Vidya Vikas Education Trust's
Universal College of Engineering
(Permanently unaided | Approved by AICTE, DTE & Affiliated to University of Mumbai)

Department of Computer Science & Engineering

Vision:
To be recognized globally as a department provides quality technical education that eventually caters to helping and serving the community.

Mission:
To develop human resources with sound knowledge in theory and practice of computer science and engineering. To motivate the students to solve real-world problems to help the society grow. To provide a learning ambience to enhance innovations, team spirit and leadership qualities for students.

LAB MANUAL

(CSL 601)

System Software Lab (SPCC)

Name: **VIVEK.SHIVAKUMAR. HOTTI**

Class: **T. E–Computer Engineering**

Semester: **VI**

Roll No.: **37**

Batch: **A2**

Faculty Name: **Mrs. Hezal Lopes**

2021 – 2022

Lab Code	Lab Name	Credits
CSL502	Computer Network Lab	1

Lab Outcomes:

After successful completion of this course student will be able to:

1. Generate machine code by using various databases generated in pass one of two pass assembler.
2. Construct different databases of single pass macro processor. Identify and validate different tokens for given high level language code.
3. Parse the given input string by constructing Top down /Bottom-up parser.
4. Implement synthesis phase of compiler with code optimization techniques.
5. Explore various tools like LEX and YACC.

Description:

The current System Software is highly complex with huge built in functionality offered to the programmer to develop complex applications with ease. This laboratory course aims to make a student
understand-

- The need for modular design
- The need for well-defined data structures and their storage management
- The increase in the complexity of translators as we move from assembly level to high level programming
- The need to produce an efficient machine code that is optimized for both execution speed and memory requirement
- The efficient programming constructs that make them a good coder

Term work:

Laboratory work will be based on above syllabus with minimum 10 experiments to be incorporated.
The distribution of marks for term work shall be as follows:

 Laboratory work (experiments/case studies): …….(15) Marks.
 Assignment: ...…….……...(05) Marks.
 Attendance ……..……(05) Marks.
TOTAL: ……..(25) Marks.

Oral & Practical exam will be based on the above and CSC602 syllabus.

Vidya Vikas Education Trust's
Universal College of Engineering
(Permanently unaided | Approved by AICTE, DTE & Affiliated to University of Mumbai)

INDEX

Name: Vivek. Hotti **Roll:** 37
Class: TE- Computers **Division:** A
Semester: 6

Reference Books:

1. Modern Compiler. Implementation in Java, Second. Edition. Andrew W. Appel Princeton University. Jens Palsberg Purdue University. CAMBRIDGE.

2. Crafting a compiler with C, Charles N. Fischer, Ron K. Cytron, Richard J. LeBlanc.

EXPERIMENT – 01

AIM: Case Study of LEX & YACC.

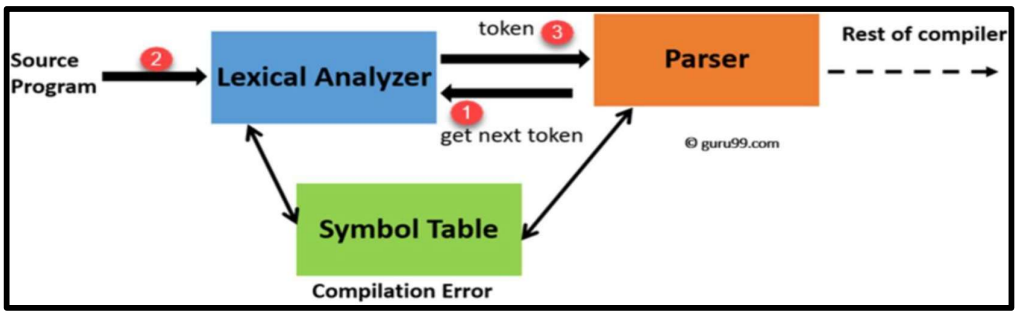
THEORY:

LEX COMPILERS:

What is Lex:

- LEX is Lexical analyzer. It is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens. The output is a sequence of tokens that is sent to the parser for syntax analysis. The tokens are in the form <token, value> for example a=b+c; the token is generated as follows "<id,1> <=> <id,2> <+> <id,3> <;>". A character sequence which is not possible to scan into any valid token is a lexical error.
- The Lexical analyzer is a part of the analysis phase of the compiler. Lexical analyzer is responsible to perform the below given tasks:
  - Help to identify the tokens into the symbol table.
  - Remove white spaces and comments from the source program.
  - Correlate error messages with the source program.
  - Help to expand the macros if it is found in the source program.
  - Read input characters from the source program.
- The code written in Lex recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions.
- Lex helps in writing the programs whose control flow is directed by instances of regular expressions in the input stream.

LEX Compiler Architecture:



- Lexical analyzer scans the entire source code of the program. It identifies each token one by one. Scanners are usually implemented to produce tokens only when requested by a parser. The architecture is divided into 3steps:
    - "Get next token" is a command which is sent from the parser to the lexical analyzer.
    - On receiving this command, the lexical analyzer scans the input until it finds the next token.
    - It returns the token to Parser.

- Lexical analyzer skips whitespaces and comments while creating these tokens. If any error is present, then Lexical analyzer will correlate that error with the source file and line number.

How to write program in LEX:

LEX is written in C language The LEX program is divided into 3 parts : definitions, rules and subroutines. They are separated from each other using the "%%" symbol as follows.

... definitions ...

%%

... rules ...
%%

... subroutines ...

The definition part contains all header files and initialization of global variables, rules contain the main logic of the program and the subroutines contain C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer. The file containing the lex code should be saved with .l or .lex extension.

Steps to Run a LEX Program:

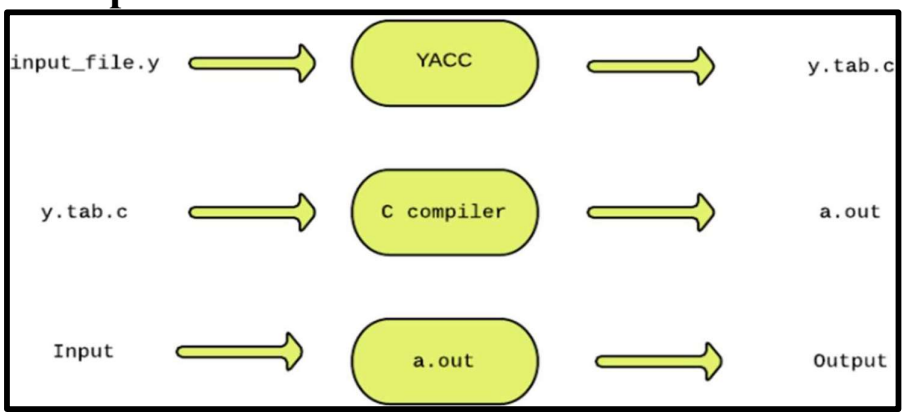
Step 1: lex filename.l or lex filename.lex depending on the extension used while saving the file
Step 2: gcc lex.yy.c
Step 3: ./a.out
Step 4: Provide the input to program in case it is required

YACC COMPILERS

What is YACC:

- YACC stands for Yet Another Compiler-Compiler. It is a computer program for the Unix operating system It is a Look Ahead Left-to-Right Rightmost derivation producer with 1 lookahead token (LALR(1)) parser generator, generating a LALR parser that is the part of a compiler that tries to make syntactic sense of the source code based on a formal grammar.
- 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 was originally designed for being complemented by Lex.

