with any character except newline and third rule does not take input after the enter.
c. Code Section prints the number of capital letter present in the given input.

Input: Any string of Characters and numbers
Output:Number of capital letters in given input

B. To demonstrate the concept of how to separates the tokens into lexical analysis phase.

In a compiler linear analysis is called as lexical analysis or scanning. In this the Streamof chars making to right & group into tokens, The blanks separating the chars of these tokens would normally be eliminated during lexical analysis Here the sequence of char, have a collective meaning.

To know the concept of lexical analyzer and how it works.
Consider the statement
a:=b+c*d
Tokens:
A, b, c, d - identifiers
:= - Assignment Operator
+ - Addition operator
*- Multiplication operator

Algorithm:

Input : LEX specification files for the token Output : Produces the source code for the Lexical Analyzer with the name lex.yy.c and displays the tokens from an input file.

1. Start
2. Open a file in text editor
3. Create a Lex specifications file to accept keywords, identifiers, constants, operators and relational operators in the following format. a) %{ Definition of constant /header files %} b) Regular Expressions %% Transition rules %% c) Auxiliary Procedure (main( ) function)
4. Save file with .l extension e.g. mylex.l
5. Call lex tool on the terminal e.g. [root@localhost]# lex mylex.l. This lex tool will convert ,.l' file into ,.c' language code file i.e., lex.yy.c
6. Compile the file lex.yy.c using C / C++ compiler. e.g. gcc lex.yy.c. After compilation the file lex.yy.c, the output file is in a.out
7. Run the file a.out giving an input(text/file) e.g. ./a.out.
8. Upon processing, the sequence of tokens will be displayed as output.
9. Stop

Expected Output

Input: Any string of Characters, numbers and keywords
Output: The identifier, number and keywords from input.

For example the input string **int a=9**

Expected Output is

int is a small Keyword
a is an identifier
9 is a number

Coclusion: Thus the FLEX tool is installed and the example of Lexical analysis phase is implemented on FLEX tool.

Viva Voce Questions:

1. What is the use of Lex tool?

Answer: Lex is a program designed to generate scanners, also known as tokenizers, which recognize lexical patterns in text. Lex is an acronym that stands for "lexical analyzer generator." It is intended primarily for Unix-based systems.

2. What are the components of Lex?

Answer: A lex program consists of three sections: a section containing definitions, a section containing translations, and a section containing functions.

3. Why is Lex required?

Answer: It is generally used to declare functions, include header files, or define global variables and constants. LEX allows the use of short-hands and extensions to regular expressions for the regular definitions. A regular definition in LEX is of the form : D R where D is the symbol representing the regular expression R.

4. What does the 'Definition section' contain?

Answer: Substitutions code and start states. This section will be copied into lex.yy.c.

5. What does the 'Rule section' contain?

Answer: Defines how to scan and what action to take for ach token

6. What does the last section contain?

Answer: C auxiliary subroutines: any user code and scanning function yylex()

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381,237307, Fax : 07104 – 237681,

Experiment No: 3

AIM: To demonstrate whether input identifier is number or character with LEX tool

THEORY: Lex is a computer program that generates lexical analyzers and was written by Mike Lesk and Eric Schmidt. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lex in the C programming language.

Algorithm:

a. Definition Section has one variable which can be accessed inside yylex() and main()

b. Rule Section has two rules, first rule matches with number(0-9), second rule matches with any character(a-z)(A-Z).

c. Code Section prints the given input is number or Character .

COMPUTING ENVRONMENT:

Platform: ubuntu
Tool: FLEX

Expected Output

Input: Any string of Characters,numbers
Output: The numbers , characters are displayed

For example the input string **aaB23@D**

Expected Output is

a is a small character
a is a small character
B is a capital character
2 ia a number
3 is a number
@ is not a number and character
D is a capital character

Conclusion: Thus the lex program to identify whether the input identifier is number or character is implemented.

Viva Voce Questions:

1. What is the rule to define nembers in LEX?

Answer: To define numbers in LEX [0-9] is used.

2. What is the rule to define Characters in LEX?

Answer: To define characters i.e small and capital [a-z] and [A-Z] are used.

3. What is the rule to define no number and no charaters.

Answer: ^[0-9], ^[a-z], ^[A-Z]

4. What is the difference between yylex() and scanf().

Answer: yylex() is used to accept input and call parser, but scanf() for only accepting data.

5. How is the instruction in the first section used?

Answer: It is copied as it is into lex.yy.c

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381,237307, Fax : 07104 – 237681,

Experiment No: 4

AIM: To demonstrate whether the string is in small case letter, uppercase letter or contains mixed letter with a LEX tool.

