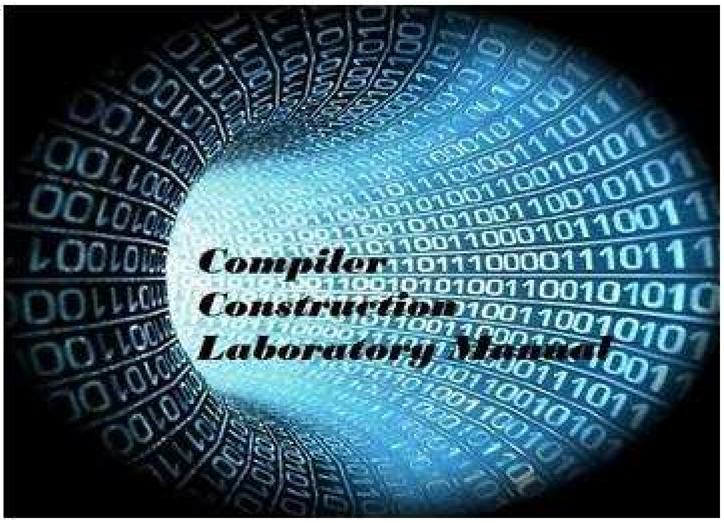
MUFFAKHAM JAH COLLEGE OF ENGINEERING AND TECHNOLOGY

Banjara Hills, Hyderabad, Telangana

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Compiler Construction Laboratory Manual



Academic Year 2016-2017

Table of Contents

Part I

Contents

1. Vision of the Institution

To be part of universal human quest for development and progress by contributing high calibre, ethical and socially responsible engineers who meet the global challenge of building modern society in harmony with nature.

2. Mission of the Institution

- To attain excellence in imparting technical education from undergraduate through doctorate levels by adopting coherent and judiciously coordinated curricular and co-curricular programs.
- To foster partnership with industry and government agencies through collaborative research and consultancy.
- To nurture and strengthen auxiliary soft skills for overall development and improved employability in a multi-cultural work space.
- To develop scientific temper and spirit of enquiry in order to harness the latent innovative talents.
- To develop constructive attitude in students towards the task of nation building and empower them to become future leaders
- To nourish the entrepreneurial instincts of the students and hone their business acumen.
- To involve the students and the faculty in solving local community problems through economical and sustainable solutions.

3. Department Vision

To contribute competent computer science professionals to the global talent pool to meet the constantly evolving societal needs.

4. Department Mission

Mentoring students towards a successful professional career in a global environment through quality education and soft skills in order to meet the evolving societal needs.

5. Programme Education Objectives

1. Graduates will demonstrate technical skills and leadership in their chosen fields of employment by solving real time problems using current techniques and tools.

2. Graduates will communicate effectively as individuals or team members and be successful in the local and global cross cultural working environment.

3. Graduates will demonstrate lifelong learning through continuing education and professional development.

4. Graduates will be successful in providing viable and sustainable solutions within societal, professional, environmental and ethical contexts

6. Programme Outcomes

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

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

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

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

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

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

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

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

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

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

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

12. **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

7. Programme Specific Outcomes

The graduates will be able to:

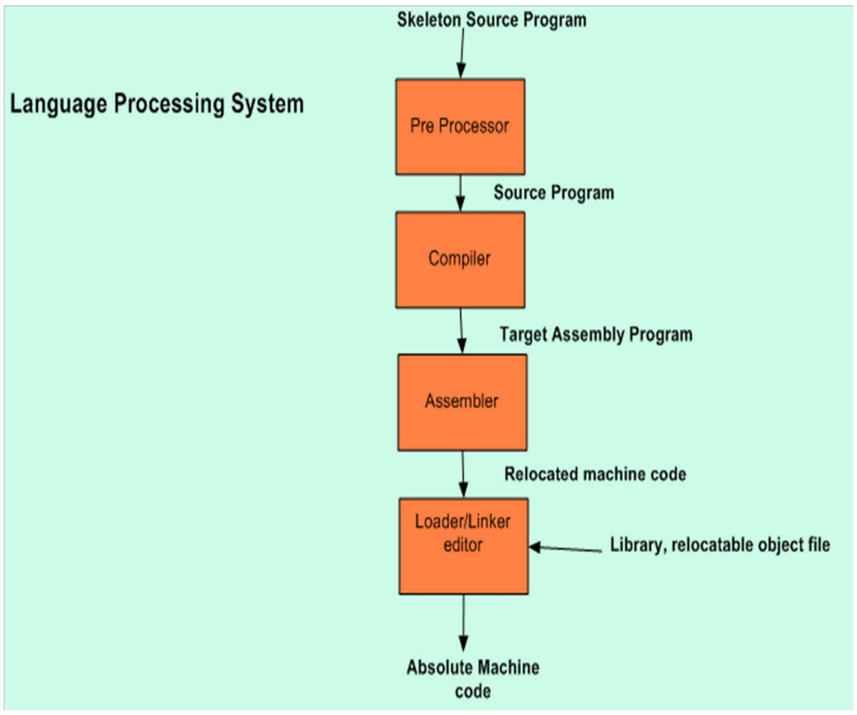
PSO1: Demonstrate understanding of the principles and working of the hardware and software aspects of computer systems.