YACC Compiler Architecture:



- yacc is designed for use with C code and generates a parser written in C.
- The parser is configured for use in conjunction with a lex-generated scanner and relies on standard shared features (token types, yylval, etc.) and calls the function yylex as a scanner coroutine.
- You provide a grammar specification file, which is traditionally named using a .y extension.
- You invoke yacc on the .y file and it creates the y.tab.h and y.tab.c files containing a thousand or so lines of intense C code that implements an efficient LALR (1) parser for your grammar, including the code for the actions you specified.
- The file provides an external function yyparse.y that will attempt to successfully parse a valid sentence.
- You compile that C file normally, link with the rest of your code, and you have a parser! By default, the parser reads from stdin and writes to stdout, just like a lex-generated scanner does.

How to write program in YACC:

The YACC input file is divided into three parts Just like lex That is definitions rules and Auxiliary routines All separated by the symbol "%%".

/* definitions */

....

%%

/* rules */

....

%%

/* auxiliary routines */

....

The definition part includes information about the tokens used in the syntax definition. Yacc automatically assigns numbers for tokens, but it can be overridden. 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. The rules part contains the grammar definition in a modified BNF form. Actions is C code in { } and can be embedded inside (Translation schemes). The auxiliary routines part is only C code. It includes function definitions for every function needed in the 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(). The file is saved with the .y extension.

Steps to Run a YACC Program:

Step 1: Write lex program in a file file.l and yacc in a file file.y

Step 2: Open Terminal and Navigate to the Directory where you have saved the files
Step 3: Type lex file.1
Step 4: Type yacc file.y
Step 5: Type cc lex.yy.c y.tab.g -11
Step 6: Type ./a.out

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 definitions (particularly important are token definitions to be included, for example, in a Lex input file). Else 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.

Discussion:

1. What is system program?
   Ans: System program is defined as an environment where programs can be developed and executed. In the simplest sense, system programs also provide a bridge between the user interface and system calls.

2. Define assembler, compiler, Interpreter, Linker Loader, Editor?
   Ans: An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory. A compiler is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language). An interpreter is a common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user. 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 a separate assembler.

Loader is a part of the operating system and is responsible for loading executable files into memory and executing them. Editors or text editors are software programs that enable the user to create and edit text files. In the field of programming, the term editor usually refers to source code editors that include many special features for writing and editing code.

3. Differentiate Application program and system program.
   Ans:

Application program	System program
1) It is the set of programs that view computers as a tool for solving a particular problem.	1) It is the collection of components and art or designing of a given program.
2) Application programming is used to build application software which includes software.	2) Programming is done using assembly language which interacts with hardware.
3) Application software is software that has been used by the end user.	3) System software that executes the application software.
4) Examples: - Library management System, calculator, web browser	4)Examples: - Loader, Linker, Compiler

4. Differentiate compiler and interpreter.
   Ans:

Compiler	Interpreter
1)Compiler scans the entire program and translates the whole of it into machine code at once.	1)Interpreter translates just one statement of the program at a time into machine code.
2)A compiler takes a lot of time to analyze the source code. However, the overall time taken to execute the process is much faster.	2)An interpreter takes very less time to analyze the source code. However, the overall time to execute the process is much slower.

3)A compiler always generates an intermediary object code. It will need further linking. Hence more memory is needed.	3)An interpreter does not generate an intermediary code. Hence, an interpreter is highly efficient in terms of its memory.
4)Compilers are used by programming languages like C and C++ for example.	4)Interpreters are used by programming languages like Ruby and Python for example.

5. Goals of system software
   Ans: Primary goal is to make computer systems easier for users. So, we have an OS that provides interface between user and Hardware. Allocate system resources to various application programs as efficient as possible.

CONCLUSION:

Hence, we have studied about LEX and YACC Compilers and how to run their programs.

x-x-x

Experiment 01 Over

EXPERIMENT – 02

AIM: Implementation of Lexical Analyzer..

THEORY:

LEXICAL ANALYZERS

- What are the functions of a Lexical Analyzer?
- It produces stream of tokens.
- It eliminates blank space and comments
- It generates symbol table which stores the information about identifier, constant.
- It reports error.

What is LEXEME:

- Lexeme is the smallest logical unit of a program. It consists of a sequence of characters for which a token is produced. E.g. "Hello", 25, int, +,etc.
- The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. e.g.- "float", "abs_zero_Kelvin", "=", "-", "273", ";" .

What are TOKENS:

- A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.
- Example of tokens:
  - Type token (id, number, real, . . .)
  - Punctuation tokens (IF, void, return, . . .)
  - Alphabetic tokens (keywords)

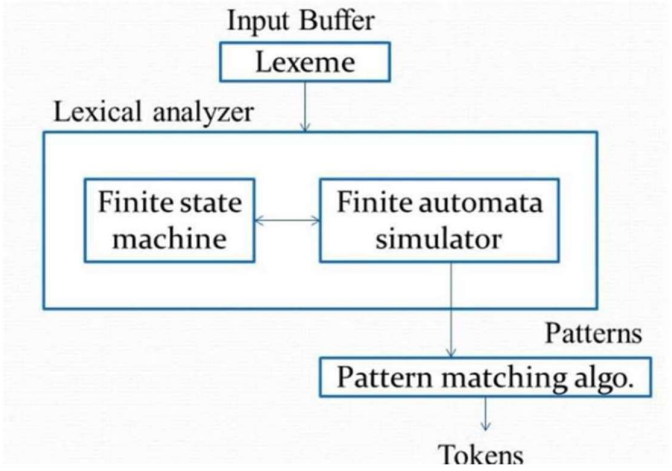
What are Patterns:

- Lexeme Pattern is a template or model. It consists of a set of Rules that describes how a token is formed.

What is the use of Finite Automata in Lexical Analysis?

- To design a lexical analyzer generator the patterns of RE are designed first. These patterns are for recognizing various tokens from the input string. From this pattern it is very easy to design NFA. Then convert NFA to DFA because simulation of DFA is easier for programs.

Block Schematic of Lexical Analyzer?



