

METHODIST COLLEGE OF ENGINEERING AND TECHNOLOGY**Affiliated to Osmania University, Hyderabad)****King Koti Road, Abids, Hyderabad****COMPILER DESIGN LAB [PC631CS]****LAB MANUAL***For***B.E - VI Semester****Prepared By****Mr. A. RAJESH,****Assistant Professor, Dept. of CSE***Name:**Roll. No:**Academic Year:***DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**



Estd : 2008

METHODIST

COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION

To become a leader in providing Computer Science and Engineering education with emphasis on knowledge and innovation.

MISSION

- To offer flexible programs of study with collaborations to suit industry needs.
- To provide quality education and training through novel pedagogical practices.
- To expedite high performance of excellence in teaching, research and innovations.
- To impart moral, ethical values and education with social responsibility.





Estd : 2008

METHODIST**COLLEGE OF ENGINEERING & TECHNOLOGY****DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING****PROGRAM OUTCOMES (POs)**

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

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

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

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

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

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

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

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

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

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

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

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

PROGRAM SPECIFIC OUTCOMES (PSOs)**PSO1:**

Apply the knowledge of Computer Science and Engineering in various domains like networking and data mining to manage projects in multidisciplinary environments.

PSO2:

Develop software applications with open-ended programming environments.

PSO3:

Design and develop solutions by following standard software engineering principles and implement by using suitable programming languages and platforms.



Estd : 2008

METHODIST

COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

After 3-5 years of graduation, the graduates will be able to

PEO1:

Apply technical concepts, analyze, synthesize data to design and create novel products and solutions for the real life problems.

PEO2:

Apply the knowledge of Computer Science and Engineering to pursue higher education with due consideration to environment and society.

PEO3:

Promote collaborative learning and spirit of team work through multidisciplinary projects.

PEO4:

Engage in life-long learning and develop entrepreneurial skills.



University framed Laboratory Syllabus:

With effect from the academic year 2020-21

Course Code	Course Title					Core/ Elective	
PC 631 CS	COMPILER DESIGN LAB					CORE	
Prerequisite	Contact Hours Per Week				CIE	SEE	Credits
	L	T	D	P			
-	-	-	-	2	25	50	2
Course Objectives <ul style="list-style-type: none"> ➤ To learn usage of tools LEX, YACC ➤ To develop a code generator ➤ To implement different code optimization schemes Course Outcomes <ul style="list-style-type: none"> ➤ Generate scanner and parser from formal specification. ➤ Generate top down and bottom up parsing tables using Predictive parsing, SLR and LR Parsing techniques. ➤ Apply the knowledge of YACC to syntax directed translations for generating intermediate code – 3 address code. ➤ Build a code generator using different intermediate codes and optimize the target code. 							

List of Experiments to be performed:

1. Sample programs using LEX.
2. Scanner Generation using LEX.
3. Elimination of Left Recursion in a grammar.
4. Left Factoring a grammar.
5. Top down parsers.
6. Bottom up parsers.
7. Parser Generation using YACC.
8. Intermediate Code Generation.
9. Target Code Generation.
10. Code optimization.



METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

Approved by AICTE New Delhi | Affiliated to Osmania University, Hyderabad

Estd : 2008 Address : King Koti Road, Abids, Hyderabad, Telangana, 500001 | Email : principal@methodist.edu.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Outcomes (CO's):

SUBJECT NAME: COMPILER DESIGN LAB

CODE: PC631CS

SEMESTER: VI

CO No.	Course Outcome	Taxonomy Level
PC631CS.1	Practise the usage of LEX tool with regular expressions.	L3 - Apply
PC631CS.2	Generate scanner and parser from formal specification.	L2 - Understand
PC631CS.3	Generate top down and bottom up parsing tables using Predictive parsing, SLR and LR Parsing techniques.	L2 - Understand
PC631CS.4	Evaluate the efficiency of Symbol table implementation.	L6 - Evaluate
PC631CS.5	Apply the knowledge of YACC to syntax directed translations for generating intermediate code – 3 address code.	L3 - Apply
PC631CS.6	Build a code generator using different intermediate codes and optimize the target code.	L2 - Understand



METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

Approved by AICTE New Delhi | Affiliated to Osmania University, Hyderabad

Estd : 2008 Address : King Koti Road, Abids, Hyderabad, Telangana, 500001 | Email : principal@methodist.edu.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

List of Experiments

Sl.No.	Name of the Experiment	Date of Experiment	Date of Submission	Page No	Faculty Signature
1.	Basic LEX Programs			01-04	
2.	Write a LEX program to implement standalone			05-08	
3.	Write a C/C++ program to eliminate Left recursion.			09-10	
4.	Write a C/C++ program to eliminate Left factoring.			11-12	
5..	Write a C/C++ program to construct predictive parsing table			13-15	
4.	Write a C / C++ program for SLR parser table			16-21	
6.	Write a C/C++ program for LR Parser table generation			22-24	
7.	Write a program to implement parser using YACC			25-27	
	To write a program for implementing a calculator for computing the given expression using semantic rules of the YACC tool.			28-29	
8.	Write a C/C++ program on code generation			30-32	
9.	Write a C/C++ program on code optimization			33-37	



METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

Approved by AICTE New Delhi | Affiliated to Osmania University, Hyderabad

Estd : 2008 Address : King Koti Road, Abids, Hyderabad, Telangana, 500001 | Email : principal@methodist.edu.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ADDITIONAL EXPERIMENTS

Sl.No.	Name of the Experiment	Date of Experiment	Date of Submission	Page No	Faculty Signature
10.	Write a C/C++ program to construct DFA from NFA			38-44	
11.	Write a C/C++ program to implement recursive descent parsing			45-47	
12.	Write a C/C++ program to find FIRST and FOLLOW for the given grammar			48-51	
13.	Write a C/C++ program to implement Symbol table using Hashing.			52-55	

Signature of the Faculty

HoD – CSE.

PROGRAM 1: Sample LEX Programs**A] Aim: Program to identify octal or hexadecimal number using LEX**

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

Expected Output:

```
./a.out  
o5  
this is an octal number  
ox23  
this is a hexadecimal number
```

B] Aim :Program to capitalize the given comments using LEX

```
%{
#include<stdio.h>
#include<ctype.h>
int k;
void display(char *);
%}
letter [a-z]
com [//]
%%
{com} {k=1;}
{letter} {if(k==1) display(yytext);}
%%
main()
{
  yylex();
}
void display(char *s)
{
  int i;
  for(i=0;s[i]!='\0';i++)
    printf("%c", toupper(s[i]));
}
int yywrap()
{
  return 1;
}
```

Expected Output:

```
lex caplex.l
cc lex.yy.c
./a.out
//hello world
HELLO WORLD
```

C] Aim: Program to find complete real precision using LEX

```
%{
    /*Program to find complete real precision using LEX*/
}%
integer ([0-9]+)
float    ([0-9]+\.[0-9]+)|([+|-]?[0-9]+\.[0-9]*[e|E][+|-][0-9]*)
%%
{integer} printf("\n %s is an integer\n",yytext);
{float}   printf("\n %s is a floating number\n",yytext);
%%
main()
{
yylex();
}
int yywrap()
{
return 1;
}
```

Expected Output:

```
lex real.l
gcc lex.yy.c
./a.out 1234
1234 is an integer
```

D] Aim: Lex Program to classify tokens as words

```
%{
    int tokenCount =0;
}%
%%
[a-zA-Z]+ {printf("%d WORD\\'%s\\'\\n",++tokenCount,yytext); }
[0-9]+ {printf("%dNUMBER\\'%s\\'\\n",++tokenCount,yytext); }
[^a-zA-Z0-9]+ {printf("%dOTHER\\'%s\\'\\n", ++tokenCount,yytext); }
%%

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

Expected Output:

Input:

Hello! World ...this is 21 st century

OUTPUT:

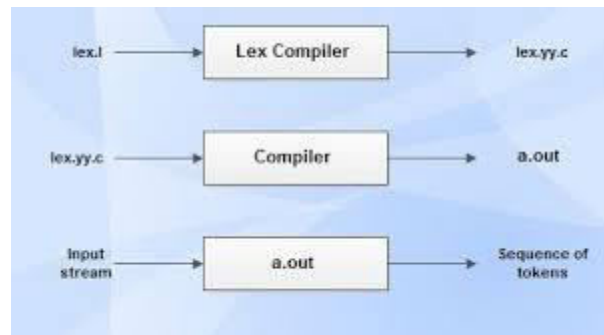
- 1.WORD Hello
- 2.OTHER !
- 3.WORD World
- 4.OTHER ...
- 5.WORD this
- 6.WORD is
- 7.NUMBER 21
- 8.WORD st century

PROGRAM 2: Scanner program using LEX

Aim: Write a LEX program to implement standalone scanner

Description:

Lex is a popular scanner (lexical analyzer) generator developed by M.E. Lesk and E. Schmidt of AT&T Bell Labs . Other versions of Lex exist, most notably flex (for Fast Lex) . Input to Lex is called Lex specification or Lex program . Lex generates a scanner module in C from a Lex specification file . Scanner module can be compiled and linked with other C/C++ modules.



Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations

%%

translation rules

%%

auxiliary procedures

The definition section contains a literal block and regular definitions

The literal block is C code delimited by %{ and %} and contains variable declarations and function prototypes

A regular definition gives a name to a regular expression

A regular definition has the form: name expression

A regular definition can be used by writing its name in braces: {name}

The rules section contains regular expressions and C code.

It has the form:

r 1 action 1

r 2 action 2 . . .

r n action n • r

Lex Operators

\	C escape sequence \n is newline, \t is tab, \\ is backslash, \" is double quote, etc.
*	Matches zero or more of the preceding expression: x* matches , x, xx, ...
+	Matches one or more of the preceding expression: (ab)+ matches ab, abab, ababab, ...
?	Matches zero or one occurrence of the preceding expression: (ab)? Matches or ab
	Matches the preceding or the subsequent expression: a b matches

	a or b
()	Used for grouping sub-expressions in a regular expression
[]	Matches any one of the characters within brackets
.	Matches any single character except the newline character
“ ”	Matches everything within the quotation marks literally "x*" matches exactly x* Meta-characters, other than \, loose their meaning inside " " C escape sequences retain their meaning inside " "
{ }	{name} refers to a regular definition from the first section [A-Z]{3} matches strings of exactly 3 capital letters [A-Z]{1,3} matches strings of 1, 2, or 3 capital letters
/	The lookahead operator matches the left expression but only if followed by the right expression 0/1 matches 0 in 01, but not in 02 Only one slash is permitted per regular expression
^	As first character of a regular expression, ^ matches beginning of a line
\$	As last character of a regular expression, \$ matches end of a line Same as /\n

Algorithm:

1. Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

The format is as follows:

definitions

%%

rules

%%

user_subroutines

2. In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric.
3. In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.
4. Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.
5. When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.
6. In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.

7. The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

Program:

```
%{
    int COMMENT=0;
}%

id [a-z][a-z0-9]*

%%
#.*          {printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int|double|char {printf("\n\t%s is a KEYWORD",yytext);}
if|then|endif {printf("\n\t%s is a KEYWORD",yytext);}
else          {printf("\n\t%s is a KEYWORD",yytext);}
"/*"         {COMMENT=1;}
"*/"         {COMMENT=0;}
{id}\(       {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
{id}\(\\[[0-9]*\\)? {if(!COMMENT) printf("\n\tidentifier\t%s",yytext);}
\{           {if(!COMMENT) printf("\n BLOCK BEGINS");ECHO; }
\}           {if(!COMMENT)printf("\n BLOCK ends");ECHO; }
\".*\"       {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[+|-]?[0-9]+ {if(!COMMENT)printf("\n\t%s is a NUMBER",yytext);}
\(           {if(!COMMENT)printf("\n\t");ECHO;printf("\t delim
                                openparanthesis\n");}
\)           {if(!COMMENT)printf("\n\t");ECHO;printf("\t delim closed
                                paranthesis");}
\;           {if(!COMMENT)printf("\n\t");ECHO;printf("\t delim semicolon");}
\=           {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT
                                OPERATOR",yytext);}
\<|>        {printf("\n\t %s is relational operator",yytext);}
"+"|"-"|"*"|"\/" {printf("\n %s is an operator\n",yytext);}
"\n" ;
%%

main(int argc ,char **argv)
{
    if (argc > 1)
        yyin = fopen(argv[1],"r");
    else
        yyin = stdin;
    yylex ();
    printf ("\n");
}
int yywrap()
```