THEORY: The key to solving this problem lies in the ASCII value of a character. It is the simplest way to find out about a character. This problem is solved with the help of the following detail:

- - Capital letter Alphabets (A-Z) lie in the range 65-91 of the ASCII value
- - Small letter Alphabets (a-z) lie in the range 97-122 of the ASCII value
- - Any other ASCII value is a non-alphabetic character.

Algorithm:

```
In Rule section define [a-z]+
{printf( "\n String contains only lower case letters " );}
[A-Z]+ { printf( "\n String contains only upper case letters " );}

[a-zA-Z]+ {printf( "\n String contains both lower & upper case letters" );}
```

COMPUTING ENVRONMENT:

Platform: ubuntu
Tool: FLEX

Expacted Output:

Input: ch = 'AA' **Output:** String contains only uppercase letters

Input: ch = 'aa' **Output:** String contains only lowercase letters

Input: ch = 'bD' **Output:** String contains both lowe & uppercase letters

Conclusion: Thus the lex program to identify whether the string is in small case letter, uppercase letter or contains mixed letter

Viva Voce Questions:

1. What does the lex prg contain?
   Answer: A specification of a lexical analyzer is prepared by creating a program lex.l in the lex language. Then this lex is run thru the lex compiler to produce a c prg- lex.yy.c.
2. What is Token?

Answer: A token is the smallest unit used in a C program.

3. What is lexeme, Pattern?

Answer: Lexeme is a sequence of characters in the source code that are matched by given predefined language rules for every lexeme to be specified as a valid token.
Pattern specifies a set of rules that a scanner follows to create a token.

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No:5

AIM: To implement a program to find FIRST() for any given Production.

THEORY:

FIRST (α) is defined as the collection of terminal symbols which are the first letters of strings derived from α.

FIRST (α) = {α | α →* α β for some string β }

If the compiler would have come to know in advance, that what is the "first character of the string produced when a production rule is applied", and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

Algorithm:

If X is Grammar Symbol, then First (X) will be –

- If X is a terminal symbol, then FIRST(X) = {X}
- If X → ε, then FIRST(X) = { ε }
- If X is non-terminal & X → a α, then FIRST (X) = {a}
- If X → Y1, Y2, Y3, then FIRST (X) will be

(a) If Y is terminal, then
FIRST (X) = FIRST (Y1, Y2, Y3) = {Y1}

(b) If Y1 is Non-terminal and
If Y1 does not derive to an empty string i.e., If FIRST (Y1) does not contain ε then,
FIRST (X) = FIRST (Y1, Y2, Y3) = FIRST(Y1)

(c) If FIRST (Y1) contains ε, then.
FIRST (X) = FIRST (Y1, Y2, Y3) = FIRST(Y1) – { ε } ∪ FIRST(Y2, Y3)

Similarly, FIRST (Y2, Y3) = {Y2}, If Y2 is terminal otherwise if Y2 is Non-terminal then
FIRST (Y2, Y3) = FIRST (Y2), if FIRST (Y2) does not contain ε.
If FIRST (Y2) contain ε, then
FIRST (Y2, Y3) = FIRST (Y2) – { ε } ∪ FIRST (Y3)

Similarly, this method will be repeated for further Grammar symbols, i.e., for Y4, Y5, Y6 …. YK

COMPUTING ENVRONMENT

Platform: ubuntu
Programming Language: C / C++ / Java

Expacted Output:

OUTPUT:

```
Enter the no. of productions :6
Enter the productions :
S/aBDh
B/cC
C/bC/@
E/g/@
D/E/F
F/f/@
First(S) : [ a ].
First(B) : [ c ].
First(C) : [ b,@ ].
First(E) : [ g,@ ].
First(D) : [ g,@,f ].
First(F) : [ f,@ ].

Process returned 6 (0x6)   execution time : 110.862 s
Press any key to continue.
```


Conclusion: Thus the program to find FIRST() is implemented.

Viva Voce Questions:

1. What is FIRST()?

Answer: **FIRST ()**– It is a function that gives the set of terminals that begin the strings derived from the production rule.

2. Why FIRST() is CALCULATED.?

Answer: If the compiler would have come to know in advance, that what is the "first character of the string produced when a production rule is applied", and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No:6

AIM: To implement a program to find follow() for any given Production.
THEORY:Follow(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form.