INPUT PROGRAM

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
ch == '[' || ch == ']' || ch == '{' || ch == '}')
return (true);
return (false);
```

```
}
// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
if (ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == '>' || ch == '<' ||
ch == '=')
return (true);
return (false);
}
// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
{
if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
str[0] == '3' || str[0] == '4' || str[0] == '5' ||
str[0] == '6' || str[0] == '7' || str[0] == '8' ||
str[0] == '9' || isDelimiter(str[0]) == true)
return (false);
return (true);
}
// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
if (!strcmp(str, "if") || !strcmp(str, "else") ||
!strcmp(str, "while") || !strcmp(str, "do") ||
!strcmp(str, "break") ||
!strcmp(str, "continue") || !strcmp(str, "int")
|| !strcmp(str, "double") || !strcmp(str, "float")
|| !strcmp(str, "return") || !strcmp(str, "char")
|| !strcmp(str, "case") || !strcmp(str, "char")
|| !strcmp(str, "sizeof") || !strcmp(str, "long")
|| !strcmp(str, "short") || !strcmp(str, "typedef")
|| !strcmp(str, "switch") || !strcmp(str, "unsigned")
|| !strcmp(str, "void") || !strcmp(str, "static")
|| !strcmp(str, "struct") || !strcmp(str, "goto"))
return (true);
return (false);
}
// Returns 'true' if the string is an INTEGER.
bool isInteger(char* str)
{
int i, len = strlen(str);
if (len == 0)
return (false);
for (i = 0; i < len; i++) {
if (str[i] != '0' && str[i] != '1' && str[i] != '2'
&& str[i] != '3' && str[i] != '4' && str[i] != '5'
&& str[i] != '6' && str[i] != '7' && str[i] != '8'
&& str[i] != '9' || (str[i] == '-' && i > 0))
return (false);
}
```

```
return (true);
}
// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char* str)
{
int i, len = strlen(str);
bool hasDecimal = false;
if (len == 0)
return (false);
for (i = 0; i < len; i++) {
if (str[i] != '0' && str[i] != '1' && str[i] != '2'
&& str[i] != '3' && str[i] != '4' && str[i] != '5'
&& str[i] != '6' && str[i] != '7' && str[i] != '8'
&& str[i] != '9' && str[i] != '.' ||
(str[i] == '-' && i > 0))
return (false);
if (str[i] == '.')
hasDecimal = true;
}
return (hasDecimal);
}
// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
int i;
char* subStr = (char*)malloc(
sizeof(char) * (right - left + 2));
for (i = left; i <= right; i++)
subStr[i - left] = str[i];
subStr[right - left + 1] = '\0';
return (subStr);
}
// Parsing the input STRING.
void parse(char* str)
{
int left = 0, right = 0;
int len = strlen(str);
while (right <= len && left <= right) {
if (isDelimiter(str[right]) == false)
right++;
if (isDelimiter(str[right]) == true && left == right) {
if (isOperator(str[right]) == true)
printf("'%c' IS AN OPERATOR\n", str[right]);
right++;
left = right;
} else if (isDelimiter(str[right]) == true && left != right
|| (right == len && left != right)) {
char* subStr = subString(str, left, right - 1);
if (isKeyword(subStr) == true)
printf("'%s' IS A KEYWORD\n", subStr);
else if (isInteger(subStr) == true)
```

```
printf("'%s' IS AN INTEGER\n", subStr);
else if (isRealNumber(subStr) == true)
printf("'%s' IS A REAL NUMBER\n", subStr);
else if (validIdentifier(subStr) == true
&& isDelimiter(str[right - 1]) == false)
printf("'%s' IS A VALID IDENTIFIER\n", subStr);
else if (validIdentifier(subStr) == false
&& isDelimiter(str[right - 1]) == false)
printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
left = right;
}
}
return;
}
// DRIVER FUNCTION
int main()
{
// maximum length of string is 100 here
char str[100] = " int sum = a + b; ";
parse(str); // calling the parse function
return (0);
}
```

RECEIVED OUTPUT:

```
PS C:\Users\Vivek hotti\desktop\NasaProject> cd "c:\Users\Vivek hotti\Desktop\"
'int' IS A KEYWORD
'sum' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'a' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
PS C:\Users\Vivek hotti\Desktop>
```

(As can be seen from above program, our input to the program was : int sum = a + b;)

CONCLUSION:

Hence, we have studied how to implement a LEX Analyzer.

Discussion:

1. What Phases of compiler.

Ans: - There are 6 phases of compiler i.e., Lexical Analyzer, Syntax Analyzer, Semantic Analyzer, Intermediate Code Generator, Code Optimization, Code Generation.

2. What cross compiler.
   Ans: - A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on an Android smartphone is a cross compiler.

3. What is source to source compiler.
   Ans: - A It is a type of translator that takes the source code of a program written in a programming language as its input and produces an equivalent source code in the same or a different programming language.

4. Analysis phase and synthesis phase
   Ans:- Analysis phase is defined as phase that creates an intermediate representation from the given source code whereas Synthesis phase is defined as phase that creates an equivalent target program from the intermediate representation.

5. What is lexical analysis
   Ans:- The lexical analyzer breaks the syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

6. What is syntax analysis
   Ans:- Syntax Analysis or Parsing is the second phase which checks the syntactic structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not.

7. What is the name for lexical analyzer?
   Ans:- Lexical Analyzer is also known as Scanner.

8. What is the another name for syntax analyzer
   Ans:- Syntax analyzer is also called as Parser.

x-x-x

Experiment 02 Over

EXPERIMENT – 03

AIM: Program to implement first().

THEORY:

first ()

Why is first() required ? :

- When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation.
- The given input can be produced by the given grammar; therefore, the input is correct in syntax.
- But backtrack is needed to get the correct syntax tree, which is really a complex process to implement.
- There can be an easier way to solve this, which we shall see in the next article "Concepts of FIRST and FOLLOW sets in Compiler Design".
- FIRST(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X.

Rules to compute first() ?:

- 1. If *X* is terminal, then FIRST(*X*) is {X}.
- 2. If *X* → ε is a production, then add ε to FIRST(*X*).
- 3. If *X* is non-terminal and X → *a*α is a production, then add *a* to FIRST(X).
- 4. If X is non-terminal and *X* → *Y1Y2…Yk* is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(*Yi*) and ε is in all of FIRST(*Y1*) …, FIRST(*Yi-1*);
- that is, *Y1*, …, *Yi-1* → ε. If ε is in FIRST(*Yj*) for all j=1, 2, …, k, then add ε to FIRST(*X*).

INPUT CODE TO COMPUTE first():

```
#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');
}
/*
*Function FIRST:
*Compute the elements in FIRST(c) and write them
*in Result Array.
*/
void FIRST(char* Result,char c)
{
int i,j,k;
char subResult[20];
int foundEpsilon;
subResult[0]='\0';
Result[0]='\0';
//If X is terminal, FIRST(X) = {X}.
if(!(isupper(c)))
{
addToResultSet(Result,c);
return ;
}
//If X is non terminal
```

```
//Read each production
for(i=0;i<numOfProductions;i++)
{
//Find production with X as LHS
if(productionSet[i][0]==c)
{
//If X → ε is a production, then add ε to FIRST(X).
if(productionSet[i][2]=='$') addToResultSet(Result,'$');
//If X is a non-terminal, and X → Y1 Y2 ... Yk
//is a production, then add a to FIRST(X)
//if for some i, a is in FIRST(Yi),
//and ε is in all of FIRST(Y1), ..., FIRST(Yi-1).
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;
}
//No ε found, no need to check next element
if(!foundEpsilon)
break;
j++;
}
}
}
}
return ;
}
/* addToResultSet adds the computed
*element to result set.
*This code avoids multiple inclusion of elements
*/
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

```
C:\Users\Vivek hotti\Desktop\a.exe
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 : y
```

CONCLUSION:

Hence, we have studied first(), the rules to implement first(), and written a program to execute the same.