PSO2: Use professional engineering practices, strategies and tactics for the development, operation and maintenance of software

PSO3: Provide effective and efficient real time solutions using acquired knowledge in various domains.

8. Introduction

OVERVIEW OF LANGUAGE PROCESSING SYSTEM



Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

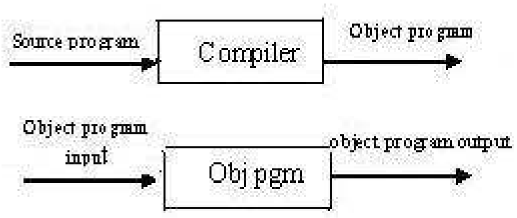
1. *M*acro processing: A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

COMPILER

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.

Source Program
Compilers
Target program

Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

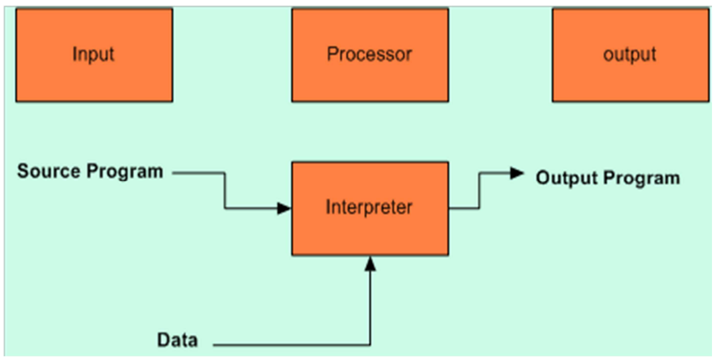


ASSEMBLER:

programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

INTERPRETER:

An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Synatx analysis
3. Semantic analysis
4. Direct Execution

Advantages:

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes a various may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages:

- The execution of the program is slower.
- *Memory* consumption is more.

Loader and Link-editor:

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To over come this problems of wasted translation time and memory. System programmers developed another component called loader

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could "relocate" directly behind the user's program. The task of adjusting programs they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of translator are:

1. Translating the hll program input into an equivalent ml program.
2. Providing diagnostic messages wherever the programmer violates specification of the hll.

TYPE OF TRANSLATORS:-

INTERPRETOR

COMPILER

PREPROSSESSOR

EXAMPLES OF COMPILERS

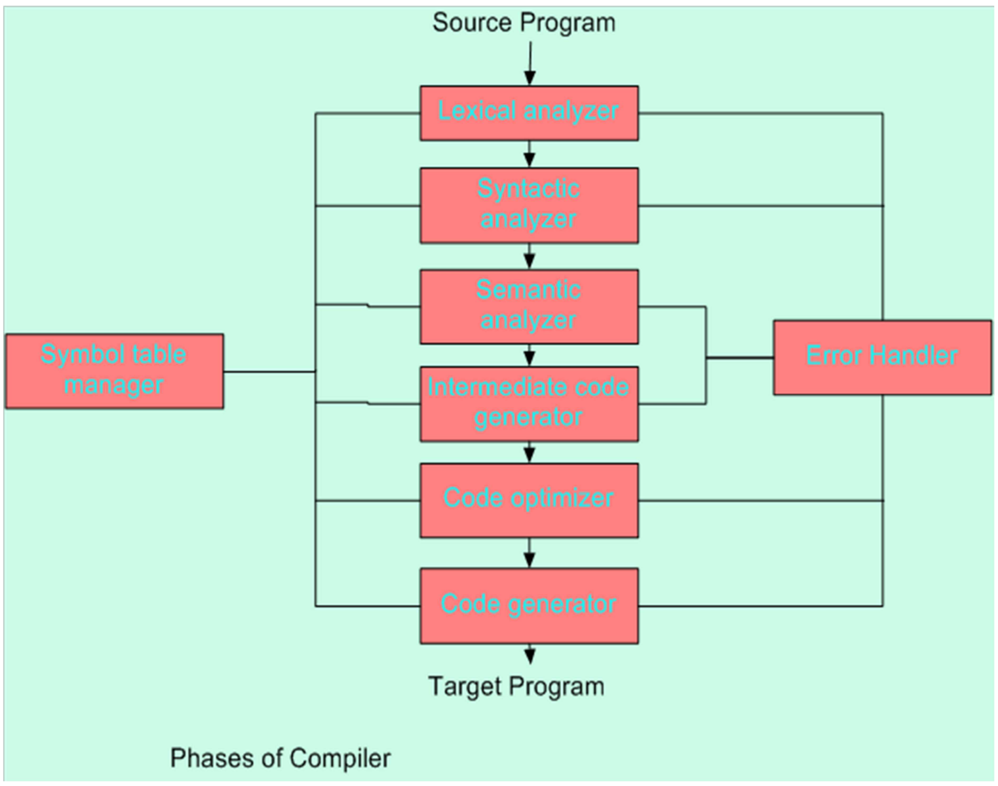
1. Ada compilers
2. ALGOL compilers
3. BASIC compilers
4. C# compilers
5. C compilers
6. C++ compilers
7. COBOL compilers
8. D compilers
9. Common Lisp compilers
10. ECMAScript interpreters
11. Eiffel compilers
12. Felix compilers
13. Fortran compilers
14. Haskell compilers
15. Java compilers