```
{  
    return 0;  
}
```

Expected Output:

1. Save the file with .l extension\
2. Create a text file for eg: input.txt and write #include<stdio.h> , int

```
lex lexscanner.l  
cc lex.yy.c  
./a.out input.txt
```

#include<stdio.h> is a PREPROCESSOR DIRECTIVE
int is a KEYWORD

Result: The above scanner program using LEX was successfully executed

PROGRAM 3: Elimination of Left Recursion in a grammar.

Aim: Write a C/C++ program to eliminate Left recursion.

Description:

Left Recursion: The generation is left-recursive if the leftmost symbol on the right side is equivalent to the non-terminal on the left side. **For Ex:** $\text{exp} \rightarrow \text{exp} + \text{term}$.

A grammar that contains a production having left recursion is called as a Left-Recursive Grammar. Similarly, if the rightmost symbol on the right side is equal to left side is called Right-Recursion.

Consider, $E \rightarrow E + T$

$E = a$

$T = b$

In its parse tree E will grow left indefinitely, so to remove it

$E = Ea \mid b$

we take as

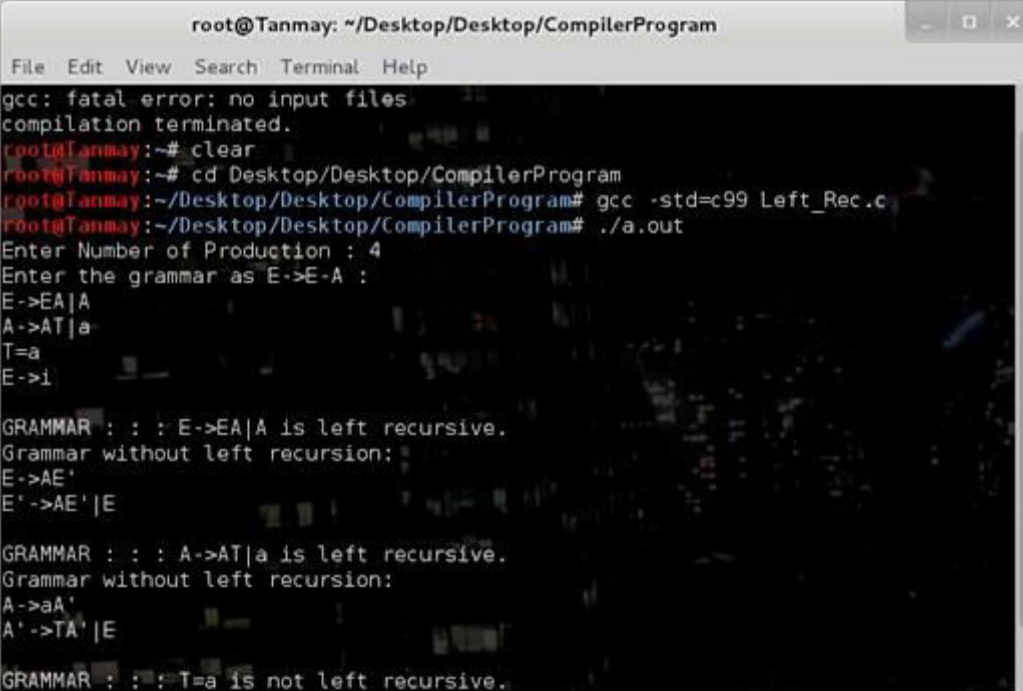
$E = bE'$

$E' = aE' \mid \epsilon$

Program:

```
1: #include<stdio.h>
2: #include<string.h>
3: #define SIZE 10
4: int main () {
5:     char non_terminal;
6:     char beta,alpha;
7:     int num;
8:     char production[10][SIZE];
9:     int index=3; /* starting of the string following "->" */
10:    printf("Enter Number of Production : ");
11:    scanf("%d",&num);
12:    printf("Enter the grammar as E->E-A :\n");
13:    for(int i=0;i<num;i++){
14:        scanf("%s",production[i]);
15:    }
16:    for(int i=0;i<num;i++){
17:        printf("\nGRAMMAR : : : %s",production[i]);
18:        non_terminal=production[i][0];
19:        if(non_terminal==production[i][index]) {
20:            alpha=production[i][index+1];
21:            printf(" is left recursive.\n");
22:            while(production[i][index]!=0 && production[i][index]!='|')
23:                index++;
24:            if(production[i][index]!=0) {
25:                beta=production[i][index+1];
26:                printf("Grammar without left recursion:\n");
```

```
27:         printf("%c->%c%c\\",non_terminal,beta,non_terminal);
28:         printf("\\n%c\\'->%c%c\\'|E\\n",non_terminal,alpha,non_terminal);
29:     }
30:     else
31:         printf(" can't be reduced\\n");
32:     }
33:     else
34:         printf(" is not left recursive.\\n");
35:     index=3;
36: }
37: }
```

Expected Output:

The screenshot shows a terminal window titled 'root@Tanmay: ~/Desktop/Desktop/CompilerProgram'. The user enters 'gcc: fatal error: no input files' and 'compilation terminated.' followed by '# clear'. Then, the user navigates to the directory and compiles 'Left_Rec.c' with 'gcc -std=c99 Left_Rec.c'. The program runs, asking for the number of productions (4) and the grammar rules: 'E->EA|A', 'A->AT|a', 'T=a', and 'E->i'. The program then checks for left recursion. It reports that 'E->EA|A' is left recursive and provides the transformed rule 'E'->AE'|E'. It then checks 'A->AT|a' and reports it is left recursive, providing the transformed rule 'A'->aA'' and 'A'->TA'|E'. Finally, it reports that 'T=a' is not left recursive.

```
root@Tanmay: ~/Desktop/Desktop/CompilerProgram
File Edit View Search Terminal Help
gcc: fatal error: no input files
compilation terminated.
root@Tanmay:~# clear
root@Tanmay:~# cd Desktop/Desktop/CompilerProgram
root@Tanmay:~/Desktop/Desktop/CompilerProgram# gcc -std=c99 Left_Rec.c
root@Tanmay:~/Desktop/Desktop/CompilerProgram# ./a.out
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.
```

Result : The above c-program was successfully executed.

PROGRAM 4: Elimination of Left Factoring

Aim: Write a C/C++ program to eliminate Left factoring.

Description:

Left Factoring is basically a grammar transformation technique. It has "factoring out" prefixes which are common to two or more productions or in other words Left factoring is a process of transformation, in which the grammar turns from a left-recursive form to an equivalent non-left-recursive form.

Left factoring is taking out the regular left factor that shows up in two productions of the equivalent non-terminal. It is done to keep away from back-tracking by the parser.

Consider an example below

$A \rightarrow \alpha P / \alpha Q$

where A, P, Q are non-terminals and α is a common factor

after left factoring the grammar will be:

$A \rightarrow \alpha S'$

$S' \rightarrow P / Q$

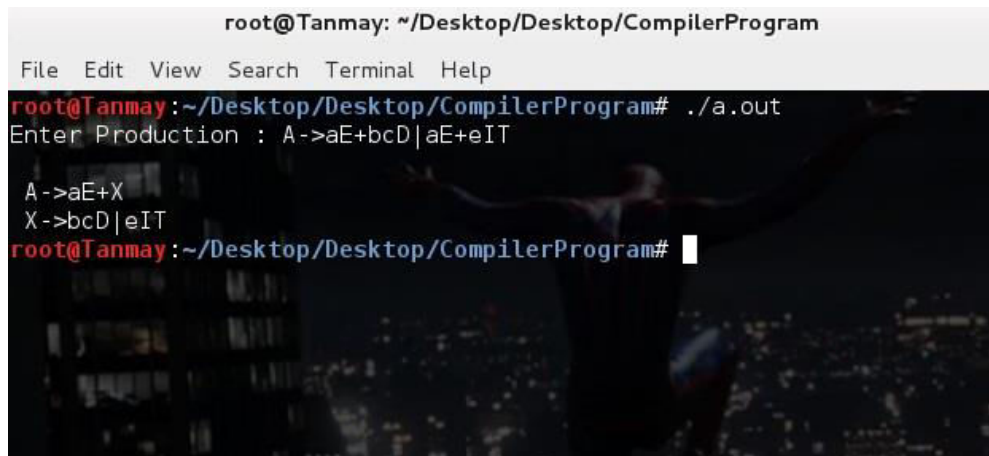
In LL(1) Parser in Top-down parsing, even if a context-free grammar is unambiguous and non-left-recursion it still cannot be a LL(1) Parser. That is because of Left Factoring.

Program:

```
1: #include<stdio.h>
2: #include<string.h>
3: int main()
4: {
5:     char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];
6:     int i,j=0,k=0,l=0,pos;
7:     printf("Enter Production : A->");
8:     gets(gram);
9:     for(i=0;gram[i]!='|';i++,j++)
10:         part1[j]=gram[i];
11:     part1[j]='\0';
12:     for(j=++i,i=0;gram[j]!='\0';j++,i++)
13:         part2[i]=gram[j];
14:     part2[i]='\0';
15:     for(i=0;i<strlen(part1)||i<strlen(part2);i++)
16:     {
17:         if(part1[i]==part2[i])
18:         {
```



```
19:      modifiedGram[k]=part1[i];
20:      k++;
21:      pos=i+1;
22:  }
23: }
24: for(i=pos,j=0;part1[i]!='\0';i++,j++){
25:     newGram[j]=part1[i];
26: }
27: newGram[j++]='|';
28: for(i=pos;part2[i]!='\0';i++,j++){
29:     newGram[j]=part2[i];
30: }
31: modifiedGram[k]='X';
32: modifiedGram[++k]='\0';
33: newGram[j]='\0';
34: printf("\n A->%s",modifiedGram);
35: printf("\n X->%s\n",newGram);
36: }
```

Expected Output:

```
root@Tanmay: ~/Desktop/Desktop/CompilerProgram
File Edit View Search Terminal Help
root@Tanmay:~/Desktop/Desktop/CompilerProgram# ./a.out
Enter Production : A->aE+bcD|aE+eIT
A->aE+X
X->bcD|eIT
root@Tanmay:~/Desktop/Desktop/CompilerProgram#
```

Result : The above c-program was successfully executed.

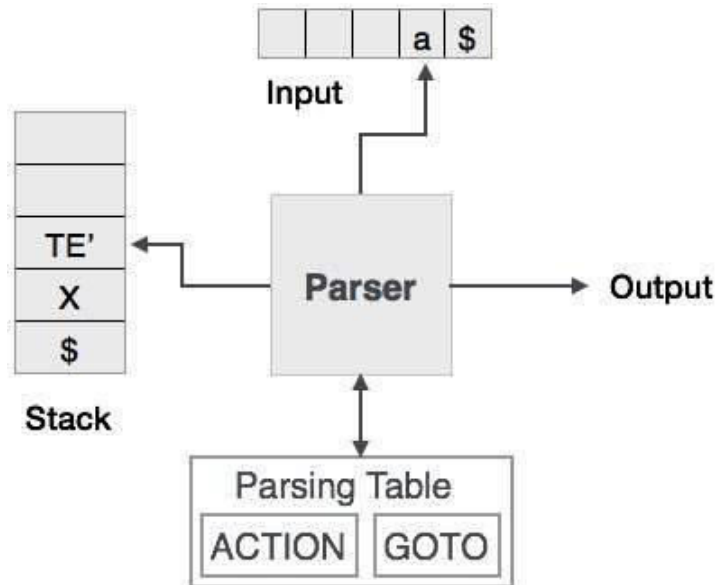
PROGRAM 5: DESIGN of a Predictive Parsing Table

Aim: Write a C/C++ program to construct predictive parsing table

Description :

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol \$ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

Program :