Discussion:

1. Predictive parser table

Ans:- Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string.

2. Top down and bottom up parser

Ans:-

S.No	Top Down Parsing	Bottom Up Parsing
1.	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
2.	Top-down parsing attempts to find the left most derivations for an input string.	Bottom-up parsing can be defined as an attempts to reduce the input string to start symbol of a grammar.
3.	In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of parse tree) in top-down manner.	In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start symbol of parse tree) in bottom-up manner.
4.	This parsing technique uses Left Most Derivation.	This parsing technique uses Right Most Derivation.
5.	It's main decision is to select what production rule to use in order to construct the string.	It's main decision is to select when to use a production rule to reduce the string to get the starting symbol.

3. Steps for LL1 Parser

Ans.

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a].
3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $].

4. Operator precedence parser

Ans: - An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR (1) grammars. More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive nonterminal and epsilons never appear in the right-hand side of any rule.

5. What is semantic analysis

Ans: - Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

6. Inherited Attribute

Ans: - An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or sibling nodes.

7. Synthesized Attributes

Ans: - An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.

x-x-x

Experiment 03 Over

EXPERIMENT – 04

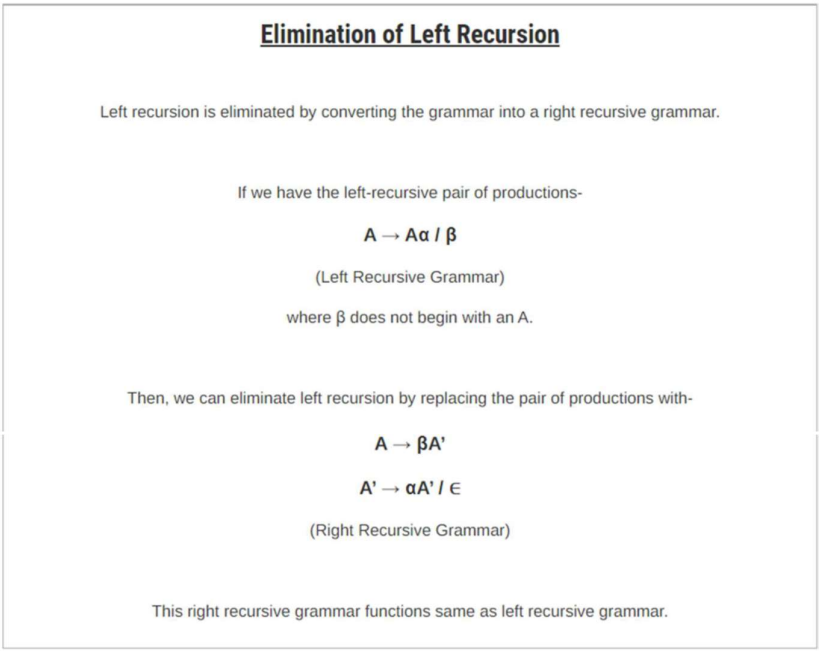
AIM: Program to implement & eliminate left recursion.

THEORY:

What is left recursion?

- Left recursion is a special case of recursion where a string is recognized as part of a language by the fact that it decomposes into a string from that same language (on the left) and a suffix (on the right).
- When a production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having left recursion is called as Left Recursive Grammar.
- But Left recursion is considered to be a problematic situation for Top-down parsers.
- Therefore, left recursion has to be eliminated from the grammar.

Rules to eliminate left recursion?



INPUT to eliminate left recursion:

```
#include<stdio.h>
#include<string.h>
#define SIZE 10
int main () {
char non_terminal;
char beta,alpha;
int num;
char production[10][SIZE];
int index=3; /* starting of the string following "->" */
printf("Enter Number of Production : ");
scanf("%d",&num);
printf("Enter the grammar as E->E-A :\n");
for(int i=0;i<num;i++){
scanf("%s",production[i]);
}
for(int i=0;i<num;i++){
printf("\nGRAMMAR : : : %s",production[i]);
non_terminal=production[i][0];
if(non_terminal==production[i][index]) {
alpha=production[i][index+1];
printf(" is left recursive.\n");
while(production[i][index]!=0 && production[i][index]!='|')
index++;
if(production[i][index]!=0) {
beta=production[i][index+1];
printf("Grammar without left recursion:\n");
printf("%c->%c%c\'",non_terminal,beta,non_terminal);
printf("\n%c\'->%c%c\'|E\n",non_terminal,alpha,non_terminal);
}
else
printf(" can't be reduced\n");
}
else
printf(" is not left recursive.\n");
index=3;
}
}
```

OUPUT To Above Program

(Next page)

```
 } ; if ($?) { .\exp4VH }
Enter Number of Production : 4
Enter the grammar as E->E-A :
E->EA|A
A->AT|a
T=a
E->i

GRAMMAR : : : E->EA|A is left recursive.
Grammar without left recursion:
E->AE'
E'->AE'|E

GRAMMAR : : : A->AT|a is left recursive.
Grammar without left recursion:
A->aA'
A'->TA'|E

GRAMMAR : : : T=a is not left recursive.

GRAMMAR : : : E->i is not left recursive.
PS C:\Users\Vivek hotti\Desktop>
```

CONCLUSION:

Hence, we have written a program to implement and eliminate left recursion.

Discussion:

1. What is intermediate code generation?
Ans:- Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code.

2. Types of intermediate code generation.
Ans:-

Postfix Notation – The ordinary (infix) way of writing the sum of a and b is with the operator in the middle : a + b . The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. In postfix notation the operator follows the operand.

Three-Address Code – A statement involving no more than three references(two for operands and one for result) is known as three address statements. A sequence of three address statements is known as three address code. Three address statements are of the form x = y op z , here x, y, z will have address (memory location).

Syntax Tree – Syntax tree is nothing more than a condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by a single link in the syntax tree; the internal nodes are operators and child nodes are operands. To form a syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

3. What is DAG?
Ans:- Directed acyclic graph (DAG) is a directed graph with no directed cycles. That is, it consists of vertices and edges (also called arcs), with each edge directed from one vertex to another, such that following those directions will never form a closed loop.

4. Difference between parse tree and syntax tree.
Ans:-

Parse Tree	Syntax Tree
Parse tree is a graphical representation of the replacement process in a derivation.	Syntax tree is the compact form of a parse tree.
Each interior node represents a grammar rule. Each leaf node represents a terminal.	Each interior node represents an operator. Each leaf node represents an operand.
Parse trees provide every characteristic information from the real syntax.	Syntax trees do not provide every characteristic information from the real syntax.

5. What is the need of code optimization?

Ans:- 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. The basic requirements optimization methods should comply with, is that an optimized program must have the same output and side effects as its non-optimized version. This requirement, however, may be ignored in the case that the benefit from optimization is estimated to be more important than probable consequences of a change in the program behaviour.

x-x-x

Experiment 04 Over

EXPERIMENT – 05

AIM: Program to design parser (Operator Precedence Parser).