16. Pascal compilers
17. PL/I compilers
18. Python compilers
19. Scheme compilers
20. Smalltalk compilers
21. CIL compilers

STRUCTURE OF THE COMPILER DESIGN

***P*hases of a compiler:** A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below There are two parts of compilation.

1. Analysis (Machine Independent/Language Dependent)
2. Synthesis (Machine Dependent/Language independent) Compilation process is partitioned into no-of-sub processes called 'phases'.



a. **Lexical Analysis phase:-** Lexical Analysis or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called tokens.

b. **Syntax Analysis phase:-** The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc... are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

c. **Semantic analysis phase:-** In this phase semantics of the program will be checked. Ex: Type checking.

d. **Intermediate Code Generation phase:-** An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

e. **Code Optimization phase :-** This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

f. **Code Generation phase:-** The last phase of translation is code generation. A number of optimizations to reduce the length of machine language program are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

   i. **Table Management (or) Book-keeping:-** This is the portion to keep the names used by the program and records essential information about each. The data structure used to record this information called a 'Symbol Table'.

   ii. **Error Handlers:-** It is invoked when a flaw error in the source program is detected.The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as expression. Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

Lab Objective

The aim of this (compiler construction) lab is to provide a foundational base of the different phases involved in compiler design and construction. It is achieved by way of making the students implement simple programs related to the different phases associated with the design of compilers . The programming is mostly done through C under Linux Operating System, and LEX and YACC tools are also used

The programs implemented cover the following compiler phases which include: Lexical Analysis, Syntax Analysis, Code optimization, Code Generation.

At the end of the lab course the student is equipped with a basic idea and practical orientation of a simple compiler construction.

Part II

Programs

Program 1

Scanner programs using C (Lexical Analysis)

Problem Definition

The Scanner is used to the analyse the source program by reading the input character by character and grouping character into individual words and symbols.

Problem Description

The Scanner is a program which is responsible for transforming the source program into a compact and uniform format (tokens).The tokens represent basic program entities such as identifiers, Integers, reserve words, and delimiters.

The Scanner also removes white spaces (like newlines, tabs, blanks) and commands from a program.

Explanation

```
main( )
{

           int  sum =0;
           int  k = 11;
           if ( k <= 0)
             sum  =  sum  +  k;
          printf( \%d ",sum);

}
```

The scanner will scan the above program and splits into meaning full sequence of letters called lexemes and for each lexeme it generates a token.