Algorithm:
Computation of FOLLOW

Follow (A) is defined as the collection of terminal symbols that occur directly to the right of A.

$$FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta \text{ where } \alpha, \beta \text{ can be any strings}\}$$

If S is the start symbol, FOLLOW (S) ={$}
If production is of form $A \rightarrow \alpha B \beta$, $\beta \neq \varepsilon$.
(a) If FIRST (β) does not contain ε then, FOLLOW (B) = {FIRST (β)}

Or

(b) If FIRST (β) contains ε (i. e. , $\beta \Rightarrow^* \varepsilon$), then

$$FOLLOW(B) = FIRST(\beta) - \{\varepsilon\} \cup FOLLOW(A)$$

∵ when β derives ε, then terminal after A will follow B.

- If production is of form $A \rightarrow \alpha B$, then Follow (B) ={FOLLOW (A)}.

COMPUTING ENVRONMENT
Platform: ubuntu
Programming Language: C / C++ / Java

Expacted 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
S=*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....)?
```

Conclusion: Thus the program to find FOLLOW() is implemented.

Viva Voce Questions:

1. What is Follow()?

Answer: **Follow (A) is defined as the collection of terminal symbols that occur directly to the right of A.**

2. Why FOLLOW() is calculated?

Answer:FOLLOW can make a Non-terminal vanish out if needed to generate the string from the parse tree. FOLLOW sets for a given grammar so that the parser can properly apply the needed rule at the correct position.

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No:7

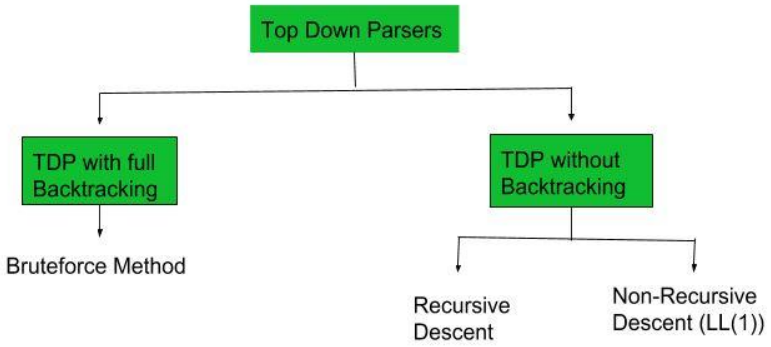
AIM: To implement a program to check whether the given grammar is LL(1) or not.

THEORY:

A top-down parser builds the parse tree from the top down, starting with the start non-terminal. There are two types of Top-Down Parsers:

1. Top-Down Parser with Backtracking
2. Top-Down Parsers without Backtracking

Top-Down Parsers without backtracking can further be divided into two parts:



Here the 1st **L** represents that the scanning of the Input will be done from Left to Right manner and the second **L** shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the **1** represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Essential conditions to check first are as follows:

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

These conditions are necessary but not sufficient for proving a LL(1) parser.

ALGORITHM:

Step 1: First check all the essential conditions mentioned above and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

1. First(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
2. Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A -> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A -> α in the table.
2. If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A -> α in the table.