THEORY:

What is Operator Precedence Grammar?

- A grammar that is used to define mathematical operators is called an operator grammar or operator precedence grammar. Such grammars have the restriction that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its righthand side.
- An operator precedence parser is a bottom-up parser that interprets an operator grammar. This parser is only used for operator grammars. Ambiguous grammars are not allowed in any parser except operator precedence parser.
- An efficient way of constructing shift-reduce parser is called operator-precedence parsing. Operator precedence parser can be constructed from a grammar called Operator-Precedence grammar.

Rules for constructing a Operator Precedence Grammar ?

- a <· b – a yields precedence to b
- a = b – a has the same precedence as b

- a ·> b – a takes precedence over b
- Id has top most priority
- 1. ↑ is of highest precedence and right-associative
- 2. * and / are of next higher precedence and leftassociative,and
- 3. + and – are of lowest precedence and leftassociative.
- 4. $ has lowest priority.

INPUT to eliminate left recursion:

```
import numpy as np
def stringcheck():
a=list(input("Enter the operator used in the given grammar including the
terminals\n
non-treminals should be in cursive(small)letter"))
a.append('$')
print(a)
l=list("abcdefghijklmnopqrstuvwxyz")
o=list('(/*%+-)')
p=list('(/*%+-)')
n=np.empty([len(a)+1,len(a)+1],dtype=str,order="C")
for j in range(1,len(a)+1):
n[0][j]=a[j-1]
n[j][0]=a[j-1]
for i in range(1,len(a)+1):
for j in range(1,len(a)+1):
if((n[i][0] in l)and(n[0][j] in l)):
n[i][j]=""
elif((n[i][0] in l)):
n[i][j]=">"
elif((n[i][0] in o) and (n[0][j] in o)):
if(o.index(n[i][0])<=o.index(n[0][j])):
n[i][j]=">"
else:
n[i][j]="<"
elif((n[i][0] in o)and n[0][j]in l):
n[i][j]="<"
elif(n[i][0]=="$" and n[0][j]!="$"):
n[i][j]="<"
elif(n[0][j]=="$" and n[i][0]!="$" ):
```

```
n[i][j]=">"
else:
break
print("The Operator Precedence Relational
Table\n==========================================")
print(n)
i=list(input("Enter the string want to be checked(non-terminals should be in
cursive(small) letter..."))
i.append("$")
s=[None]*len(i)
q=0
s.insert(q,"$")
x=[row[0] for row in n]
y=list(n[0])
h=0
while(s[0]!=s[1]):
if((i[len(i)-2] in p)):
break
elif((s[q] in x)and(i[h]in y )):
if(n[x.index(s[q])][y.index(i[h])]=="<"):
q+=1
s.insert(q,i[h])
h+=1
elif(n[x.index(s[q])][y.index(i[h])]==">"):
s.pop(q)
q-=1
elif((n[x.index(s[q])][y.index(i[h])]==''')and ((s[q]=="$") and (i[h]=="$"))):
s[1]=s[0]
else:
break
if(s[0]!=s[1]):
return False
else:
return True
def grammarcheck(i):
print("Enter the",str(i+1)+"th grammar(production) want to be checked\n For null
production please enter any special symbol or whitespace...")
b=list(input().split("->"))
f=list("abcdefghijklmnopqrstuvwxyz")
if(b[0]==" " or b[0]=="" or b[0] in f or len(b)==1):
return False
else:
b.pop(0)
b=list(b[0])
s=list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
o=list("(abcdefghijklmnopqrstuvwxyz^/*+-|)")
sp=['!','@','#','$','?','~','`',',',',',':',''',',=',',_','&','"',"",","]
for i in range(0,len(b),2):
if(b[i]==" "):
g=False
elif(b[i] in sp):
```

```
g=False
break
elif(b[len(b)-1] in o and ((b[0]=="(" and b[len(b)-1]==")" )or
(b.count("(")==b.count(")")))):
g=True
elif(b[i] in f):
g=True
elif(b[len(b)-1] in o):
g=False
elif((i==len(b)-1) and (b[i] in s)):
g=True
elif((i==len(b)-1) and (b[i] not in s) and (b[i] in o)and b[i-1] in o):
g=True
elif((b[i] in s) and(b[i+1]in o)):
g=True
elif((b[i] in s) and (b[i+1] in s)):
g=False
break
else:
g=False
break
if(g==True):
return True
else:
return False
c=int(input("Enter the number of LHS variables..\n"))
for i in range(c):
if(grammarcheck(i)):
t=True
else:
t=False
break
if(t):
print("Grammar is accepted")
if(stringcheck()):
print("String is accepted")
else:
print("String is not accepted")
else:
print("Grammar is not accepted ")
```

OUPUT To Above Program

(next page)