The different types of tokens in the above program are:

main, int, if	(reserve words)
sum, k	(identifiers)
=,<=,+	(operators)
{,},;	(delimiters)
( )	(Parentheses)

Psuedocode: The following is the code to implement a scanner:

typedef enum token-types
BEGIN, END, READ, WRITE, ID, INTLITERAL, LPAREN, RPAREN, SEMICOLON, COMMA, ASSIGNOP, PLUSOP, MINUSOP, SCANOF token ;

```
extern token scanner (void);
#include<stdio.h>
#include<ctype.h>
extern char token_buffer [];
token scanner(void)
{
      int in_char;
      char c;
      clear_buffer( );
      if(feof(stdin))
            return SCANOF;
      while((in_char=getchar())!=EOF)
      {
            if(isspace(in_char))
                  continue;            /*doing nothing*/

            else if(isalpha(in_char))
            {
                  buffer=char (in_char);
                  for(c=getchar( ); isalnum || c = ='_' ; c=getchar())
                        buffer_char( c );
                  ungetc(c,stdin);
                  return check_reserved ( );
            }

            else if(isdigit(in_char))
            {
              buffer_char(c );
                  ungetch(c,stdin);

                  return INTLITERAL;
            }

            else if(in_char=='(')
                  return LPAREN;

            else if(in_char==')')
                  return RPAREN;

            else if(in_char==';')
                  return SEMICOLON;
            else if(in_char==’,‘)
                  return COMMA;

            else if(in_char==’+‘)
                  return PLUSOP;

            else if(in_char==’:’)
            {
                  /*        looking for :=   */
                  c=getchar();

                  if(c==’=’)
                        return ASSIGNOP;
                  else
                  {
                        ungetc(c,stdin);
                        lexical_error(in_char);
                  }

                  else if(in_char==’_’)
                  {

                        /* is it comment start? */

                        c=getchar( );
                        if(c==’_’)
                        {
                              do
                                    in_char=getchar();
                              while(in_char != ’\n’);
                        }
                        else
                        {
                              unget(c,stdin);
                              return MINUSOP;
                        }
                  }
                  else
                        lexical_error  (in_char);
            }
      }
}
```

Input :
Program written in High level Language.

Output :

1. Uniform token representation of source program.
2. Tokens may be represented by integer values.

The output will be the input to the Parser for syntax analysis.

Program 2

Scanner programs Using LEX

Problem Definition

The LEX Scanner is a tool that analyzes the source program by reading input program and generates tokens.

Problem Description

LEX is a scanner generator. A LEXER takes a set of description of tokens and produces a function in C language. The token descriptions that lex uses are known as regular expressions. The lexer produced by lex is a C routine called yylex( ). The output of lex program is a C language program called: lex.yy.c.

Library /System Calls

The library associated with LEX is specified by the command –line option -ll. The command for executing lex program is Lex file_ name . l

Psuedocode:

```
E                                            [Ea]
Other Letter                                 [A-DF-Za-df-z]
Digit                                        [0-9]
Letter                                       {E} | { OtherLetter }
IntLit                                       {Digit}+
\%\%
[\t\n]+                                      { /*delete*/ }
[Bb][Ee][Gg][Ii][Nn]                         {minor=0;return(4);}
[Ee][Nn][Dd]                                 {minor=0;return(5);}
[Rr][Ee][Aa][Dd]                             {minor=0;return(6);}
[Ww][Rr][Ii][Tt][Ee]                         {minor=0;return(7);}
{Letter}({Letter}|{Digit}|_)*                {minor=0;return(1);}
{IntLit}                                     {minor=1;return(2);}
({IntLit}[.]{IntLit})({E}[+-]?{IntLit})?    {minor=2;return(2);}
\"([^\"\n]|\"\")*\n                     {stripquotes();minor=0,return(3);}
\"([^\"\n] | \"\")*\"                   {stripquotes();minor=3,return(2);}
\(\                                          {minor=0;return(8);}
\)"                                          {minor=0;return(9);}
\;"                                          {minor=0;return(10);}
\,"                                          {minor=0;return(11);}
\:="                                        {minor=0;return(12);}
\+"                                          {minor=0;return(13);}
\-\                                          {minor=0;return(14);}
\%\%
```

```
/* Strip unwanted quotes from string in yytext; adjust  yyleng */

Void stripquotes(void)
{
      int frompos,topos=0; numquotes=2;

      for ( frompos=1;frompos<yyleng ; frompos++ )
      {
            yytext[topos++] = yytext[frompos];
            if(yytext[frompos]=="" && yytext[frompos+1]=="")
            {
                  frompos++;
                  numquotes++;
            }
      }

      yyleng -= numquotes;
      yytext[yyleng]='\0';
}
```

Input :
The input is a set of token specification in the form of regular expression as shown in psuedocode.

Output :
The output is a C Language program, which will behave as lexical analyzer (scanner) for the given language.