3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A -> α in the table for the $.

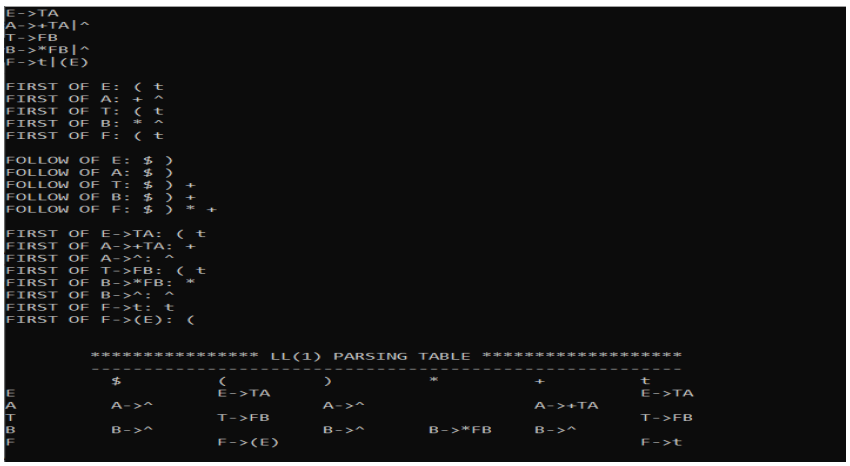
To construct the parsing table, we have two functions:

In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

COMPUTING ENVRONMENT

Platform: ubuntu
Programming Language: C / C++ / Java

Expacted Output:



Conclusion: Thus the program to check whether the given grammar is LL(1) or not.

Viva Voce Questions:

1. **Define A Context Free Grammar.**

Answer :

A context free grammar G is a collection of the following

- V is a set of non terminals
- T is a set of terminals
- S is a start symbol
- P is a set of production rules

G can be represented as G = (V,T,S,P)
Production rules are given in the following form
Non terminal → (V U T)*

2. **Briefly Explain The Concept Of Derivation.**

Answer :

Derivation from S means generation of string w from S. For constructing derivation two things are important.
i) Choice of non terminal from several others.
ii) Choice of rule from production rules for corresponding non terminal.
Instead of choosing the arbitrary non terminal one can choose
i) either leftmost derivation – leftmost non terminal in a sentinel form.
ii) or rightmost derivation – rightmost non terminal in a sentinel form.

3. **Define Ambiguous Grammar.**

Answer :

A grammar G is said to be ambiguous if it generates more than one parse tree for some sentence of language L(G). i.e. both leftmost and rightmost derivations are same for the given sentence.

4. What is LL(1) parser?

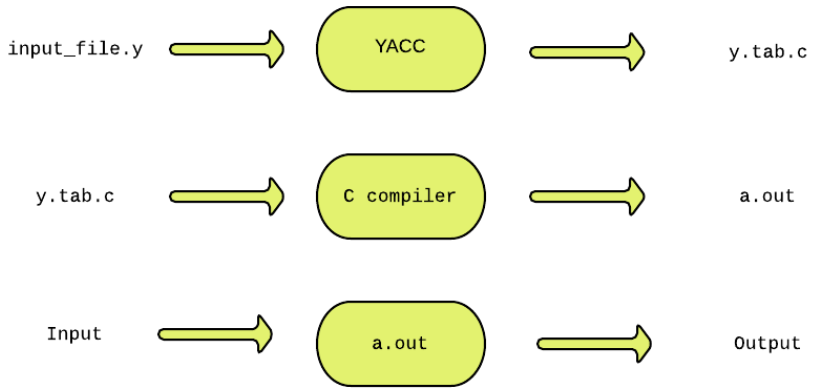
Answer: A top-down parser that uses a one-token lookahead is called an LL(1) parser. The first L indicates that the input is read from left to right. The second L says that it produces a left-to-right derivation. And the 1 says that it uses one lookahead token. (Some parsers look ahead at the next 2 tokens, or even more than that.

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No:8

AIM: To install YACC tool and implement Calculator using YAAC.

THEORY: A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.
YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.



Input File:
YACC input file is divided into three parts.
/* definitions */
....
%%
/* rules */
....
%%
/* auxiliary routines */
....

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:

%token NUMBER
%token ID

- Yacc automatically assigns numbers for tokens, but it can be overridden by

%token NUMBER 621

- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should not overlap ASCII codes.

- The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.
- It can also include the specification of the starting symbol in the grammar:

```
%start nonterminal
```

- **Input File: Rule Part:**
- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:

```
#include "lex.yy.c"
```

- YACC input file generally finishes with:

```
.y
```

- **Output Files:**

The output of YACC is a file named **y.tab.c**

If it contains the **main()** definition, it must be compiled to be executable.

Otherwise, the code can be an external function definition for the function **int yyparse()**

If called with the **-d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).

If called with the **-v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

For Compiling YACC Program:

1. Write lex program in a file file.l and yacc in a file file.y
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex file.l
4. type yacc file.y
5. type cc lex.yy.c y.tab.h -ll
6. type ./a.out

To implement Calculator using YAAC.

Algorithm:

1. For Lexical Analyzer:

In Rule section define [0-9]+ as number.

2. For Parser

In definition Section define Operators +, -, *, /

Define Arithmetic Expression

```
E:E'+'E {$$=$1+$3;}
  |E'-'E {$$=$1-$3;}
  |E'*'E {$$=$1*$3;}
  |E'/'E {$$=$1/$3;}
  |E'%'E {$$=$1%$3;}
  |'('E')' {$$=$2;}
  | NUMBER
```

Expected Output:

acc (for "yet another compiler compiler.") is the standard parser generator for the Unix operating system. An open source program, yacc generates code for the parser in the C

programming language. The acronym is usually rendered in lowercase but is occasionally seen as YACC or Yacc.

Examples:

Input: 4+5 **Output:** Result=9

Entered arithmetic expression is Valid

Input: 10-5**Output:** Result=5

Entered arithmetic expression is Valid

Input: 10+5-**Output:**

Entered arithmetic expression is Invalid

Input: 10/5**Output:** Result=2

Entered arithmetic expression is Valid

Input: (2+5)*3**Output:** Result=21

Entered arithmetic expression is Valid

Input: (2*4)+**Output:**

Entered arithmetic expression is Invalid

Conclusion: Thus the YACC tool is installed and basic Calculator is implemented using YAAC.

Viva Voce Questions:

1. What is YACC?

Answer:YACC is an automatic tool for generating the parser program. YACC stands for Yet Another Compiler Compiler which is basically the utility available from UNIX. Basically YACC is LALR parser generator.

2. Is Yacc a compler!
Answer:No, Yacc is available as a command on the unix system and has been used to implement of hundred of compiler.

3. Explain construction of yacc prg.
Answer: A prg containing yacc specification is provided to yacc. This provides y.tab.c as a c prg. This is compiled using c compiler to get exe ie a.out.

4. what is the difference between Lex and Yacc
Answer:lex prg to lex.yy.c to scanner to parser to exe
yac prg to y.tab.c to parser to exe

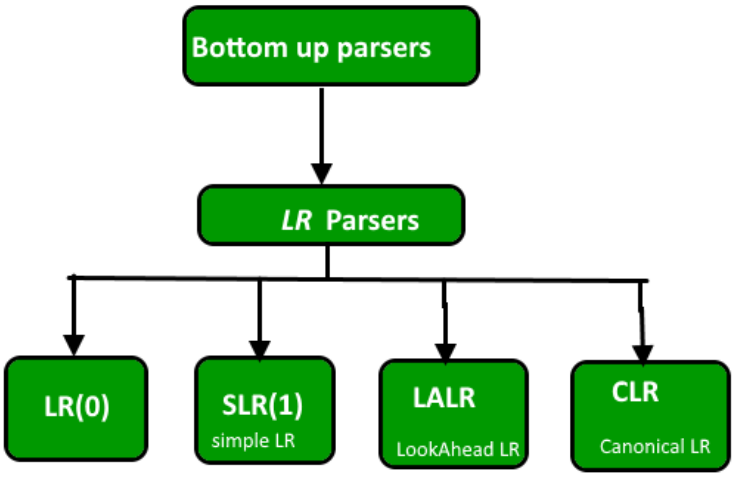
PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No:9

AIM: To implement Bottom Up Parser.

THEORY: Build the parse tree from leaves to root. Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.

Classification of bottom up parsers



A general shift reduce parsing is LR parsing. The L stands for scanning the input from left to right and R stands for constructing a rightmost derivation in reverse.

The bottom-up name comes from the concept of a parse tree, in which the most detailed parts are at the bottom of the (upside-down) tree, and larger structures composed from them are in successively higher layers, until at the top or "root" of the tree a single unit describes the entire input stream. A bottom-up parse discovers and processes that tree starting from the bottom left end, and incrementally works its way upwards and rightwards.[1] A parser may act on the structure hierarchy's low, mid, and highest levels without ever creating an actual data tree; the tree is then merely implicit in the parser's actions.

Here we describe a skeleton algorithm of an LR parser:

```
1. token = next_token()
2. repeat forever
   s = top of stack
3. if action[s, token] = “shift si” then
4. PUSH token
5. PUSH si
      token = next_token()
6.  else if action[s, token] = “reduce A::= β “ then
7. POP 2 * |β| symbols
      s = top of stack
8. PUSH A
9. PUSH goto[s,A]
10.  else if action[s, token] = “accept” then
11.  return
12.    else
      error()
```

COMPUTING ENVRONMENT

Platform: ubuntu
Programming Language: C / C++ / Java

Expected OUTPUT