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:\Users\hp\Desktop\CHEK.py ==========
Enter the number of LHS variables..
2
Enter the 1th grammar(production) want to be checked
 For null production please enter any special symbol or whitespace...
NoneA->a
Enter the 2th grammar(production) want to be checked
 For null production please enter any special symbol or whitespace...
NoneA->A+A
Grammar is accepted
Enter the operator used in the given grammar including the terminals
 non-treminals should be in cursive(small)lettera+
['a', '+', '$']
The Operator Precedence Relational Table
=========================================
[['' 'a' '+' '$']
 ['a' '' '>' '>']
 ['+' '<' '>' '>']
 ['$' '<' '<' '']]
Enter the string want to be checked(non-terminals should be in cursive(small) le
tter...a+a
String is accepted
>>>
```

CONCLUSION:

Thus, we have written a program to implement operator precedence parser.

DISCUSSION:

1. What are issues in code generation?

Ans:-

Input to code generator – The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc.

Target program – The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

Memory Management – Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

Instruction selection – Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also play a major role when efficiency is considered.

Register allocation issues – 38 Use of registers makes the computations faster in comparison to that of memory, so efficient utilization of registers is important.

Evaluation order – The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code.

Approaches to code generation issues - Code generators must always generate the correct code. It is essential because of the number of special cases that a code generator might face.

2. What is basic block and flow graph
Ans:- Basic Block is a straight line code sequence which has no branches in and out branches except to the entry and at the end respectively. Basic Block is a set of statements which always executes one after another, in a sequence. The first task is to partition a sequence of three-address code into basic blocks. A new basic block is begun with the first instruction and instructions are added until a jump or a label is met. In the absence of jump control moves further consecutively from one instruction to another. Flow graph is a

directed graph containing the flow-of-control information for the set of basic blocks making up a program. The nodes of the flow graph are basic blocks. It has a distinguished initial node.

x-x-x

Experiment 05 Over

EXPERIMENT – 06

AIM: Program to Optimize Code.

THEORY:

The 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.

Type of Code Optimization:

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

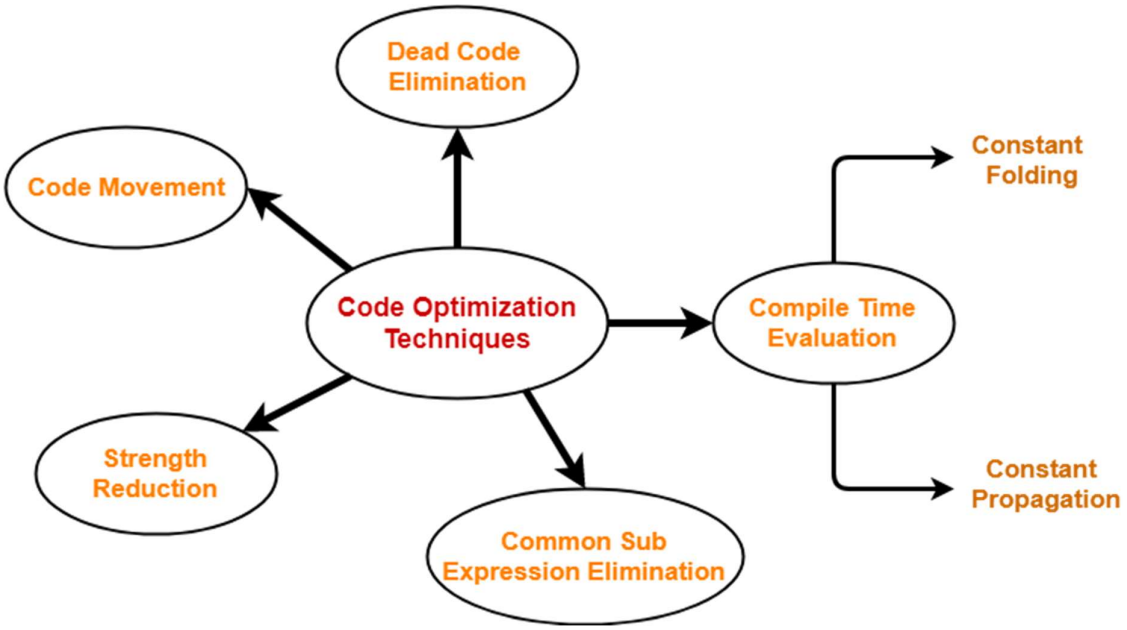
Machine Independent Optimization – This code optimization phase attempts to improve the intermediate code to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.

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.

Principle sources of Optimization

- Function preserving transformation
- Loop optimization
- Optimization for basic block
- Reducible flow graph
- Machine dependent/ Machine independent optimization.

Techniques of Optimization



1. **Compile Time Evaluation**: Two techniques that falls under compile time evaluation are
   - Constant Folding-
     - As the name suggests, it involves folding the constants.
     - The expressions that contain the operands having constant values at compile time are evaluated.
     - Those expressions are then replaced with their respective results.
   - Constant Propagation-
     - If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.

- The condition is that the value of variable must not get alter in between.

2. Common sub-expression elimination:

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

3. Dead Code Elimination:

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

4. Code Movement:

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

5. Strength Reduction:

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

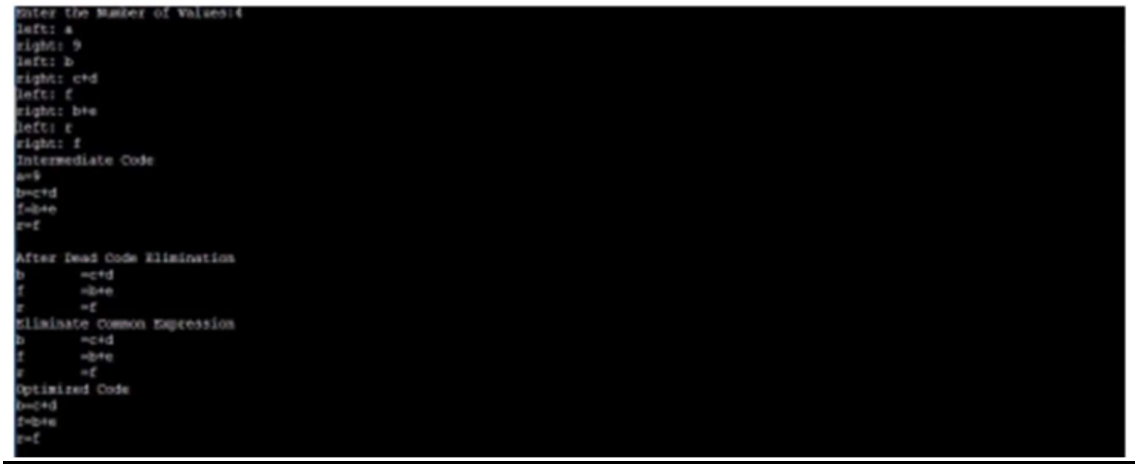
INPUT for DEAD CODE ELIMINATION:

```
#include<stdio.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;
printf("Enter the Number of Values:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
```

```
printf("left: ");
scanf(" %c",&op[i].l);
printf("right: ");
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\n",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';
}}}
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);
}
```

OUTPUT:



CONCLUSION:

Thus, A Program to Optimize Code has been implemented.

x-x-x

Experiment 06 Over

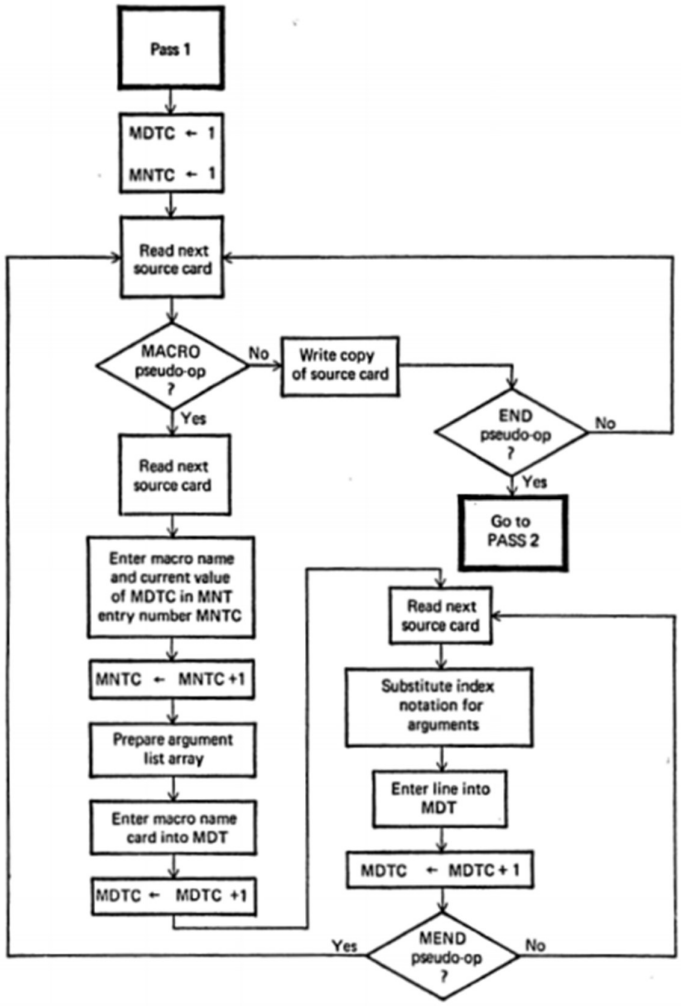
EXPERIMENT – 07

AIM: Program for PASS 1 Assembler (Generate table POT MOT ST LT BT).

THEORY:

- MOT : The Machine Operation Table is a table that indicates symbolic mnemonic for each instruction and its length
- POT : A Pseudo Operation Table is a table that indicates the symbolic mnemonic and action to be taken for each pseudo-op in pass1.
- ST : A Symbol Table is a table that is used to store each label and its corresponding value.
- LT : A Literal Table is a table that is used to store each literal encountered and its corresponding value and its corresponding assigned location.
- BT : A Base Table is a table that indicates which registers are currently specified as base registers by using presudo-ops and what are the specified contents of these registers.

Flowchart Pass 1 assembler:



INPUT

(main.java)