Program 3

Finding FIRST and FOLLOW set of the production

Problem Definition

Computing FIRST AND FOLLOW set of the production

Problem Description

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is X.
2. If X− >e is a production, then add e to FIRST(X).
3. If X is nonterminal and X− >Y1Y2...Yk is a production, then place a in FIRST(X) if for

   some i, a is in FIRST(Yi) and e is in all of FIRST(Y1),...,FIRST(Yi-1) that is, Y1.......Yi-1=∗ >e. If e is in FIRST(Yj) for all j=1,2,...,k, then add e to FIRST(X). For

   example, everything in FIRST(Yj) is surely in FIRST(X). If y1 does not derive e, then we add nothing more to FIRST(X), but if Y1=∗ >e, then we add FIRST(Y2) and so on.

To compute the FIRST(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $inFOLLOW(S), whereSisthestartsymboland$ in the input right endmarker.
2. If there is a production A=>aBs where FIRST(s) except e is placed in FOLLOW(B).
3. If there is aproduction A− >aB or a production A− >aBs where FIRST(s) contains e, then everything in FOLLOW(A) is in FOLLOW(B).

Code for finding FIRST:

```
#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;
     }
}
```

Output :

```
$ ./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

Code for finding FOLLOW:

```
#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]);//Rule1
          }
          for(j=3;prods[i][j]!='\0';j++)
          {
               int temp2=j;
            //Rule-2: production A->xBb then everything in first(b) is in
  follow(B)
               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;
          }
     }
}
```

Output :

```
$ ./a.out
Enter the number of productions
3
Enter grammar
S− >AB
A− >a
B− >b
Follow(S)=S
Follow(A)=b
Follow(B)=Follow(S)
```

Program 4

Top Down parsers

Problem Definition

Construct the recursive descent parser and parse it.

Problem Description

For construction of recursive descent parser we have to write a procedure for each non-terminal that present in the grammar.

Code :

```
#include<stdio.h>
#include<stdlib.h>
char token;
int exp(void);
int term(void);
int fac(void);
void match(char etoken)
{
        if (token==etoken)
          token=getchar();
        else
        {
            printf("ERROR\n");exit(1);
        }
}
main()
{
       token=getchar();
       exp();
       if (token=='\n')
            printf("SUCCESS\n");
       else
            printf("ERROR\n");
}
int exp(void)
{
      term();
      while((token=='+') ||(token=='-'))
         switch(token)
         {
            case '+': match('+');
```

```
                    term();
                     break;

              case '-': match('-');
                    term();
                    break;
         }
}
int term(void)
{
         fac();
         while(token=='*')
         {
               match('*');
               fac();
         }
}
int fac(void)
{
         if (token=='(')
         {
              match('(');
              exp();
              match(')');
         }
         else if(isdigit(token))
         {
              match(token);
         }
         else
         {printf("ERROR\n");exit(1);}
```

Input :
2+3
Success
2+
Error

Program 5

Bottom up Parsers (SLR PARSER)

Problem Definition

Construct SLR parsing table and parses it.

Problem Description

Simple LR parser or **SLR parser** reads a BNF grammar and constructs an LR(0) state machine and computes the look-aheads sets for each reduction in a state.

SLR Parse Table Construction

- item
- item set
- closure (of items)
- goto
- set-of-items construction
- populating the table

Item

Given a production, A $\rightarrow$ XYZ, an LR(0) item is any string of the following form

A $\rightarrow$. XYZ
A $\rightarrow$ X . YZ
A $\rightarrow$ XY . Z
A $\rightarrow$ XYZ .

- The dot means that the input has been seen up to the dot - could have been derived, top down, to that point in the production.

- LR(0) item for A $\rightarrow \varepsilon$ is A $\rightarrow$.

Closure(I), I is a set of items

1. I is in the closure of I Until fixed-point
2. if A $\rightarrow \alpha$. B β is in closure of I, then add B $\rightarrow$. γ to the closure of I

goto(I, X)

If A $\rightarrow \alpha$. X β is in I,
then
add A $\rightarrow \alpha$ X. β J, Goto(I,X) = closure(J)

set-of-items construction

Let C be the sets-of-items (set of item sets)

1. C = closure(S' $\rightarrow$ .S) until fixed-point
2. let c ε C and X a symbol of G', add goto(c, X) to C

SLR table construction

1. Construct sets-of-items
2. Create action table,
   - one row for each item-set, // which forms a state
   - one column for each token + $
3. Create goto table
   - same rows as action
   - one column for nonterminal

Populate the action table:

The ith row, corresponds to the ith item set, Ii

1. If [A $\rightarrow \alpha$. a β] is in Ii, and goto(Ii,a) = Ij
   • then set action[i,a] to shift j,
2. If [A $\rightarrow \alpha$.] is in Ii, and a is in Follow(A)
   • then set action[i,a] to reduce A $\rightarrow \alpha$
3. If [S' $\rightarrow$ S .] is in Ii,
   • then set action[i,$] to accept

Populate the goto table

For each nonterminal A

If goto(Ii,A) = Ij
then goto[i,A]= j

Follow Set for the non terminal associated with the reduction..

Algorithm :

The SLR parsing algorithm