```
#include<stdio.h>
#include<iostream>
#include<string.h>
using namespace std;
char prol[7][10]={"S","A","A","B","B","C","C"};
char pror[7][10]={"A","Bb","Cd","aB","@","Cc","@"};
char prod[7][10]={"S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C->@"}; char
first[7][10]={"abcd","ab","cd","a@","@","c@","@"}; char
follow[7][10]={"$","$","$","a$","b$","c$","d$"};
char table[5][6][10];
int numr(char c)
{
    switch(c)
    {
        case 'S': return 0;
        case 'A': return 1;
```

```

        case 'B': return 2;
        case 'C': return 3;
        case 'a': return 0;
        case 'b': return 1;
        case 'c': return 2;
        case 'd': return 3;
        case '$': return 4;
    }
    return(2);
}
int main(int argc, char *argv[])
{
    int i,j,k;
    for(i=0;i<5;i++)
        for(j=0;j<6;j++)
            strcpy(table[i][j], " ");
    printf("\nThe following is the predictive parsing table for the following grammar:\n");
    for(i=0;i<7;i++)
        printf("%s\n",prod[i]);
    printf("\nPredictive parsing table is\n");
    fflush(stdin);
    for(i=0;i<7;i++)
    {
        k=strlen(first[i]);
        for(j=0;j<10;j++)
            if(first[i][j]!='@')
                strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);
    }
    for(i=0;i<7;i++)
    {
        if(strlen(pror[i])==1)
        {
            if(pror[i][0]=='@')
            {
                k=strlen(follow[i]);
                for(j=0;j<k;j++)
                    strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);
            }
        }
    }
    strcpy(table[0][0], " ");
    strcpy(table[0][1], "a");
    strcpy(table[0][2], "b");
    strcpy(table[0][3], "c");
    strcpy(table[0][4], "d");
    strcpy(table[0][5], "$");
    strcpy(table[1][0], "S");
    strcpy(table[2][0], "A");
    strcpy(table[3][0], "B");
    strcpy(table[4][0], "C");
    printf("\n-----\n");
}

```

```

for(i=0;i<5;i++)
    for(j=0;j<6;j++)
    {
        printf("%-10s",table[i][j]);
        if(j==5)
            printf("\n-----\n");
    }
    system("PAUSE"); // statement in Bloodshed dev c++ IDE requirement
}

```

Expected Output:

```

D:\cc\scanner.exe
The following is the predictive parsing table for the following grammar:
S->A
A->Bb
A->Cd
B->aB
B->ε
C->Cc
C->ε

Predictive parsing table is
-----
      a      b      c      d      $
-----
S      S->A    S->A    S->A    S->A
-----
A      A->Bb    A->Bb    A->Cd    A->Cd
-----
B      B->aB    B->ε     B->ε     B->ε
-----
C              C->ε     C->ε     C->ε
-----
Press any key to continue . . . _

```

Result: The above program for generation of predictive parsing table was successfully executed

PROGRAM 6: SLR Parser table generation

Aim: Write a C / C++ program for SLR parser table generation

Description:

The SLR(simple LR) parser is similar to LR(0) parser except that the reduced entry. The simple LR or SLR parser is a type of LR parser with small parse tables and a relatively simple parser generator algorithm. As with other types of LR(1) parser, an SLR parser is quite efficient at finding the single correct bottom-up parse in a single left-to-right scan over the input stream, without backtracking. The parser is mechanically generated from a formal grammar for the language. The reduced productions are written only in the FOLLOW of the variable whose production is reduced.

Algorithm:

DESIGN of SLR parsing table –

Input : An augmented grammar G'

Output : The SLR parsing table functions action and goto for G'

1. Construct $C = \{ I_0, I_1, \dots, I_n \}$, the collection of sets of LR(0) items for G'.
2. State i is constructed from I_i . The parsing actions for state i are determined as follow :
 - If $[A \rightarrow ?a?]$ is in I_i and $GOTO(I_i, a) = I_j$, then set $ACTION[i, a]$ to "shift j ". Here a must be terminal.
 - If $[A \rightarrow ?.]$ is in I_i , then set $ACTION[i, a]$ to "reduce $A \rightarrow ?$ " for all a in $FOLLOW(A)$; here A may not be S' .
 - If $[S \rightarrow S.]$ is in I_i , then set action $[i, \$]$ to "accept". If any conflicting actions are generated by the above rules we say that the grammar is not SLR.
3. The goto transitions for state i are constructed for all non-terminals A using the rule:
if $GOTO(I_i, A) = I_j$ then $GOTO[i, A] = j$.
4. All entries not defined by rules 2 and 3 are made error.

Program:

```
#include<stdio.h>
#include<string.h>
#include<iostream>
using namespace std;
char a[8][5],b[7][5];
int c[12][5];
int w=0,e=0,x=0,y=0;
int st2[12][2],st3[12];
char sta[12],ch;
void v1(char,int);
void v2(char,int,int,int);
int main(int argc, char **argv[])
{
    int i,j,k,l=0,m=0,p=1,f=0,g,v=0,jj[12];
    printf("\n\n\t*****Enter the Grammar Rules (max=3)*****\n\t");
    for(i=0;i<3;i++)
    {
```

```
        gets(a[i]);
        printf("\t");
    }
    for(i=0;i<3;i++)
    {
        for(j=0;j<strlen(a[i]);j++)
        {
            for(k=0;k<strlen(a[i]);k++)
            {
                if(p==k)
                {
                    b[l][m]='.';
                    m+=1;
                    b[l][m]=a[i][k];
                    m+=1;
                }
                else
                {
                    b[l][m]=a[i][k];
                    m++;
                }
            }
            p++;
            l++;
            m=0;
        }
        p=1;
    }
    i=0; p=0;
    while(l!=i)
    {
        for(j=0;j<strlen(b[i]);j++)
        {
            if(b[i][j]=='.')
            {
                p++;
            }
        }
        if(p==0)
        {
            b[i][strlen(b[i])]='.';
        }
        i++;
        p=0;
    }
    i=0;
    printf("\n\t*****Your States will be*****\n\t");
    while(l!=i)
    {
        printf("%d--> ",i);
        puts(b[i]);
    }
```



```
        i++;
        printf("\t");
    }
    printf("\n");
    v1('A',l);
    p=c[0][0];
    m=0;
    while(m!=6)
    {
        for(i=0;i<st3[m];i++)
        {
            for(j=0;j<strlen(b[p]);j++)
            {
                if(b[p][j]=='.' && ((b[p][j+1]>=65 && b[p][j+1]<=90)
                                   ||(b[p][j+1]>=97&&b[p][j+1]<=122)))
                {
                    st2[x][0]=m;
                    sta[x]=b[p][j+1];
                    v2(b[p][j+1],j,l,f);
                    x++;
                }
                else
                {
                    if(b[p][j]=='.')
                    {
                        st2[x][0]=m;
                        sta[x]='S';
                        st2[x][1]=m;
                        x++;
                    }
                }
            }
            p=c[m][i+1];
        }
        m++;
        p=c[m][0];
    }
    g=0; p=0; m=0;x=0;
    while(p!=11)
    {
        for(i=0;i<st3[p];i++)
        {
            for(k=0;k<p;k++)
            {
                for(j=0;j<3;j++)
                {
                    if(c[k][j]==c[p][j])
                    {
                        m++;
                    }
                }
            }
        }
    }
}
```

```

        }
        if(m==3)
        {
            m=0;
            goto ac;
        }
        m=0;
    }
    if(m!=3)
    {
        if(v==0)
        {
            printf("\tI%d=",g);
            v++;
            jj[g]=p;
        }
        printf("%d",c[p][i]);
    }
}
printf("\n");
g++;
ac:
p++;
v=0;
}
printf("\t*****Your DFA will be *****");
for(i=0;i<9;i++)
{
    printf("\n\t%d",st2[i][0]);
    printf("-->%c",sta[i]);
}
getchar();
}

void v1(char ai,int kk)
{
    int i,j;
    for(i=0;i<kk;i++)
    {
        if(b[i][2]==ai&& b[i][1]=='.')
        {
            c[w][e]=i;
            e++;
            if(b[i][2]>=65 && b[i][2]<=90)
            {
                for(j=0;j<kk;j++)
                {
                    if(b[j][0]==ai && b[j][1]=='.')
                    {
                        c[w][e]=j;
                        e++;
                    }
                }
            }
        }
    }
}