```
import java.io.*;
import java.util.*;

class Main{
   public static void main (String[] args) throws Exception {
      System.out.println("Vivek Hotti - 37");
      String m[] = {"L","S","ST"};
      String b[] = {"1001","1011","1111"};
      String p[] = {"DC","EQU","DS","USING","END"};
      int l[] = {4,1,4};
      int LC = 0;
      Vector<String>mneumonic = new Vector();
      Vector<String>bincode = new Vector();
```

```
Vector<Integer>mlength = new Vector();
Vector<String>pseudo = new Vector();
Vector<Integer>ppos = new Vector();
Vector<String>symbolname = new Vector();
Vector<Integer>symbolvalue = new Vector();
Vector<String>format = new Vector();
Vector<String>RA = new Vector();
Vector<Integer>symbollength = new Vector();
Scanner sc = new Scanner(new File("prog.txt"));
String xx = sc.nextLine();
StringTokenizer stt = new StringTokenizer(xx);
Vector<String>readd = new Vector();
while(stt.hasMoreTokens())
readd.add(stt.nextToken());
symbolname.add(readd.elementAt(0));
LC = Integer.parseInt(readd.elementAt(readd.size()-1));
symbolvalue.add(LC);
symbollength.add(1);
RA.add("R");
Vector<String>alreadyread = new Vector();
while(sc.hasNext()){
  String x = sc.nextLine();
  StringTokenizer st = new StringTokenizer(x);
  Vector<String>read = new Vector();
  while(st.hasMoreTokens())
  read.add(st.nextToken());
  for(int i = 0; i<read.size();i++){
    int potpos = getIndex(p,read.elementAt(i));
    if(potpos == -1)continue;
    pseudo.add(p[potpos]);
    ppos.add(potpos+1);
    if(!p[potpos].equals("USING") && !p[potpos].equals("END")){
      symbolname.add(read.elementAt(i-1));
      symbollength.add(l[potpos]);
    }
  }
}
sc = new Scanner(new File("prog.txt"));
String ss = sc.nextLine();
while(sc.hasNext()){
  String x = sc.nextLine();
  StringTokenizer st = new StringTokenizer(x);
  Vector<String>read = new Vector();
  while(st.hasMoreTokens())
  read.add(st.nextToken());
  boolean flag = false;
  for(int i = 0; i<read.size();i++){
    int motpos = getIndex(m,read.elementAt(i));
    if(motpos != -1){
      mneumonic.add(m[motpos]);
      bincode.add(b[motpos]);
```

```
            flag = true;

          }

        }
        if(flag){
          boolean pushed = false;
          String last = read.elementAt(read.size()-1);
          alreadyread.add(last);
          for(int i = 0; i<symbolname.size();i++)
          if(symbolname.elementAt(i).equals(last)){
            format.add("RX");
            pushed = true;
            break;
          }
          if(!pushed)format.add("AX");

        }
      }
      sc = new Scanner(new File("prog.txt"));
      ss = sc.nextLine();
      while(sc.hasNext()){
        String x = sc.nextLine();
        StringTokenizer st = new StringTokenizer(x);
        Vector<String>read = new Vector();
        while(st.hasMoreTokens())
        read.add(st.nextToken());
        for(int i = 0;i<read.size();i++){
          int motpos = getIndex(m,read.elementAt(i));
          if(motpos!=-1){
            LC+=4;
          }
          int potpos = getIndex(p,read.elementAt(i));
          if(potpos!=-1 && !p[potpos].equals("USING") &&
!p[potpos].equals("END")){
            symbolvalue.add(LC);
            if(!p[potpos].equals("EQU"))
            LC+=4;
            continue;
          }

        }
      }
      for(int i=0;i<symbolname.size();i++){
        String ii = symbolname.elementAt(i);
        boolean there = false;
        for(int j=0;j<alreadyread.size();j++)
        if(ii.equals(alreadyread.elementAt(j))){
          there = true;
          break;
        }
```

```
        RA.add(there?"R":"A");
      }
      System.out.println("MOT");
      System.out.println("Mneumonic   Binopcode   length   format");
      for(int i=0;i<mneumonic.size();i++){
        System.out.printf("%s      \t%s   \t%d   \t%s\n",
mneumonic.elementAt(i),bincode.elementAt(i),4,format.elementAt(i));
      }
      System.out.println("\nPOT");
      System.out.println("Pseudo-Opcode   Address");
      for(int i=0;i<pseudo.size();i++){
        System.out.printf("%s   \t\t%d\n", pseudo.elementAt(i),ppos.elementAt(i));
      }
      System.out.println("\nST");
      System.out.println("Name   Counter   Length   R/A");
      for(int i=0;i<symbolname.size();i++){
        System.out.printf("%s \t%d\t%d\t%s\n",
symbolname.elementAt(i),symbolvalue.elementAt(i),symbollength.elementAt(i),RA
.elementAt(i));
      }
   }
   static int getIndex(String st[], String s){
      for(int i=0;i<st.length;i++)
        if(s.equals(st[i]))
          return i;
      return -1;

   }
}
```

(prog.txt)

```
JOHN START 0
   USING * 15
   L 1 FIVE
   A 1 FOUR
   ST 1 TEMP
FOUR DC F 4
FIFTY EQU 3
FIVE DC F'5'
TEMP DS IF
```

OUTPUT:

```
Vivek Hotti - 37
MOT
Mneumonic     Binopcode    length    format
L                1001             4        RX
ST               1111             4        RX

POT
Pseudo-Opcode     Address
USING                      4
DC               1
EQU              2
DC               1
DS               3

ST
Name     Counter     Length     R/A
JOHN     0        1        R
FOUR     8        4        A
FIFTY    12       1        A
FIVE     12       4        A
TEMP     16       4        R


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

CONCLUSION:

A program for PASS 1 Assembler (Generate table POT MOT ST LT BT) has been implemented.

DISCUSSION:

1. Forward reference problem
Ans: -

1. When a symbol is referred before it is defined, it is called a forward reference.
The function of the assembler is to replace each symbol by its machine address and if we refer to that symbol before it is defined its address is not known by the assembler. Such a problem is called a forward reference problem.