```
  Enter Number of productions:4
  Enter productions:
E->E+E
E->E*E
E->(E)
E->a
  Enter Input:(a+a)*a
  Stack     Input     Action
(             a+a)*a   Shifted
(a            +a)*a    Shifted
(E            +a)*a    Reduced
(E+           a)*a     Shifted
(E+a          )*a      Shifted
(E+E          )*a      Reduced
(E             )*a       Reduced
(E)           *a        Shifted
E               *a        Reduced
E*              a         Shifted
E*a                       Shifted
E*E                       Reduced
E                           Reduced
  String Accepted
```

Conclusion: Thus the Bottom up parser(Shift Redused Parser) is implemented.

Viva Voce Questions:

1. What is used in bottom-up parsing?

Answer: Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.

2. Why Bottom-up parser is more powerful?

Answer:The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique.

3. What is the role of parser?

Answer: The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

4. What is most common type bottom-up parser?

Answer: Shift-reduce parsing is the most commonly used and the most powerful of the bottom-up techniques.

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No:10

AIM: To implement a program to eliminate left recursion.

THEORY:Left recursion

A grammar in the form G = (V, T, S, P) is said to be in left recursive form if it has the production rules of the form A → A α |β. In the production rule above, the variable in the left side occurs at the first position on the right side production, due to which the left recursion occurs. If we have a left recursion in our grammar, then it leads to infinite recursion, due to which we cannot generate the given string.

How to eliminate left recursion

We can eliminate left recursion by replacing a pair of production with:

A → βA′

A′ → αA′ |ϵ

Algorithm:

1. Declare arrays for input, L, r, temp, pemprod, production
2. Declare variables
3. Input the production in l, r
4. Print r
5. Check for left recursion in string
6. remove by grammer rule
7. update the arrays
8. display the production after elemination of left recursion.

COMPUTING ENVRONMENT
Platform: ubuntu
Programming Language: C / C++ / Java

Expected Outcome:

Enter the productions: E->E+E|T

The productions after eliminating Left Recursion are:

E->+EE'

E'->TE'

E-> ε

Conclusion: Thus the program to eliminate left recursion is implemented.

Viva Voce Questions:

1. What is left recursion?

Answer:A grammar in the form G = (V, T, S, P) is said to be in left recursive form if it has the production rules of the form A → A α |β. In the production rule above, the variable in the left side occurs at the first position on the right side production, due to which the left recursion occurs. If we have a left recursion in our grammar, then it leads to infinite recursion, due to which we cannot generate the given string.

2. How to eliminate Left recursion?

Answer:We can eliminate left recursion by replacing a pair of production with:

A → βA′

A′ → αA′ |ϵ

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381,237307, Fax : 07104 – 237681,

Experiment No:11

AIM: To convert given expression into Three Address Code.
Write three address code for following expression

a=b+c*d

Theory:

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 -

a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

Implementation of Three Address Code -

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple -

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage -

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage -

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

ALGORITHM:

Step1: Begin the program
Step2 : The expression is read from the file using a file pointer
Step3 : Each string is read and the total no. of strings in the file is calculated.
Step4: Each string is compared with an operator; if any operator is seen then the previous string and next string are concatenated and stored in a first temporary value and the three address code expression is printed
Step5 : Suppose if another operand is seen then the first temporary value is concatenated to the next string using the operator and the expression is printed.
Step6 : The final temporary value is replaced to the left operand value.
Step7 : End the program.

COMPUTING ENVRONMENT

Platform: ubuntu
Programming Language: C /C++

Expected Output:

Enter the expression: a=b+c*d

A:= c*d B:= b+A C:= A + B

Coclusion: Thus the given expression is converted into Three Address code.

VIva Voce Questions:

1. Why it is called 3 address code?

Answer:The reason for the name “three address code” is that **each statement generally includes three addresses, two for the operands, and one for the result.** In the three-address code, atmost three addresses are define any statement. Two addresses for operand & one for the result. Hence, op is an operator.

2. What are the different forms of three address code?

Answer:The three address code can be represented in three forms: quadruples , triples and indirect.

3. What is three address code in compiler construction?

Answer: In computer science, three-address code (often abbreviated to TAC or 3AC) is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations. Each TAC instruction has at most three operands and is typically a combination of assignment and a binary operator.

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No:12

AIM: To convert given expression into Three Address Code.

Write three address code for following expression

a=-c*d+b-c

ALGORITHM:

Step1: Begin the program
Step2 : The expression is read from the file using a file pointer
Step3 : Each string is read and the total no. of strings in the file is calculated.
Step4: Each string is compared with an operator; if any operator is seen then the previous string and next string are concatenated and stored in a first temporary value and the three address code expression is printed
Step5 : Suppose if another operand is seen then the first temporary value is concatenated to the next string using the operator and the expression is printed.
Step6 : The final temporary value is replaced to the left operand value.
Step7 : End the program.

COMPUTING ENVRONMENT
Platform: ubuntu
Programming Language: C / C++
Expected Output:

Input: a=-c*d+b-c

Output : out.txt