```
Initialize the stack with S
Read input symbol
while (true)
     if Action(top(stack), input) = S
          NS $<-$ Goto(top(stack),input)
          push NS
          Read next symbol
     else if Action(top(stack), input) = Rk
               output k
               pop $|$RHS$|$ of production k from stack
               NS $<-$ Goto(top(stack), LHS_k)
               push NS
         else if Action(top(stack),input) = A
                    output valid, return
               else
                    output invalid, return
```

Example

A grammar that can be parsed by an SLR parser is the following:

(0) S $\rightarrow$ E
(1) E $\rightarrow$ 1 E
(2) E $\rightarrow$ 1

Constructing the action and goto table as is done for LR(0) parsers would give the following item sets and tables:

Item set 0
S $\rightarrow$ $\bullet$ E
E $\rightarrow$ $\bullet$ 1 E
E $\rightarrow$ $\bullet$ 1
Item set 1
E $\rightarrow$ 1 $\bullet$ E
E $\rightarrow$ 1 $\bullet$
E $\rightarrow$ $\bullet$ 1 E
E $\rightarrow$ $\bullet$ 1
Item set 2
S $\rightarrow$ E $\bullet$
Item set 3
E $\rightarrow$ 1 E $\bullet$

The action and goto tables:

	Action		**goto**
State	**1**	**$**	**E**
0	s1		2
1	s1	r2	3
2		Acc	
3		r1	

Input : An augmented grammar G'.

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

Constructing Canonical LR Parsing Tables

Algorithm:

Step 1: Construction of the sets of LR(1) items

```
function closure( I );
begin

        repeat
            for each item [A → $\alpha$. B$\beta$, a] in I,
            each production B → $\gamma$ in G',
            and each terminal b in FIRST($\beta$ a)
            such that [B →. $\gamma$ , b] is not in I do
            add [B →. $\gamma$, b] to I
        until no more sets of items can be added to I
end
return I
end;
```

Step 2:

```
function goto(I, X)
begin

        let J be the set of items [A → X. $\beta$, a] such that
        [A → X $\beta$, a] is in I
        return closure(J)
end;
```

Step 3:

```
procedure items(G')
begin

        C := closure(S'→. S,$);
        repeat
           for each set of items I in C and each grammar symbol X
           such that goto(I , X) is not empty and not in C do
           add goto(I , X) to C
        until no more sets of items can be added to C
end;
```

Step 4: Construction of the canonical LR parsing table.

Method :

1. Construct C=I0,I1. . . . .,In, the collection of sets of LR(1) items for G'.
2. State I of the parser is constructed from Ii. The parsing actions for state I are determined as follows :

   a. If [A → α. a β, b] is in Ii, and goto(Ii, a) = Ij, then set action[i,a] to "shift j." Here, a is required to be a terminal.

   b. If [A → α., a] is in Ii, A ≠ S', then set action[i,a] to "reduce A → α."

   c. If [S'→S.,$] is in Ii, then set action[i ,$] to "accept." If a conflict results from above rules, the grammar is said not to be LR(1), and the algorithm is said to be fail.

3. The goto transition for state i are determined as follows: If goto(Ii , A)= Ij ,then goto[i,A]=j.
4. All entries not defined by rules(2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set containing item [S'→.S, $].

Input: An augmented grammar G'.

Output: The canonical LR parsing table functions action and goto for G'.

Example:

Consider the following augmented grammar:-
S' → S
S → CC
C → Cc | d
The initial set of items is:-
I0 : S' → .S , $
S → .CC, $
C → .Cc, c | d
C → .d, c | d
We have next set of items as:-
I1 : S' → S., $
I2 : S → .Cc, $
C → .Cc, $
C → .d, $
I3 : C → c.C, $
C → .c C , c | d
C → .d, $
I4 : C → d. , c | d
I5 : S → CC. , $
I6 : C → c.C, $
C → .c C ,$
C → .d , $
I7 : C → d. , $
I8 : C → c C. , c | d
I9 : C → c C. , $

Program 6

Parser program using YACC

Problem Definition

yacc is a parser generator. The name is an acronym for ”Yet Another Compiler Compiler.”

Problem Description

- It generates a parser (the part of a compiler that tries to make syntactic sense of the source code) based on an analytic grammar written in a notation similar to BNF. YACC generates the code for the parser in the C programming language.
- The parser generated by yacc requires a lexical analyzer. Lexical analyzer generators, such as Lex or Flex are widely available.
- YACC uses LALR(1) grammars with disambiguating rules.
- Every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent “%%” marks

In other words, a full specification file looks like

Declarations
%%
Rules
%%
programs

Library / System Calls

The library for running YACC program is specified by the option –ly. The command for executing a YACC source file is : yacc filename . y

Program :

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

int  regs[26];
int  base;

%}
```

```
%start  list

%token  DIGIT  LETTER

%left  '+'  '-'
%left  '*'  '/'
%left  UMINUS      /*  supplies  precedence  for  unary  minus  */

%%      /*  beginning  of  rules  section  */

list :    /*  empty  */
     |    list  stat  '\n'
     |    list  error  '\n'
              {    yyerrok;  }
     ;

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

     ;

expr :    '('  expr  ')'
              {    $$  =  $2;  }
     |    expr  '+'  expr
              {    $$  =  $1  +  $3;  }
     |    expr  '-'  expr
              {    $$  =  $1  -  $3;  }
     |    expr  '*'  expr
              {    $$  =  $1  *  $3;  }
     |    expr  '/'  expr
              {    $$  =  $1  /  $3;  }

     |    number
     ;

number    :    DIGIT
              {    $$ = $1;    base  =  ($1==0)  ?  8  :  10;  }
     |    number  DIGIT
              {    $$  =  base * $1  +  $2;  }
     ;

%%      /*  start  of  programs  */

yylex( ) {

     int  c;
while( (c=getchar()) == ' ' ) {/* skip blanks */ }

     if( islower( c ) ) {
     yylval = c - 'a';
     return ( LETTER );
     }
if( isdigit( c ) ) {
     yylval = c - '0';
     return( DIGIT );
     }
return( c );
}
```

Input: 3+6

Output: 9

Program 7

Intermediate Code generation

Problem Definition

To generate the intermediate code that is machine independent code to achieve portability in compilers.

Problem Description

We could translate the source program directly into the target language. However, there are benefits to having an intermediate, machine-independent representation.

- A clear distinction between the machine-independent and machine-dependent parts of the compiler
- Retargeting is facilitated the implementation of language processors for new machines will require replacing only the back-end.
- We could apply machine independent code optimization techniques Intermediate representations span the gap between the source and target languages

Code:

```
#define SIZE 50            /* Size of Stack */
#include <ctype.h>
char s[SIZE];
int top=-1;       /* Global declarations */

push(char elem)
{                       /* Function for PUSH operation */
    s[++top]=elem;
}

char pop()
{                      /* Function for POP operation */
    return(s[top--]);
}

int pr(char elem)
{                 /* Function for precedence */
    switch(elem)
    {
    case '#': return 0;
    case '(': return 1;
    case '+':
    case '-': return 2;
    case '*':
    case '/': return 3;
    }
}


main()
{                         /* Main Program */
    char infx[50],pofx[50],ch,elem;
    int i=0,k=0;
    printf("\n\nRead the Infix Expression ? ");
    scanf("%s",infx);
    push('#');
    while( (ch=infx[i++]) != '\0')
    {
        if( ch == '(') push(ch);
        else
            if(isalnum(ch)) pofx[k++]=ch;
            else
                if( ch == ')')
                {
                    while( s[top] != '(')
                        pofx[k++]=pop();
                    elem=pop(); /* Remove ( */
                }
                else
                {       /* Operator */
                    while( pr(s[top]) >= pr(ch) )
                        pofx[k++]=pop();
                    push(ch);
                }
    }
    while( s[top] != '#')     /* Pop from stack till empty */
        pofx[k++]=pop();
    pofx[k]='\0';          /* Make pofx as valid string */
    printf("\n\nGiven Infix Expn: %s  Postfix Expn: %s\n",infx,pofx);
}
```

Input:
a+b

output:
ab+

Program 8

Code Optimization

Problem Definition

Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output.

Problem Description

Classification of optimizations types

Optimizations that are performed automatically by a compiler or manually by the programmer, can be classified by various characteristics.