2. Forward reference is solved by making different passes (i.e. One Pass, Two Pass, and MultiPass) over the assembly code. Problem of Forward Reference-When the variables are used before their definition at that time problem of forward reference accurse.

x-x-x

Experiment 07 Over

EXPERIMENT – 08

AIM: To implement Program for Macro Processor (Generate Table MDT MNT ALA).

THEORY:

1) ALA :
Argument List Array stores arguments along with index.
Table format:

Argument List Array

8 bytes per entry

Index	*Argument*
0	"LOOP1*bbb*"
1	"DATA1*bbb*"
2	"DATA2*bbb*"
3	"DATA3*bbb*"

(*b* denotes the blank character)

2) MDT :
MDT or Macro Definition table is a table used to store each line of the macro definition except the MACRO line, is stored in the MDT. MEND is also kept to indicate the end of the definition.
Table format:

Macro Definition Table

80 bytes per entry

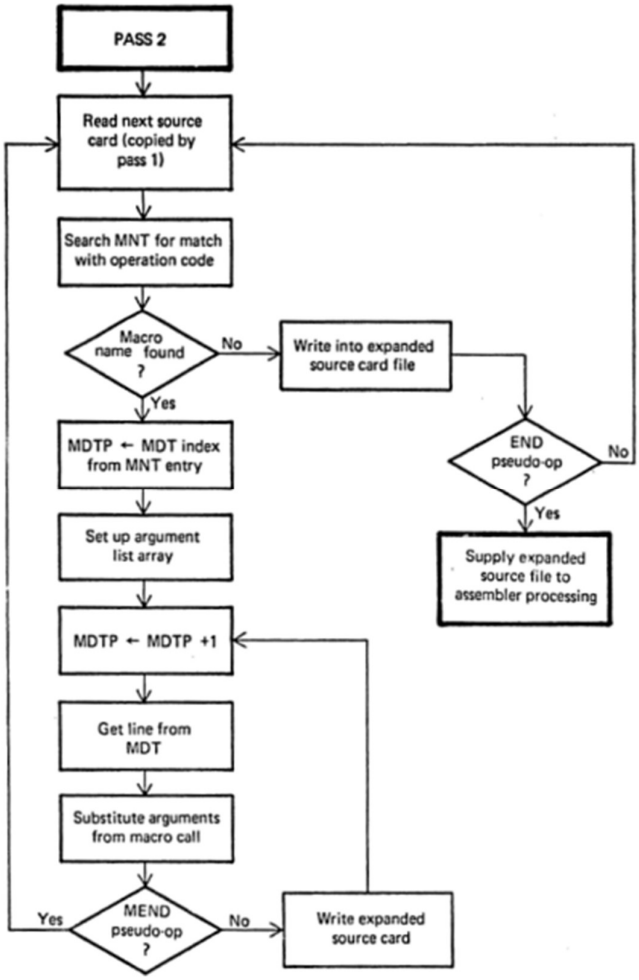
Index	*Card*		
⋮		⋮	
15	&LAB	INCR	&ARG1,&ARG2,&ARG3
16	#0	A	1, #1
17		A	2, #2
18		A	3, #3
19		MEND	
⋮		⋮	

3) MNT :

MNT or Macro Name table indicates entry contains a character string (the macro name) and a pointer (index) to the entry in MDT that corresponds to the beginning of the macro definition.
Table format:

	8 bytes	4 bytes
Index	*Name*	*MDT index*
3	"INCR*bbbb*"	15

Flowchart for PASS 2 Microprocessor



INPUT:

(main.java)

```
#include<stdio.h>
#include<conio.h> #include<string.h>
structmdt
{ char lab[10];
char
opc[10]; char oper[10];
}d[10];
void main()
{ char label[10],opcode[10],operand[10],newlabel[10],newoperand[10]; char
macroname[10]; inti,lines;
FILE *f1,*f2,*f3;
clrscr();
f1 = fopen("MACIN.txt","r"); f2 =
fopen("MACOUT.txt","w"); f3 = fopen("MDT.txt","w");
fscanf(f1,"%s %s %s",label,opcode,operand);
while(strcmp(opcode,"END")!=0)
{ if(strcmp(opcode,"MACRO")==0)
{ strcpy(macroname,label);
fscanf(f1,"%s%s%s",label,opcode,operand); lines
= 0; while(strcmp(opcode,"MEND")!=0)
{ fprintf(f3,"%s\t%s\t%s\n",label,opcode,operand);
strcpy(d[lines].lab,label);
strcpy(d[lines].opc,opcode);
strcpy(d[lines].oper,operand); fscanf(f1,"%s %s
%s",label,opcode,operand);
lines++; }
}
else if(strcmp(opcode,macroname)==0)
{
printf("lines=%d\n",lin
es); for(i=0;i<lines;i++)
{ fprintf(f2,"%s\t%s\t%s\n",d[i].lab,d[i].opc,d[i].oper);
printf("DLAB=%s\nDOPC=%\nDOPER=%s\n",d[i].lab,d[i].opc,d[i].oper);
}}
else fprintf(f2,"%s\t%s\t%s\n",label,opcode,operand);
fscanf(f1,"%s%s%s",label,opcode,operand);
} fprintf(f2,"%s\t%s\t%s\n",label,opcode,operand);
fclose(f1); fclose(f2); fclose(f3); printf("FINISHED");
getch();
}
```

FINAL OUTPUT:

```
1   CALC     START   1000
2   **   LDA LENGTH
3   **   COMP    ZERO
4   **   JEQ LOOP
5   **   LDA #5
6   **   ADD #10
7   **   sTA 2000
8   LENGTH   WORD    S
9   ZERO     WORD    S
10  **   LDA #5
11  **   ADD #10
12  **   sTA 2000
13  **   END **
```

CONCLUSION:

Hence, a program for Macro Processor (Generate Table MDT MNT ALA)

DISCUSSION:

1. Features of macro processor

1.Compile-and-Go Loaders: A compile and go loader is one in which the assembler itself does the processes of compiling then place the assembled instruction in the designated memory locations. The assembly process is first executed and then the assembler causes a transfer to the first instruction of the program.

2.Absolute loaders: In this scheme the assembler outputs the machine language translation of the source program in almost the same form as in the "Compile and go", except that the data is punched on cards. Here it will be directly placed in memory.

3.Bootstrap loader: A bootstrap 54 loader is a program that resides in the computer's EPROM, ROM, or another non-volatile memory. When the computer is turned on or restarted, the bootstrap loader first performs the power-on self-test, also known as POST

4.Relocating loader (relative loader): A loader in which some of the addresses in the program to be loaded are expressed relative to the start of the program rather than in absolute form.

5.Direct linking loader: This scheme has an advantage that it allows the programmer to use multiple procedures and multiple data segments.

6.General Loader: In Compile-and-Go the outputting instruction and data are assembled. In which assembler is placed in main memory that results in wastage of memory. To overcome that we require the addition of the new program of the system, a loader. Generally the size of the loader is less than that of the assembler.

x-x-x

Experiment 08 Over

((((( ))))))

(You have reached the end of the SPCC Lab Manual)