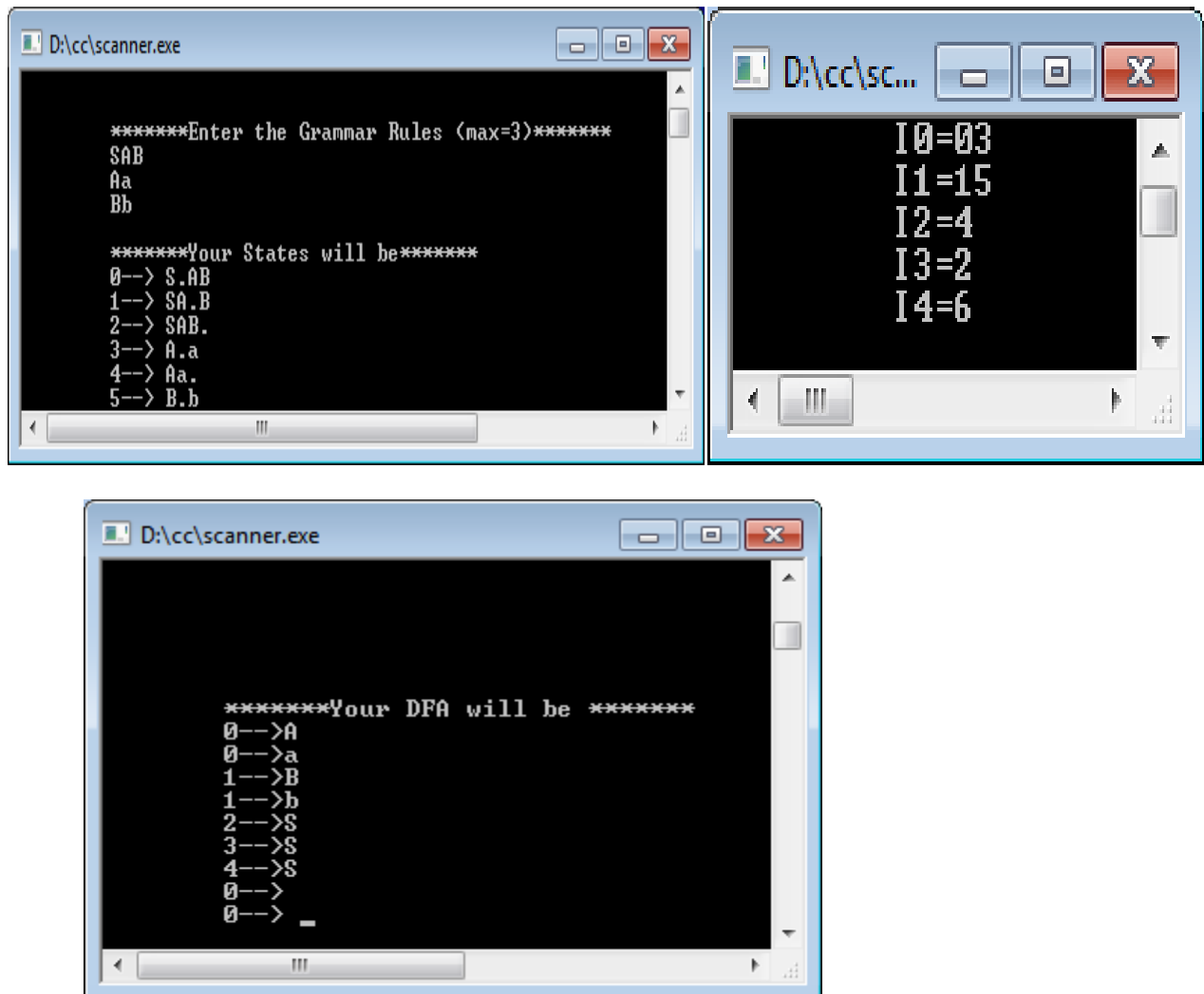
```

```

    }
    }
    }
    }
    }
    st3[w]=e;
    w++;
    e=0;
}
void v2(char ai,int ii,int kk,int tt)
{
    int i,j,k;
    for(i=0;i<kk;i++)
    {
        if(b[i][ii]=='.' && b[i][ii+1]==ai)
        {
            for(j=0;j<kk;j++)
            {
                if(b[j][ii+1]=='.' && b[j][ii]==ai)
                {
                    c[w][e]=j;
                    e++;
                    st2[tt][1]=j;
                    if(b[j][ii+2]>=65 && b[j][ii+1]<=90)
                    {
                        for(k=0;k<kk;k++)
                        {
                            if(b[k][0]==b[j][ii+2] && b[k][1]=='.')
                            {
                                c[w][e]=k;
                                e++;
                            }
                        }
                    }
                }
            }
        }
        if((b[i][ii+1]>=65 && b[i][ii+1]<=90) && tt==1)
        {
            for(k=0;k<kk;k++)
            {
                if(b[k][0]==ai && b[k][1]=='.')
                {
                    c[w][e]=k;
                    e++;
                }
            }
        }
    }
    st3[w]=e;
    w++;
}

```

```
e=0;  
system("PAUSE");  
}
```

Expected Output:

Result : The above program for generation of SLR parsing table was successfully executed

PROGRAM 6: LR Parser table generation

Aim :- Write a C/C++ program for LR Parser table generation

Description:

LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.

In the LR parsing, "L" stands for left-to-right scanning of the input. "R" stands for constructing a right most derivation in reverse. "K" is the number of input symbols of the look ahead used to make number of parsing decision. LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing. The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.

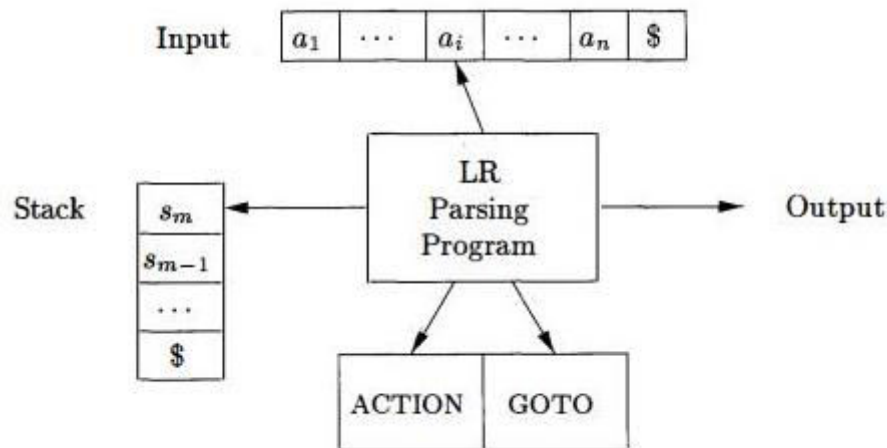


Fig: Block diagram of LR parser

Input buffer is used to indicate end of input and it contains the string to be parsed followed by a \$ Symbol. A stack is used to contain a sequence of grammar symbols with a \$ at the bottom of the stack. Parsing table is a two dimensional array. It contains two parts: Action part and Go To part.

Algorithm:

1. Get the input expression and store it in the input buffer.
2. Read the data from the input buffer one at the time and convert in to corresponding Non Terminal using production rules available.
3. Perform push & pop operation for LR parsing table DESIGN.
4. Display the result with conversion of corresponding input symbols to production and production reduction to start symbol. No operation performed on the operator.

Program:

```

#include<stdio.h>
#include<iostream>
using namespace std;

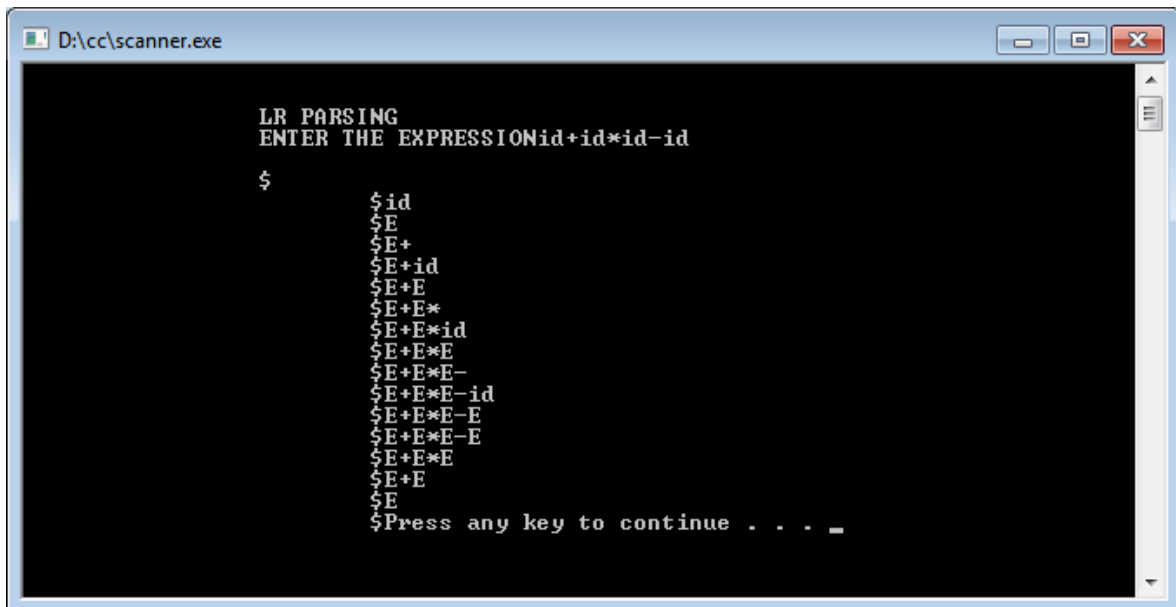
char stack[30];
int top=-1;
void push(char c)

```

```
{
    top++;
    stack[top]=c;
}
char pop()
{
    char c;
    if(top!=-1)
    {
        c=stack[top];
        top--;
        return c;
    }
    return 'x';
}
void printstat()
{
    int i;
    printf("\n\t\t $");
    for(i=0;i<=top;i++)
        printf("%c",stack[i]);
}
int main(int argc, char *argv[])
{
    int i,j,k,l;
    char s1[20],s2[20],ch1,ch2,ch3;
    printf("\n\n\t\t LR PARSING");
    printf("\n\t\t ENTER THE EXPRESSION");
    scanf("%s",s1);
    l=strlen(s1);
    j=0;
    printf("\n\t\t $");
    for(i=0;i<l;i++)
    {
        if(s1[i]=='i' && s1[i+1]=='d')
        {
            s1[i]=' ';
            s1[i+1]='E';
            printstat(); printf("id");
            push('E');
            printstat();
        }
        else if(s1[i]=='+' || s1[i]=='-' || s1[i]=='*' || s1[i]=='/' || s1[i]=='d')
        {
            push(s1[i]);
            printstat();
        }
    }
    printstat();
    l=strlen(s2);
    while(l)
```



```
{
    ch1=pop();
    if(ch1=='x')
    {
        printf("\n\t\t $");
        break;
    }
    if(ch1=='+'||ch1=='/'||ch1=='*'||ch1=='-')
    {
        ch3=pop();
        if(ch3!='E')
        {
            printf("error");
            exit(0);
        }
        else
        {
            push('E');
            printstat();
        }
    }
    ch2=ch1;
}
system("PAUSE");
}
```

Expected Output:

```
D:\cc\scanner.exe

LR PARSING
ENTER THE EXPRESSIONid+id*id-id

$
    $id
    $E
    $E+
    $E+id
    $E+E
    $E+E*
    $E+E*id
    $E+E*E
    $E+E*E-
    $E+E*E-id
    $E+E*E-E
    $E+E*E-E
    $E+E*E
    $E+E
    $E
    $Press any key to continue . . . _
```

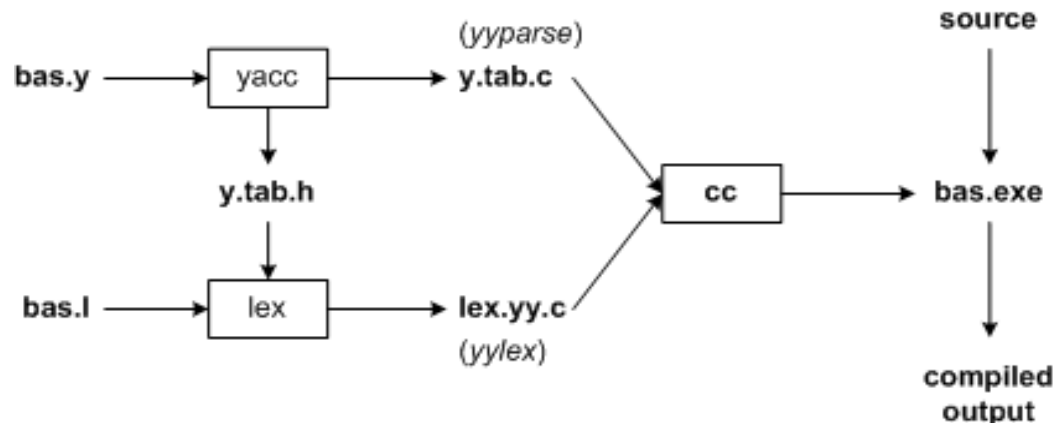
Result : The above program for implementing LR parser table was successfully executed

PROGRAM 7: Parser Generation using YACC

Aim: A] Write a program to implement parser using YACC

Description:

Yacc generates C code for a syntax analyzer, or parser. Yacc uses grammar rules that allow it to analyze tokens from lex and create a syntax tree. A syntax tree imposes a hierarchical structure on tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly.