```
t1=- int1
t2=t1*int2
t4=t2+int3
t5=t4-int1
int4=t5
```

Conclusion: Thus the program to covert given expression into Three Address Code is executed.

VIva Voce Questions:

1. Which of the following is are the forms of intermediate code representation?

Answer: Intermediate code can be represented in three forms, which are postfix notation, Syntax trees, Three address code.

2. What is Indirect Triple:

Answer: Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381,237307, Fax : 07104 – 237681,

Experiment No:13

AIM: To implement Code optimization.

THEORY:he code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

When to Optimize?

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Why Optimize?

Optimizing an algorithm is beyond the scope of the code optimization phase. So the program is optimized. And it may involve reducing the size of the code. So optimization helps to:

- Reduce the space consumed and increases the speed of compilation.
- Manually analyzing datasets involves a lot of time. Hence we make use of software like Tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer.
- An optimized code often promotes re-usability.

Types of Code Optimization: The optimization process can be broadly classified into two types :

1. **Machine Independent Optimization:** This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
2. **Machine Dependent Optimization:** Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

ALGORITHM:

Step1: Generate the program for factorial program using for and do-while loop to specify optimization technique.

Step2: In for loop variable initialization is activated first and the condition is checked next. If the condition is true the corresponding statements are executed and specified increment / decrement operation is performed.

Step3: The for loop operation is activated till the condition failure.

Step4: In do-while loop the variable is initialized and the statements are executed then the condition checking and increment / decrement operation is performed.

Step5: When comparing both for and do-while loop for optimization dowhile is best because first the statement execution is done then only the condition is checked. So, during the statement execution itself we can find the inconvenience of the result and no need to wait for the specified condition result.

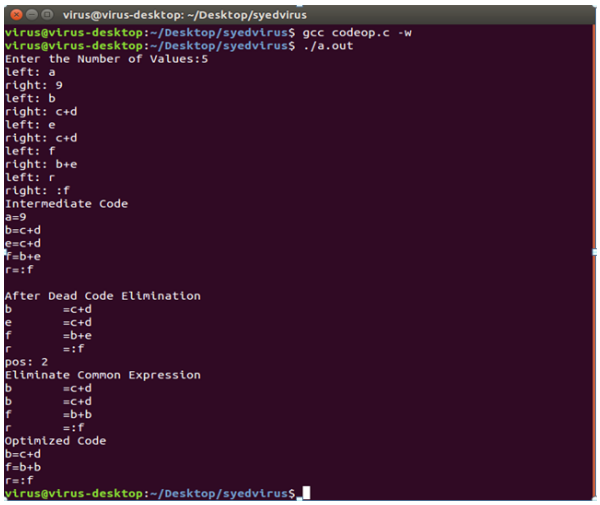
Step6: Finally when considering Code Optimization in loop do-while is best with respect to performance.

COMPUTING ENVRONMENT

Platform: ubuntu

Programming Language: FLEX, C / C++

Expected Output:



Viva VOce Questions:

1. What is the purpose of code optimization?

Answer: Code optimization is any method of code modification to improve code quality and efficiency. A program may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer input/output operations.

2. What is the key part of code optimization?

Answer: Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed. In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes

3. Why code optimization is called optional phase?

Answer: Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

PRIYADARSHINI COLLEGE OF ENGINEERING
(Recognised by A.I.C.T.E., New Delhi & Govt. of Maharashtra, Affiliated to R.T.M.Nagpur University)Near CRPF Campus, Hingna Road, Nagpur-440 019, Maharashtra (India)
Phone : 07104 – 236381, 237307, Fax : 07104 – 237681,

Experiment No:14

AIM: To create symbol table for any given code.

THEORY: A Symbol table is a data structure used by the compiler, where each identifier in program's source code is stored along with information associated with it relating to its declaration. It stores identifier as well as it's associated attributes like scope, type, line-number of occurrence, etc.

Symbol table can be implemented using various data structures like:

- LinkedList
- Hash Table
- Tree

A common data structure used to implement a symbol table is HashTable.

Operations of Symbol table – The basic operations defined on a symbol table include:

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Consider the following C++ function:

```
// Define a global function
int add(int a, int b)
{
int sum = 0;
sum = a + b;
return sum;
}
```

Symbol Table for above code:

Name	Type	Scope
add	function	global
a	int	function parameter
b	int	function parameter
sum	int	local

Below is the C++ implementation of Symbol Table using the concept of Hashing with separate chaining:

Algorithm/ Functions

```
/ C++ Functions to implement Symbol Table
#include <iostream>

class Node {

        string identifier, scope, type;
        int lineNo;
        Node* next;

public:
        Node()
        {
                next = NULL;
        }

        Node(string key, string value, string type, int lineNo)
        {
                this->identifier = key;
                this->scope = value;
                this->type = type;
                this->lineNo = lineNo;
                next = NULL;
        }

        void print()
        {
                cout << "Identifier's Name:" << identifier
                        << "\nType:" << type
                        << "\nScope: " << scope
                        << "\nLine Number: " << lineNo << endl;
```

```
        }
        friend class SymbolTable;
};

class SymbolTable {
        Node* head[MAX];

public:
        SymbolTable()
        {
                for (int i = 0; i < MAX; i++)
                        head[i] = NULL;
        }

        int hashf(string id); // hash function
        bool insert(string id, string scope,
                                string Type, int lineno);

        string find(string id);

        bool deleteRecord(string id);

        bool modify(string id, string scope,
                                string Type, int lineno);
};