```

yacc -d bas.y           # create y.tab.h, y.tab.c
lex bas.l               # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe # compile/link
  
```

Yacc reads the grammar descriptions in `bas.y` and generates a parser, function `yyparse`, in file `y.tab.c`. Included in file `bas.y` are token declarations. These are converted to constant definitions by yacc and placed in file `y.tab.h`. Lex reads the pattern descriptions in `bas.l`, includes file `y.tab.h`, and generates a lexical analyzer, function `yylex`, in file `lex.yy.c`. Finally, the lexer and parser are compiled and linked together to form the executable, `bas.exe`. From main, we call `yyparse` to run the compiler. Function `yyparse` automatically calls `yylex` to obtain each token.

1. A Yacc source program has three parts as follows:

```

Declarations
%%
translation rules
%%
supporting C routines
  
```

2. Declarations Section:

This section contains entries that:

- i. Include standard I/O header file.
- ii. Define global variables.
- iii. Define the list rule as the place to start processing.

- iv. Define the tokens used by the parser.
- v. Define the operators and their precedence.

3. Rules Section:

The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

4. Programs Section:

The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.

- 5. Main- The required main program that calls the yyparse subroutine to start the program.
- 6. yyerror(s) -This error-handling subroutine only prints a syntax error message.
- 7. yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmer file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.
- 8. Lex file contains the rules to generate these tokens from the input stream.

Program:

FILE 1: parser.l

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

FILE 2 :parser.y

```
%token NUM  
%%  
cmd :E {printf("%d\n",$1);}  
;  
E :E '+' T {$$=$1+$3;}  
|T {$$=$1;}
```

```

    T      ;
          :T'*'F {$$=$1*$3;}
          |F {$$=$1;}
          ;
    F      : '(' E ')' {$$=$2;}
          | NUM {$$=$1;}
          ;
%%
```

```
int main()
{
    yyparse();
}
yyerror(char *s)
{
    printf("%s",s);
}
```

Expected Output:

```
lex parser.l
yacc -d parser.y
gcc lex.yy.c y.tab.c -ll -ly
./a.out
2+3
5
```

Result: The above program generation of parser using YACC was successfully executed

Aim: B] To write a program for implementing a calculator for computing the given expression using semantic rules of the YACC tool.

Program:

FILE 1 : cal.l

```
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+ {yylval.dval=atoi(yytext); return DIGIT;}
\n|.  return yytext[0];
%%
```

FILE 2 :Cal.y

```
%{
/* */
%}
%union
{
int dval;
}
%token <dval> DIGIT
%type <dval> expr
%type <dval> expr1

%%
line : expr '\n' {printf("%d\n", $1);}
    ;
expr : expr '+' expr1 {$$=$1+$3;}
    | expr '-' expr1 {$$=$1-$3;}
    | expr '*' expr1 {$$=$1*$3;}
    | expr '/' expr1 {$$=$1/$3;}
    | expr1
    ;
expr1 : '(' expr ')' {$$=$2;}
    | DIGIT
    ;

%%
int main()
{
    yyparse();
}
yyerror(char *s)
{
    printf("%s", s);
}
```

Expected Output:

```
$ lex cal.l
$ yacc -d cal.y
$ gcc lex.yy.c y.tab.c -ll
$ ./a.out
1+2
3
```

Result : The above programs for implementing calculator using YACC was successfully executed

Program 8: Write a program on code generation

Aim: Write a C/C++ program on code generation

Description: Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.

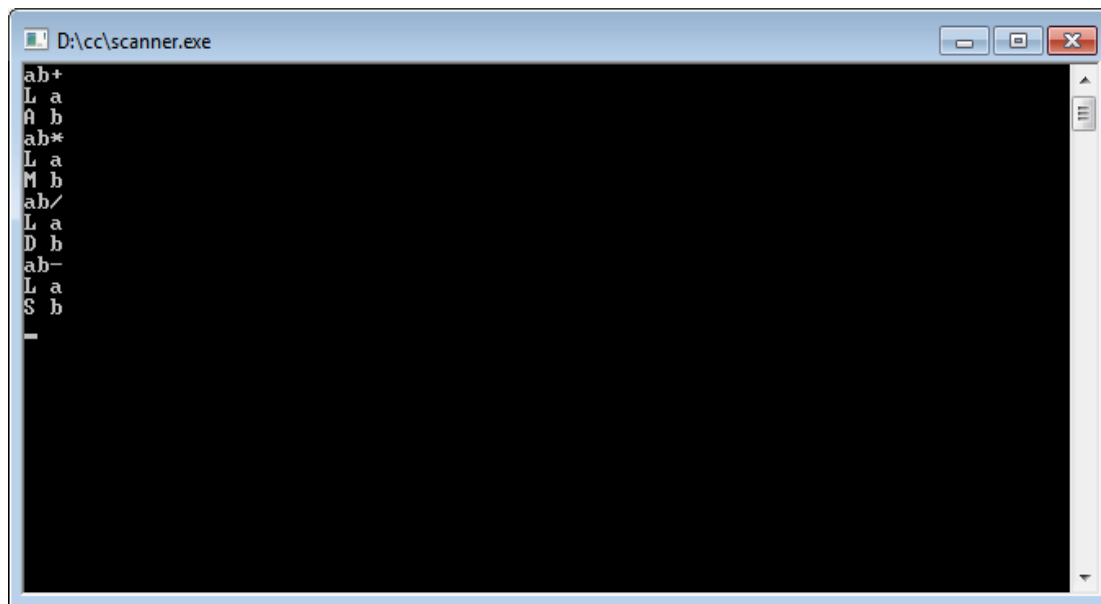
Properties of code optimization:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

Program:

```
#include<stdio.h>
#include<iostream>
using namespace std;
char stk[100],stktop=-1,cnt=0;
void push(char pchar)
{
    stk[++stktop]=pchar;
}
char pop()
{
    return stk[stktop--];
}
char checkoperation(char char1)
{
    char oper;
    if(char1=='+')
        oper='A';
    else if(char1=='-')
        oper='S';
    else if(char1=='*')
        oper='M';
    else if(char1=='/')
        oper='D';
    else if(char1=='@')
        oper='N';
    return oper;
}
int checknstore(char check)
{
    int ret;
    if(check!='+' && check!='-' && check!='*' && check!='/' && check!='@')
    {
        push(++cnt);
        if(stktop>0)
            printf("ST $%d\n",cnt);
        ret=1;
    }
```

```
    }
    else
    ret=0;
    return ret;
}
int main(int argc, char *argv[])
{
    char msg[100],op1,op2,operation;
    int i,val;
    while(scanf("%s",msg)!=EOF)
    {
        cnt=0;
        stktop=-1;
        for(i=0;msg[i]!='\0';i++)
        {
            if((msg[i] >='A' && msg[i]<='Z') ||(msg[i]>='a' && msg[i]<='z'))
                push(msg[i]);
            else
            {
                op1=pop();
                op2=pop();
                printf("L %c\n",op2);
                operation=checkoperation(msg[i]);
                printf("%c %c\n",operation,op1);
                val=checknstore(msg[i+1]);
                while(val==0)
                {
                    op1=pop();
                    cnt--;
                    operation=checkoperation(msg[++i]);
                    if(operation=='S'&&stktop>=-1)
                    {
                        printf("N\n");
                        operation='A';
                    }
                    printf("%c %c\n",operation,op1);
                    val=checknstore(msg[i+1]);
                }
            }
        }
    }
    system("PAUSE");
}
```


Expected Output:

```
D:\cc\scanner.exe
ab+
L a
a h
ab*
L a
M h
ab/
L a
D h
ab-
L a
S b
-

```

Result: The above program for code generation was successfully executed.

Program 9: Write a program on code optimization

Aim: Write a C/C++ program on code optimization

Description:

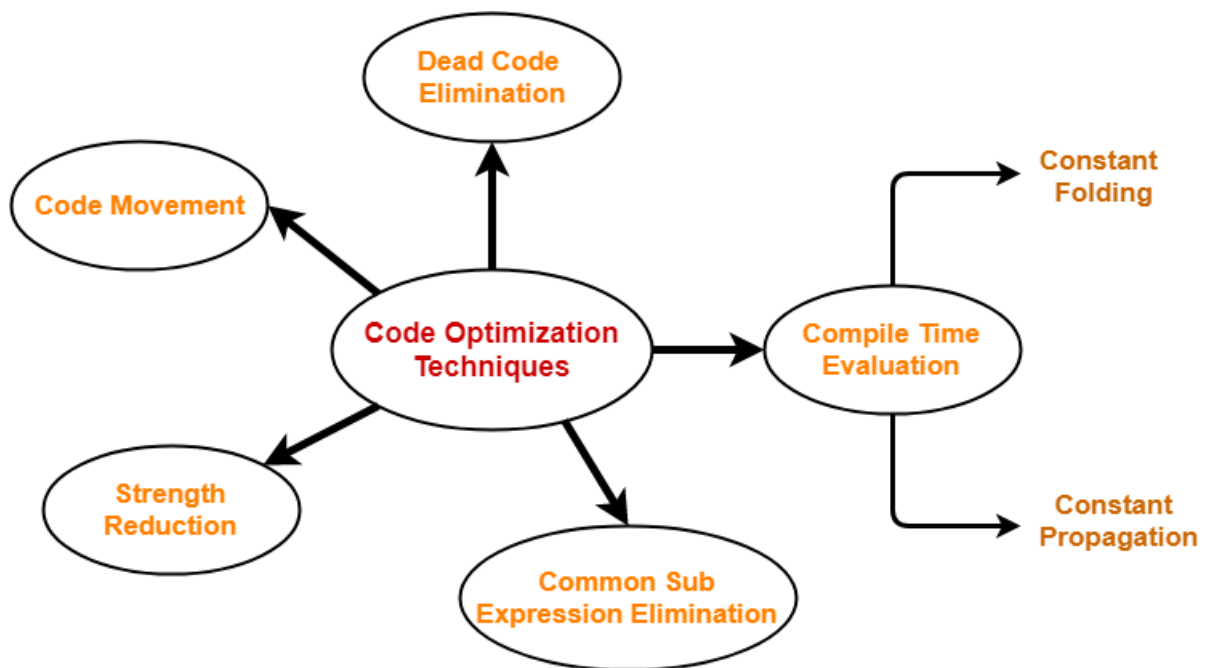
Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed. In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.

A code optimizing process must follow the three rules given below:

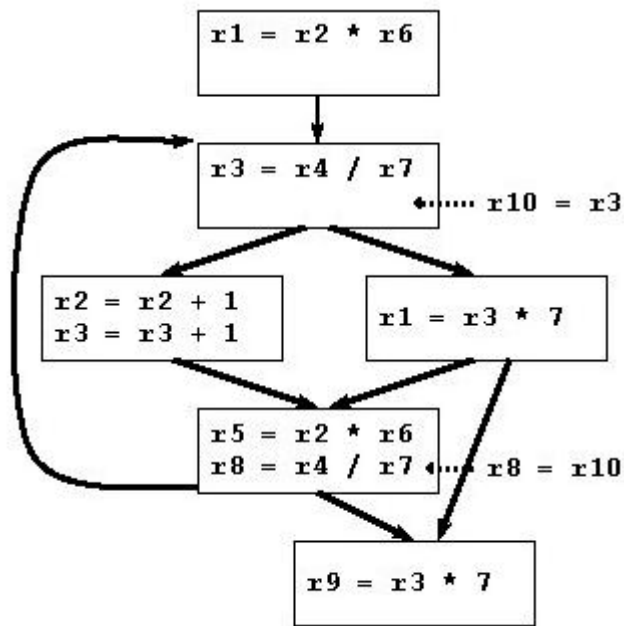
- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

The optimized code has the following advantages-

- Optimized code has faster execution speed.
- Optimized code utilizes the memory efficiently.
- Optimized code gives better performance.



Code Optimization Techniques

**Rules:**

1. X and Y have the same opcode and X dominates Y
2. $\text{src}(X) = \text{src}(Y)$ for all sres
3. For all sres, no def of a src on any path between X and Y (excluding Y)
4. Insert $\text{rx} = \text{dest}(X)$ immediately after X for new register rx
5. Replace Y with $\text{move dest}(Y) = \text{rx}$

Optimization by Common Sub-expression elimination**Algorithm:**

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

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

Program:

```

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

```

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

```

```

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

```

Expected Output:

Enter the Number of Values: 5

Left: a right: 9

Left: b right: c+d

Left: e right: c+d

Left: f right: b+e

Left: r right: f

Intermediate Code

a=9

b=c+d

e=c+d

f=b+e

r=:f

After Dead Code Elimination

b =c+d

e =c+d

f =b+e

r =:f

Eliminate Common Expression

b =c+d

```
b =c+d  
f =b+b  
r =:f  
Optimized Code  
b=c+d  
f=b+b  
r=:f
```

Result: The above program for code optimization was successfully executed

ADDITIONAL PROGRAMS

PROGRAM 10. DESIGN of DFA from NFA

Aim : Write a C/C++ program to construct DFA from NFA

Description :

DFA – DFA stands for Deterministic Finite Automata which has a finite number of states, the machine is called Deterministic Finite Machine.

Formal Definition of a DFA : A DFA can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states.
- Σ is a finite set of symbols called the alphabet.
- δ is the transition function where $\delta: Q \times \Sigma \rightarrow Q$
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

NFA - NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language. The finite automata are called NFA when there exist many paths for specific input from the current state to the next state. Every NFA is not DFA, but each NFA can be translated into DFA.

Formal Definition of an NFA : An NFA can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states.
- Σ is a finite set of symbols called the alphabets.
- δ is the transition function where $\delta: Q \times \Sigma \rightarrow 2Q$

(Here the power set of Q ($2Q$) has been taken because in case of NFA, from a state, transition can occur to any combination of Q states)

- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

Conversion of NFA to DFA

Steps for converting NFA to DFA:

- 1: Initially $Q' = \phi$
- 2: Add q_0 of NFA to Q' . Then find the transitions from this start state.
- 3: In Q' , find the possible set of states for each input symbol. If this set of states is not in Q' , then add it to Q' .
- 4: In DFA, the final state will be all the states which contain F (final states of NFA)

Algorithm :

1. Start the program.
2. Accept the number of state A and B.

3. Find the E-closure for node and name if as A.
4. Find $v(a,a)$ and (a,b) and find a state.
5. Check whether a number new state is obtained.
6. Display all the state corresponding A and B.
7. Stop the program.

Program :

```
#include <iostream>
#include<stdio.h>
#include<ctype.h>
#include<process.h>
using namespace std;

typedef struct
{
    int num[10],top;
}
stack;
stack s;
int mark[16][31],e_close[16][31],n,st=0;
char data[15][15];
void push(int a)
{
    s.num[s.top]=a;
    s.top=s.top+1;
}
int pop()
{
    int a;
    if(s.top==0)
        return(-1);
    s.top=s.top-1;
    a=s.num[s.top];
    return(a);
}
void epi_close(int s1,int s2,int c)
{
    int i,k,f;
    for(i=1;i<=n;i++)
    {
        if(data[s2][i]=='e')
        {
            f=0;
            for(k=1;k<=c;k++)
                if(e_close[s1][k]==i)
                {
                    f=1;
                    if(f==0)
                    {
```



```
        c++;
        e_close[s1][c]=i;
        push(i);
    }
}
}
while(s.top!=0) epi_close(s1,pop(),c);
}
int move(int sta,char c)
{
    int i;
    for(i=1;i<=n;i++)
    {
        if(data[sta][i]==c)
            return(i);
    }
    return(0);
}
void e_union(int m,int n)
{
    int i=0,j,t;
    for(j=1;mark[m][i]!=-1;j++)
    {
        while((mark[m][i]!=e_close[n][j])&&(mark[m][i]!=-1))
            i++;
        if(mark[m][i]==-1)
            mark[m][i]=e_close[n][j];
    }
}
int main(int argc, char *argv[])
{
    int i,j,k,Lo,m,p,q,t,f;
    printf("\n enter the NFA state table entries:");
    scanf("%d",&n);
    printf("\n");
    for(i=0;i<=n;i++)
        printf("%d",i);
    printf("\n");
    for(i=0;i<=n;i++)
        printf("-----");
    printf("\n");
    for(i=1;i<=n;i++)
    {
        printf("%d|",i);
        fflush(stdin);
        for(j=1;j<=n;j++)
            scanf("%c",&data[i][j]);
    }
    for(i=1;i<=15;i++)
    for(j=1;j<=30;j++)
    {
```

```

    e_close[i][j]=-1;
    mark[i][j]=-1;
}
for(i=1;i<=n;i++)
{
    e_close[i][1]=i;
    s.top=0;
    epi_close(i,i,1);
}
for(i=1;i<=n;i++)
{
    for(j=1;e_close[i][j]!=-1;j++)
        for(k=2;e_close[i][k]!=-1;k++)
            if(e_close[i][k-1]>e_close[i][k])
            {
                t=e_close[i][k-1];
                e_close[i][k-1]=e_close[i][k];
                e_close[i][k]=t;
            }
}
printf("\n the epsilon closures are:");
for(i=1;i<=n;i++)
{
    printf("\n E(%d)={ ",i);
    for(j=1;e_close[i][j]!=-1;j++)
        printf("%d",e_close[i][j]);
    printf("}");
}
j=1;
while(e_close[1][j]!=-1)
{
    mark[1][j]=e_close[1][j];
    j++;
}
st=1;
printf("\n DFA Table is:");
printf("\n a b ");
printf("\n-----");
for(i=1;i<=st;i++)
{
    printf("\n{ ");
    for(j=1;mark[i][j]!=-1;j++)
        printf("%d",mark[i][j]);
    printf("}");
    while(j<7)
    {
        printf(" ");
        j++;
    }
    for(Lo=1;Lo<=2;Lo++)
    {

```

```
for(j=1;mark[i][j]!=-1;j++)
{
    if(Lo==1)
        t=move(mark[i][j],'a');
    if(Lo==2)
        t=move(mark[i][j],'b');
    if(t!=0)
        e_union(st+1,t);
}
for(p=1;mark[st+1][p]!=-1;p++)
for(q=2;mark[st+1][q]!=-1;q++)
{
    if(mark[st+1][q-1]>mark[st+1][q])
    {
        t=mark[st+1][q];
        mark[st+1][q]=mark[st+1][q-1];
        mark[st+1][q-1]=t;
    }
}
f=1;
for(p=1;p<=st;p++)
{
    j=1;
    while((mark[st+1][j]==mark[p][j])&&(mark[st+1][j]!=-1))
        j++;
    if(mark[st+1][j]==-1 && mark[p][j]==-1)
        f=0;
}
if(mark[st+1][1]==-1)
    f=0;
printf("\t{ ");
for(j=1;mark[st+1][j]!=-1;j++)
{
    printf("%d",mark[st+1][j]);
}
printf("}\t");
if(Lo==1)
    printf(" ");
if(f==1)
    st++;
if(f==0)
{
    for(p=1;p<=30;p++)
        mark[st+1][p]=-1;
}
}
}
system("PAUSE");
return EXIT_SUCCESS;
}
```

Expected Output:

Enter the NFA state table entries: 11

(**Note:** *Instead of '-' symbol use blank spaces in the output window*)

0 1 2 3 4 5 6 7 8 9 10 11

```

-----
1 - e - - - - e - -
2 - - e - e - - - -
3 - - - a - - - - -
4 - - - - e - - - -
5 - - - - b - - - -
6 - - - - e - - - -
7 - e - - - - e - -
8 - - - - - e - -
9 - - - - - e -
10 - - - - - e
11 - - - - -

```

The Epsilon Closures Are:

E(1)={ 12358}
 E(2)={ 235}
 E(3)={ 3}
 E(4)={ 234578}
 E(5)={ 5}
 E(6)={ 235678}
 E(7)={ 23578}
 E(8)={ 8}
 E(9)={ 9}
 E(10)={ 10}
 E(11)={ 11}

DFA Table is:

a	b	
{ 12358}	{ 2345789}	{ 235678}
{ 2345789}	{ 2345789}	{ 23567810}
{ 235678}	{ 2345789}	{ 235678}
{ 23567810}	{ 2345789}	{ 23567811}
{ 23567811}	{ 2345789}	{ 235678}

```

D:\cc\scanner.exe

enter the NFA state table entries:11
01234567891011
-----
1| e   e
2|  e e
3|   a
4|   e
5|   b e
6|   e e
7| e   e
8|   e e
9|   e e
10|  e
11|

```

```

D:\cc\scanner.exe

the epsilon closures are:
E<1>={1235891011}
E<2>={235}
E<3>={3}
E<4>={23457}
E<5>={5}
E<6>={23567}
E<7>={2357}
E<8>={891011}
E<9>={91011}
E<10>={1011}
E<11>={11}
DFA Table is:
a b
-----
<1235891011>  <23457>      <23567>
<23457>       <23457>      <23567>
<23567>       <23457>      <23567>

```

PROGRAM 11: Program to implement recursive descent parser

Aim: Write a C/ C++ program to implement recursive descent parser

Description:

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

Algorithm:

Input: Context Free Grammar without last recursion and an input string from the grammar

Output: Sequence of productions rules used to derive the sentence.

Method:

Consider the grammar

$E \rightarrow TE$

$E' \rightarrow +TE'/e$

$T \rightarrow FT$

$T \rightarrow *FT/e$

$F \rightarrow (E)/Id$

To recursive decent parser for the above grammar is given below

Procedure:

Begin

$T()$

$E_prime();$

 print $E \rightarrow TE'$

end

procedure $eprime():$

if $ip_sym \neq '+'$ then

begin

$advance();$

$T();$

$eprime();$

 prime $E' \rightarrow TE'$

end

else

 print $E' \rightarrow e$

 procedure $T();$

begin

$e();$

$Tprime();$

 print $T \rightarrow FT'$;

end;

procedure $Tprime();$

if $ip_sym = '*'$ then

```
begin
    advance();
    F();
    Tprime()
    print T'->T*FT'
end
else
    print T'->e
procedure F()
if ip_sym ==id then
begin
    advance();
    print->id
end
else
    Error();
end;
else
    Error();
```

Program:

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

```
        ip++;
        T();
        E1();
    }
    return;
}
void T()
{
    F();
    T1();
    return;
}
void T1()
{
    if(s[ip]=='*')
    {
        ip++;
        F();
        T1();
    }
    return;
}
void F()
{
    if(s[ip]=='(')
    {
        ip++;
        E();
        if(s[ip]==')')
            ip++;
    }
    else
        if(s[ip]=='i')
            ip++;
    else
        printf("\n id expected");
    return;
}
```

Expected Output:

cc recurparser.c

./a.out

enter the input

(i+i)*(i*i)\$

the string is :(i+i)*(i*i)\$

string is accepted the length of string is 11

PROGRAM 12: Program to find FIRST and FOLLOW for the given grammar**A] Aim: Write a C/C++ program to find FIRST for the given grammar****Description:**

FIRST of a non-terminal would refer to the very first character of the strings that can be derived starting from that non-terminal. The FIRST of a terminal would be the terminal itself.

RULES FOR COMPUTING FIRST**Rule-01:**For a production rule $X \rightarrow \epsilon$, $\text{First}(X) = \{ \epsilon \}$ **Rule-02:**

For any terminal symbol 'a',

 $\text{First}(a) = \{ a \}$ **Rule-03:**For a production rule $X \rightarrow Y_1 Y_2 Y_3$,Calculating $\text{First}(X)$ If $\epsilon \notin \text{First}(Y_1)$, then $\text{First}(X) = \text{First}(Y_1)$ If $\epsilon \in \text{First}(Y_1)$, then $\text{First}(X) = \{ \text{First}(Y_1) - \epsilon \} \cup \text{First}(Y_2 Y_3)$ Calculating $\text{First}(Y_2 Y_3)$ If $\epsilon \notin \text{First}(Y_2)$, then $\text{First}(Y_2 Y_3) = \text{First}(Y_2)$ If $\epsilon \in \text{First}(Y_2)$, then $\text{First}(Y_2 Y_3) = \{ \text{First}(Y_2) - \epsilon \} \cup \text{First}(Y_3)$ Similarly, we can make expansion for any production rule $X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$ **Program:**

#include<stdio.h>

#include<ctype.h>

int main()

{

int i,n,j,k;

char str[10][10],f;

printf("Enter the number of productions\n");

scanf("%d",&n);

printf("Enter grammar\n");

for(i=0;i<n;i++)

scanf("%s",&str[i]);

for(i=0;i<n;i++)

{

f= str[i][0];

int temp=i;

if(isupper(str[i][3]))

{

repeat:

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

{

if(str[k][0]==str[i][3])

```
        {
            if(isupper(str[k][3]))
            {
                i=k;
                goto repeat;
            }
            else
            {
                printf("First(%c)=%c\n",f,str[k][3]);
            }
        }
    }
}
else
{
    printf("First(%c)=%c\n",f,str[i][3]);
}
i=temp;
}
}
```

Expected Output:

```
cc first.c
./a.out
Enter the number of productions
3
Enter grammar
S->AB
A->a
B->b
First(S)=a
First(A)=a
First(B)=b
```

B] Aim: Write a C/C++ program to find FOLLOW for the given grammar**Description:**

The FOLLOW of a non-terminal A refers to the FIRST of the non-terminal or terminal that immediately comes after A in the derivation rules.