// Function to modify an identifier
bool SymbolTable::modify(string id, string s,
                                                string t, int l)
{
        int index = hashf(id);
        Node* start = head[index];

        if (start == NULL)
                return "-1";

        while (start != NULL) {
                if (start->identifier == id) {
                        start->scope = s;
                        start->type = t;
                        start->lineNo = l;
                        return true;
                }
                start = start->next;
        }

        return false; // id not found
```

```
}

// Function to delete an identifier
bool SymbolTable::deleteRecord(string id)
{
        int index = hashf(id);
        Node* tmp = head[index];
        Node* par = head[index];

        // no identifier is present at that index
        if (tmp == NULL) {
                return false;
        }
        // only one identifier is present
        if (tmp->identifier == id && tmp->next == NULL) {
                tmp->next = NULL;
                delete tmp;
                return true;
        }

        while (tmp->identifier != id && tmp->next != NULL) {
                par = tmp;
                tmp = tmp->next;
        }
        if (tmp->identifier == id && tmp->next != NULL) {
                par->next = tmp->next;
                tmp->next = NULL;
                delete tmp;
                return true;
        }

        // delete at the end
        else {
                par->next = NULL;
                tmp->next = NULL;
                delete tmp;
                return true;
        }
        return false;
}

// Function to find an identifier
string SymbolTable::find(string id)
{
        int index = hashf(id);
        Node* start = head[index];
```

```
        if (start == NULL)
                return "-1";

        while (start != NULL) {

                if (start->identifier == id) {
                        start->print();
                        return start->scope;
                }

                start = start->next;
        }

        return "-1"; // not found
}

// Function to insert an identifier
bool SymbolTable::insert(string id, string scope,
                                                string Type, int lineno)
{
        int index = hashf(id);
        Node* p = new Node(id, scope, Type, lineno);

        if (head[index] == NULL) {
                head[index] = p;
                cout << "\n"
                          << id << " inserted";

                return true;
        }

        else {
                Node* start = head[index];
                while (start->next != NULL)
                        start = start->next;

                start->next = p;
                cout << "\n"
                          << id << " inserted";

                return true;
        }

        return false;
}

int SymbolTable::hashf(string id)
```

```
{
        int asciiSum = 0;

        for (int i = 0; i < id.length(); i++) {
                asciiSum = asciiSum + id[i];
        }

        return (asciiSum % 100);
}

// Driver code
int main()
{
        SymbolTable st;
        string check;
        cout << "**** SYMBOL_TABLE ****\n";

        // insert 'if'
        if (st.insert("if", "local", "keyword", 4))
                cout << " -successfully";
        else
                cout << "\nFailed to insert.\n";

        // insert 'number'
        if (st.insert("number", "global", "variable", 2))
                cout << " -successfully\n\n";
        else
                cout << "\nFailed to insert\n";

        // find 'if'
        check = st.find("if");
        if (check != "-1")
                cout << "Identifier Is present\n";
        else
                cout << "\nIdentifier Not Present\n";

        // delete 'if'
        if (st.deleteRecord("if"))
                cout << "if Identifier is deleted\n";
        else
                cout << "\nFailed to delete\n";

        // modify 'number'
        if (st.modify("number", "global", "variable", 3))
                cout << "\nNumber Identifier updated\n";

        // find and print 'number'
```

```
        check = st.find("number");
        if (check != "-1")
                cout << "Identifier Is present\n";
        else
                cout << "\nIdentifier Not Present";

        return 0;
}
```

COMPUTING ENVRONMENT

Platform: ubuntu

Programming Language: C / C++

Expected Output:

**** SYMBOL_TABLE ****

if inserted -successfully

number inserted -successfully

Identifier's Name:if

Type:keyword

Scope: local

Line Number: 4

Identifier Is present

if Identifier is deleted

number Identifier updated

Identifier's Name:number

Type:variable

Scope: global

Line Number: 3

Identifier Is present

Conclusion: Thus the program to implement symbol table using the concept of Hashing with separate chaining is executed.

VIva Voce Questions:

1. What is the use of symbol table in compiler?

Answer: Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

2. What are the various ways a symbol table can be stored?

Answer: Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc.

A symbol table can be implemented in one of the following techniques:

- Linear (sorted or unsorted) list.
- Hash table.
- Binary search tree.

List of References

[1] Compilers: Principles, Techniques, and Tools (Second Edition),Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman.Addison-Wesley
[2] https://silcnitc.github.io/lex.html
[3] https://silcnitc.github.io/lex.html
[4] https://www.wisdomjobs.com/e-university/compiler-design-interview-questions.html
[5] https://www.geeksforgeeks.org/lex-program-to-implement-a-simple-calculator/
[6] https://www.geeksforgeeks.org/cpp-program-to-implement-symbol-table/