They are used by parsing algorithms to determine which production to use for parsing a string. If we have a choice of multiple productions, and we wish to generate a string x we would like to use the productions that can generate strings that start with x. This is where FIRST can be used to identify such productions.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place \$ in Follow(S).

Rule-02:

For any production rule $A \rightarrow \alpha B$,

$$\text{Follow}(B) = \text{Follow}(A)$$

Rule-03:

For any production rule $A \rightarrow \alpha B\beta$,

If $\epsilon \notin \text{First}(\beta)$, then $\text{Follow}(B) = \text{First}(\beta)$

If $\epsilon \in \text{First}(\beta)$, then $\text{Follow}(B) = \{ \text{First}(\beta) - \epsilon \} \cup \text{Follow}(A)$

Program:

```
#include<stdio.h>
main()
{
    int np,i,j,k;
    char prods[10][10],follow[10][10],Imad[10][10];
    printf("enter no. of productions\n");
    scanf("%d",&np);
    printf("enter grammar\n");
    for(i=0;i<np;i++)
    {
        scanf("%s",&prods[i]);
    }

    for(i=0; i<np; i++)
    {
        if(i==0)
        {
            printf("Follow(%c) = $\n",prods[0][0]);
        }
        for(j=3;prods[i][j]!='\0';j++)
        {
            int temp2=j;
```

```

if(prods[i][j] >= 'A' && prods[i][j] <= 'Z')
{
    if((strlen(prods[i])-1)==j)
    {
        printf("Follow(%c)=Follow(%c)\n",prods[i][j],prods[i][0]);
    }
    int temp=i;
    char f=prods[i][j];
    if(!isupper(prods[i][j+1])&&(prods[i][j+1]!='\0'))
    printf("Follow(%c)=%c\n",f,prods[i][j+1]);
    if(isupper(prods[i][j+1]))
    {
        repeat:
        for(k=0;k<np;k++)
        {
            if(prods[k][0]==prods[i][j+1])
            {
                if(!isupper(prods[k][3]))
                {
                    printf("Follow(%c)=%c\n",f,prods[k][3]);
                }
                else
                {
                    i=k;
                    j=2;
                    goto repeat;
                }
            }
        }
        i=temp;
    }
    j=temp2;
}
}
}

```

Expected Output:

```

./a.out
enter no. of productions
3
enter grammar
S->AB
A->a
B->b
Follow(S) = $
Follow(A)=b
Follow(B)=Follow(S)

```

Program 13: Implementation of symbol table using hashing

Aim: Write a C/C++ program to implement Symbol table using Hashing.

Description: In the symbol table information is gathered about the names that appear in the program. The symbol table primarily helps: In checking the program's semantic correctness & in generating code.

Furthermore, the following features are desirable:

- Support for block structures and scope rules, if the language requires that.
In block-structured languages the same identifier can occur several times, but in different blocks. The symbol table, therefore, ought to be organised so you have access to the variables which are visible at a given time.
- Compact storage.
An implementation requiring little memory is to be aimed for.
- Quick addition of records, and searching for names.
The need to search fast collides (as usual) with memory space demands, as mentioned above.
- Resident storage.
The symbol table may be needed later, if the compiler works in several runs, or for diagnosis of execution errors.

The information stored in the symbol table is a list of pairs (*name, attribute*), also often referred to as *bindings*:

Name: The name forms a search key in the symbol table. A problem which must be dealt with is how to efficiently store all significant characters in the identifier.

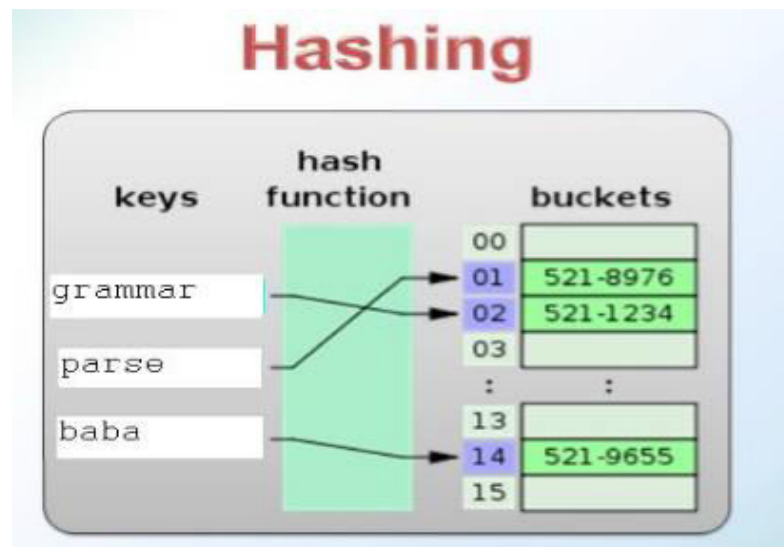
Attributes which are dependent on the language to be compiled: These include type information, array dimensions, the number of parameters in procedures and functions, as well as line numbers for declarations.

Attributes which are dependent on the language we are compiling to: For example, if we are generating machine code, we will have problems with references to addresses in memory, whose values are unknown at code generation time. If we choose to generate assembler, we can instead refer to a so-called label (symbolic position) whose value is determined later. Assembler will then perform the translation to absolute addresses in memory.

The operations we would like to perform on the symbol tables are:

- Look up a name
- Add a new name
- Add an attribute to a name
- Retrieve the value of an attribute of a name
- Remove a name and its associated information

Hash() is an mathematical function to derive index values of given data elements. Advantage is to maintain large amount of data items in a given limited collection.

**Algorithm:**

1. Start processing.
2. Declare structure for input and output files
3. Declare File pointers for input and output files
4. Open Input File(s) in Read mode and Open Output File(s) in write Mode.
5. Read the Intermediate File until EOF occurs.
 - 5.1 If Symbol is not equal to - then
 - 5.2 Generate the hash index for the symbol using the hash function.
 - 5.3 Write the Symbol Name and its address into Symbol table.
6. Close all the Files.
7. Print Symbol Table is created.
8. Stop processing.

Program:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<conio.h>
#define size 50
struct intermediate
{
    int addr;
    char label[10];
    char mnem[10]; char op[10];
}res;

struct symbol
{
    char symbol[10];
    int addr;
}sy,syy[size];
int hs=size;
unsigned hash(char *); void search();
void main()
{
    FILE *s1,*p1; char *lb;
```

```
int as=0,i=0; clrscr();
s1=fopen("source.txt","r");
p1=fopen("symbol.txt","w"); while(!feof(s1))
{
fscanf(s1,"%d %s %s
%s",&res.addr,res.label,res.mnem,res.op);
strcpy(lb,res.label); if(strcmp(res.label,"-")!=0)
{
as=hash(lb); strcpy(syy[as].symbol,res.label); syy[as].addr=res.addr; strcpy(sy.symbol,res.label);
sy.addr=res.addr; fprintf(p1,"%s\t%d\n",sy.symbol,sy.addr);
}
}
fcloseall();

printf("Symbol Table\n"); printf("Index\tLabel\tAddr\n"); for(i=0;i<size;i++)
{
if(syy[i].addr!=0)
printf("%d\t%s\t%d\n",i,syy[i].symbol,syy[i].addr);
}
search();
}

unsigned hash(char *s)
{
unsigned hashval;
for (hashval = 0; *s != '\0'; s++) hashval = *s + hashval;
return hashval % hs;
}

void search()
{
char *name; char ch;
int val=0;
do
{
printf("Do u want to search label using hashing:(y/n)");
ch=getche();
if(ch=='y')
{
printf("\nEnter the label to be searched:"); scanf("%s",name);
val=hash(name); if(strcmp(syy[val].symbol,name)!=0)
val=0; if(val!=0)
printf("Label %s found in hash table at index %d with value
%d\n",syy[val].symbol,val,syy[val].addr);
else
printf("Label not found\n");
}
}while(ch!='n');
getch();
}
```

Sample-Input & Output:

Input File:

SOURCE.TXT

```
1000 start  lda    copy
1003 endfil add    0050
1006 copy  resw    1
1009 first  stl     endfil
1012 -      end     start
```

Output File:

SYMBOL.TXT

```
start  1000
endfil 1003
copy   1006
first  1009
```

Symbol Table

Index	Label	Addr
17	copy	1006
24	start	1000
36	first	1009
48	endfil	1003

Do u want to search label using hashing:(y/n)y Enter the label to be searched:start
Label start found in hash table at index 24 with value 1000

Do u want to search label using hashing:(y/n)y Enter the label to be searched:cloop
Label not found
Do u want to search label using hashing:(y/n)n

SAMPLE VIVA-VOCE QUESTIONS**1) Translator**

A program that accepts text expressed in one language and generates semantically equivalent text expressed in another language.

2) source language

The input language of a translator.

3)target language

The output language of a translator.

4) assembler

A translator from an assembly language to the corresponding machine language.

5) compiler

A translator from a high level language to a low level language.

6) high-level translator

A translator from one high-level language to another.

7)disassembler

A translator from machine language to assembler language.

8)decompiler

A translator from a low level language to a high level language.

9)source program

The input text of an assembler or compiler.

10)object program

The output text of an assembler or compiler.

11)implementation language

The language in which a program is expressed.

12)tombstone diagram

A graphical representation of the overall function of a system.

13)cross compiler

A compiler which generates code for a machine different from the machine on which it is run .

14)Portable program

A program which can be (compiled and) run on any machine.

15)interpretive compiler

A program which combines a compiler that produces object code in an intermediate language with an interpreter for that intermediate language.

16)What is a compiler

A compiler is a computer program (or set of programs) that transforms source code written in a

programming language (the source language) into another computer language (the target language, often having a binary form known as object code).

17)Difference between compilers & interpreters.

A compiler first takes in the entire program, checks for errors, compiles it and then executes it. Whereas, an interpreter does this line by line, so it takes one line, checks it for errors and then executes it.

Eg of Compiler - C

Eg of Interpreter – PHP

18)Language processor.

a parser which parses a particular language are called language processors

19)Symbol table.

a symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type,scope level and sometimes its location.

20)Explain Different Phases Of A Compiler With An Example**1. Lexical analysis**

This is the initial part of reading and analysing the program text: The text is read and divided into tokens , each of which corresponds to a symbol in the programming language, e.g. , a variable name, keyword or number.

2. Syntax analysis

This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the syntax tree) that reflects the structure of the program. This phase is often called parsing

3. Semantic analysis - checks for errors.**4. Intermediate Code generation - generates machine code.****5. Code optimization (Machine independent)- looks for ways to make code smaller and more efficient.****6. Code Generator****7. Target program (machine dependent)- creates the output (.exe, .com, .dll, etc.).****21)RISC**

Basically, RISC CPU's (eg: ARM processor...) contain an instruction set where every instruction and operand is of the exact same length. This makes the CPU design much simpler since every instruction and operand fits in the pipeline with no wasted cycles

22)CISC

CISC CPU's (x86, 8051, etc...) contain complex, variable length instruction sets and operands. While having complex variable length instructions make low level programming much easier, it comes at the price of greatly increasing the complexity of the CPU and reducing efficiency.

23)Application of compiler technology.

pattern recognition systems.

diagram recognition and diagram-processing tasks by use of grammars

24)Lexical Analyzer

lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer, lexer or scanner.

25)Tokens, Patterns, Lexemes

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, num, real, . . .)

26)Patterns

There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Regular expressions are an important notation for specifying patterns.

For example, the pattern for the Pascal identifier token, id, is: $\text{id} \rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$.

27)Lexeme

A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

For example, (= , < , >=)

28)Regular Expressions

1. The regular expressions over alphabet specifies a language according to the following rules. is a regular expression that denotes { }, that is, the set containing the empty string.

29)Regular Definitions

A regular definition gives names to certain regular expressions and uses those names in other regular expressions.

30)Deterministic Finite Automata (DFA)

A deterministic finite automation is a special case of a non-deterministic finite automation (NFA) in which

1. no state has an ϵ -transition
2. for each state s and input symbol a , there is at most one edge labeled a leaving s .

31)Nondeterministic Finite Automata (NFA)

A nondeterministic finite automation is a mathematical model consists of

1. a set of states S ;
2. a set of input symbol, Σ , called the input symbols alphabet.
3. a transition function move that maps state-symbol pairs to sets of states.
4. a state so called the initial or the start state.
5. a set of states F called the accepting or final state.

32)Synthesized Attributes:

An attribute is synthesized if its value at a parent node can be determined from attributes of its children.

33)Syntax-Directed Definitions:

- A syntax-directed definition uses a CFG to specify the syntatic structure of the input.
- A syntax-directed definition associates a set of attributes with each grammar symbol.
- A syntax-directed definition associates a set of semantic rules with each production rule.

34)Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar. A parser

must be capable of constructing the tree, or else the translation cannot be guaranteed correct. For any language that can be described by CFG, the parsing requires $O(n^3)$ time to parse string of n tokens. However, most programming languages are so simple that a parser requires just $O(n)$ time with a single left-to-right scan over the input string of n tokens.

There are two types of Parsing

1. Top-down Parsing (start from start symbol and derive string)

A Top-down parser builds a parse tree by starting at the root and working down towards the leaves.

- o Easy to generate by hand.

- o Examples are : Recursive-descent, Predictive.

2. Bottom-up Parsing (start from string and reduce to start symbol)

A bottom-up parser builds a parse tree by starting at the leaves and working up towards the root.

- o Not easy to handle by hands, usually compiler-generating software generate bottom up parser

- o But handles larger class of grammar

- o Example is LR parser.

35) Predictive Parsing:

Recursive-descent parsing is a top-down method of syntax analysis that executes a set of recursive procedure to process the input. A procedure is associated with each nonterminal of a grammar.

A predictive parsing is a special form of recursive-descent parsing, in which the current input token unambiguously determines the production to be applied at each step.

36) Left Recursion:

The production is left-recursive if the leftmost symbol on the right side is the same as the non terminal on the left side. For example,

$\text{expr} \rightarrow \text{expr} + \text{term}.$

37) Issues in the Design of Code generator

Code generator concern with:

1. Memory management.

2. Instruction Selection.

3. Register Utilization (Allocation).

4. Evaluation order.

38) What is lex

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers").

Lex is commonly used with the yacc parser generator.

Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

39) Lexical error:

A lexical error is any input that can be rejected by the lexer. This generally results from token recognition falling off the end of the rules you've defined.

40) Types of parsers

Top-down parsers

- _ start at the root of derivation tree and _ll in

- _ picks a production and tries to match the input

- _ may require backtracking

- _ some grammars are backtrack-free (predictive)

Eg: recursive descent, LL

Bottom-up parsers

- _ start at the leaves and _ll in
 - _ start in a state valid for legal _rst tokens
 - _ as input is consumed, change state to encode possibilities (recognize valid pre_xes)
 - _ use a stack to store both state and sentential forms
- Eg: LR, CYK (look ahead) parser , slr, lalr

41)What does LL and LR mean

LL(1) means – first ‘L’ represents scanning input from left to right. Second ‘L’ represents producing leftmost derivation. (1) one input symbol of lookahead at each step.

LR(k) means – ‘L’ Left to right scanning, R-rightmost derivation in reverse. K-0 or 1 no. of i/p symbols.

Loaders:

A loader is a system program, which takes the object code of a program as input and prepares it for execution.

- The loader performs the following functions :
- Allocation - The loader determines and allocates the required memory space for the program to execute properly.
- Linking -- The loader analyses and resolve the symbolic references made in the object modules.
- Relocation - The loader maps and relocates the address references to correspond to the newly allocated memory space during execution.
- Loading - The loader actually loads the machine code corresponding to the object modules into the allocated memory space and makes the program ready to execute.

42)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 inthe designated memory loactions.

43)Absolute Loader

The assembler generates the object code equivalent of the source program bu the output is punched on to the cads forming the object decks instead of loading in memory.

44)Relocation loader

Loaders that allow for program relocation are called relocating loaders/relative loaders.

45)Linkage Editors

A linkage editor produces a linked version for the program called an executable image, which is written to a file for later execution.

46)Dynamic Linking

Dynamic linking allows an object module to include only the information that is require at load time to execute a program.

There are two types of dynamic linking, they are Load time dynamic linking and Run time dynamic linking.

47)Code Generation

code generation is the process by which a compiler's code generator converts some internal representation of source code into a form (e.g., machine code) that can be readily executed by a machine

48)code optimization

Code optimization is the process of modifying the code to make some aspect of software or hardware work more efficiently or use fewer resources or reduce compilation time or use memory efficiently etc

49)CFG

CFG is a grammar which naturally generates a formal language in which clauses can be nested inside clauses arbitrarily deeply, but where grammatical structures are not allowed to overlap

50).Define Passes

In an implementation of a compiler, portion of one or more phases are combined into a module called pass. A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which is read by subsequent pass.

51) Define Lexical Analysis?

The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens. Identifiers, keywords, constants, operators and punctuation symbols are typical tokens.

52) Write notes on syntax analysis?

Syntax analysis is also called parsing. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.

53) What is meant by semantic analysis?

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operand of expressions and statements.

54) Define optimization?

Certain compilers apply transformations to the output of the intermediate code generator. It is used to produce an intermediate-language from which a faster or smaller object program can be produced. This phase is called optimization phase. Types of optimization are local optimization and loop optimization.

55) What is cross compiler?

A compiler may run on one machine and produce object code for another machine is called cross compiler.

56) Define semantics of a programming language?

The rules that tell whether a string is a valid program or not are called syntax of the language. The rules that give meaning to programs are called the semantics of a programming language.

57) What are the data elements of a programming language?

- a) Numerical data.
- b) Logical data.
- c) Character data.
- d) Pointers.
- e) Labels.

58) Define binding?

The act of associating attributes to a name is referred to as binding the attributes to the name. Most binding done at compile time called static binding. Some languages, such as SNOBOL allow dynamic binding, binding done at run time.

59) What is coercion of types?

The translation of the operator, which the compiler must provide, includes any necessary conversion from one type to another, and this implied change in type is called coercion.

60) What are the possible error recovery actions in lexical analysis:

- a) Deleting an extraneous character
- b) Inserting a missing character
- c) Replacing an incorrect character by a correct character
- d) Transposing two adjacent characters

61) Define regular expressions?

Regular expressions are the notation we shall use to define the class of languages known as regular sets. It is used to describe tokens. In regular expression notation we could write
identifier = letter (letter | digit)*

62) Write the regular expression for denoting the set containing the string a and all strings consisting of zero or more a's followed by a b.

a | a * b

63) Define finite automata?

A better way to convert a regular expression to a recognizer is to construct a generalized transition diagram from the expression. This diagram is called a finite automaton.

64) What are the representations of three-address statements?

A three address statement is an abstract form of intermediate code. There are three representation are available. They are

- a) Quadruples
- b) Triples
- c) Indirect triples

65) Define procedure definition?

A procedure definition is a declaration that, in its simplest form, associates an identifier with a statement. The identifier is the procedure name, and the statement body. Some of the identifiers appearing in a procedure definition are special and are called formal parameters of the procedure. Arguments, known as actual parameters may be passed to a called procedure; they are substituted for the formal in the body.

66) Define activation trees?

A recursive procedure p need not call itself directly; p may call another procedure q, which may then call p through some sequence of procedure calls. We can use a tree called an activation tree, to depict the way control enters and leaves activation. In an activation tree

- a) Each node represents an activation of a procedure,
- b) The root represents the activation of the main program
- c) The node for a is the parent of the node for b if and only if control flows from activation a to b, and
- d) The node for a is to the left of the node for b if and only if the lifetime of a occurs before the lifetime of b.

67) Write notes on control stack?

A control stack is to keep track of live procedure activations. The idea is to push the node for activation onto the control stack as the activation begins and to pop the node when the activation ends.

68) Write the scope of a declaration?

A portion of the program to which a declaration applies is called the scope of that declaration. An occurrence of a name in a procedure is said to be local to procedure if it is in the scope of a declaration within the procedure; otherwise the occurrence is said to be nonlocal.

69) Define binding of names?

When an environment associates storage location s with a name x , we say that x is bound to s ; the association itself is referred to as a binding of x . A binding is the dynamic counterpart of a declaring.

70) What is the use of run time storage?

The run time storage might be subdivided to hold

- a) The generated target code
- b) Data objects, and
- c) A counterpart of the control stack to keep track of procedure activation.

71) What is an activation record?

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record or frame, consisting of the collection of fields such as

- a) Return value
- b) Actual parameters
- c) Optional control link
- d) Optional access link
- e) Saved machine status
- f) Local data
- g) Temporaries

72) What are the storage allocation strategies?

- a) Static allocation lays out storage for all data objects at compile time.
- b) Stack allocation manages the run-storage as a stack.
- c) Heap allocation allocates and deallocates storage as needed at run time from a data area known as heap.

73) What is static allocation?

In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package. Since the bindings do not change at run time, every time a procedure is activated, its names are bound to the same storage location.

74) Define LR grammar?

A grammar for which we can construct a parsing table in which every entry is uniquely defined is said to be an LR grammar.

75) What is augmented grammar?

If G is a grammar with start symbol S , then G' , the augmented grammar for G , is G with a new start symbol S' and production $S' \rightarrow S$. It is to indicate the parser when it should stop and announce acceptance of the input.

76) Define intermediate code?

In many compilers the source code is translated into a language which is intermediate in complexity between a high-level programming language and machine code. Such a language is therefore called intermediate code or intermediate text.

77) What are the benefits of using a machine-independent intermediate form?

- a) Retargeting is facilitated; a compiler for a different machine can be created by attaching a back end for the new machine to an existing frontend.
- b) A machine-independent code optimizer can be applied to the intermediate representation.

78) What are the various kinds of intermediate representations for intermediate code generation?

- a) Syntax trees
- b) Postfix notation
- c) Three address code

79) What is syntax directed translation scheme?

A syntax directed translation scheme is merely a context-free grammar in which a program fragment called an output action (or sometimes a semantic action or semantic rule) is associated with each production.

80) Define parse trees and syntax trees.

The parse tree itself is a useful intermediate language representation for a source program. A parse tree, however often contains redundant information which can be eliminated. A variant of a parse tree is what is called an syntax tree, a tree in which each leaf represents an operand and each interior node an operator.

81) What is a three-address code?

Three-address code is a sequence of statements, typically of the general form $A := B \text{ op } C$, where A, B and C are either programmer-defined names, constants or compiler-generated temporary names; op stands for any operator, such as fixed- or floating-point arithmetic operator, or a logical operator on Boolean valued data.

82) Write the three address code for the assignment statement $a := b * -c + b * -c$

$t1 := -c, t2 := b * t1, t3 := -c, t4 := b * t3$
 $t5 := t2 + t4, a := t5$

83) Name any four types of three-address statements?

- a) Assignment statements of the form $x := y \text{ op } z$
- b) Assignment instruction of the form $x := \text{op } y$
- c) Copy statement of the form $x := y$
- d) The unconditional jump goto L