The "scope" of the optimization:

1. Local optimizations - Performed in a part of one procedure.
   a. Common sub-expression elimination (e.g. those occurring when translating array indices to memory addresses.
   b. Using registers for temporary results, and if possible for variables.
   c. Replacing multiplication and division by shift and add operations.
2. Global optimizations - Performed with the help of data flow analysis (see below) and split-lifetime analysis.
   a. Code motion (hoisting) outside of loops
   b. Value propagation
   c. Strength reductions
3. Inter-procedural optimizations

PROGRAM

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

main()
{
    char str[25][50], forLoopParam[90], rightHandParam[10][40],
         leftHandParam[90];
    int j=0,k=0,i=0,m=0,n=0,q=0,s=0;
    int flag[10]={0},count[10]={0};
    printf("\n Input the loop to be optimized:\n");
```

```
/* PROCESSING FIRST LINE -- FOR DECLARATION*/

    gets(str[0]);
    while(str[k][i++]!=';');
    while(str[k][i++]!=';');

    while(str[k][i]!=')')
    {
        if(isalpha(str[k][i]))
            forLoopParam[j++]=str[k][i];
          i++;
    }
    i=0;

/* J IS INDEX TO CHANGEARRAY*/

/* PROCESSING SECOND LINE --{*/

    puts(str[0]);
    gets(str[0]);

    while(str[0][i++]!='{');
    puts(str[0]);
    k=0;
    while(gets(str[k]) && str[k][0]!='}')
    {
        while(str[k][i++]!='=');
        leftHandParam[n++]=str[k][i-2];
        k++;
        i=0;
    }
/*N IS INDEX TO TEMPCHANGE ARRAY*/
/* TEMPCHANGE ARRAY STORES LEFT SIDE PARAMETERS OF
ASSIGNMENT OPERATIONS */

     for(m=0,i=0;m<k;m++)
     {
          while(str[m][i++]!='=');
          while(str[m][i]!=';')
          {
               if(isalpha(str[m][i]))
                    rightHandParam[m][count[m]++]=str[m][i];
               i++;
          }
          i=0;
     }
/* Q IS INDEX TO TEMP2*/
/* TEMP2 STORES RIGHT HAND SIDE PARAMETERS OF
STATEMENTS*/
/*COMPARING LEFT-HAND PARAMETERS WITH RIGHT-HAND
PARAMETERS*/

     for(m=0;m<k;m++)
          for(s=0;s<count[m];s++)
               for(i=0;i<n;i++)
               {
                    if(rightHandParam[m][s]==leftHandParam[i])
                         flag[m]=1;
               }

//COMPARING LEFT-HAND SIDE PARAMETERS WITH FOR-LOOP
PARAMETERS

     for(i=0;i<k;i++)
          for(q=0;q<j;q++)
          {
               if(leftHandParam[i]==forLoopParam[q])
                    flag[i]=1;
          }

//COMPARING RIGHT-HAND PARAMETERS WITH FOR-LOOP
PARAMETERS

     for(m=1;m<k;m++)
          for(s=0;s<count[m];s++)
               for(i=0;i<j;i++)
               {
                    if(rightHandParam[m][s]==forLoopParam[i])
               }

//DISPLAYING STATEMENTS OF FOR LOOP WHICH CANT BE
//OPTIMIZED IN THE FOR LOOP

     printf("the stmts that cant be optimized\n");

     for(i=0;i<k;i++)
          if(flag[i]==1)
               puts(str[i]);

//DISPLAYING STATEMENTS OF FOR LOOP WHICH CANT BE
OPTIMIZED OUTSIDE FOR LOOP
```

```
     puts(str[k]);  //DISPLAYING THE END-FLOWER BRACE
     printf("the stmts that can be optimized");
     for(i=0;i<k;i++)
          if(flag[i]==0)
               printf("\t\t %s \t\t",str[i]);
//   puts(str[i]);
}
```

Input:

```
For(i=0;i<9;i++)
{
c=m+n;
i=i+12;
}
```

Output:

```
the stmts that cant be optimized
i=i+12;
the stmts that can be optimized
c=m+n;
```

Program 9

CODE GENERATION

Problem Definition

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.

Problem Description

The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted into linear sequence of instructions, usually in an intermediate language such as three address code for example, the tree W := ADD(X,MUL(Y,Z)) might be transformed into a linear sequence of instructions by recursively generating the sequences for t1 := X and t2 := MUL(Y,Z), and then emitting the instruction ADD W, t1, t2.

PROGRAM :

```
#include<stdio.h>
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()
{
        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--;
```

```
                                        if(operation=='S'&&stktop>=-1)
                                        {
                                              printf("N\n");
                                              operation='A';
                                        }
                                       printf("%c %s\n",operation,op1);
                                       val=checknstore(msg[i+1]);
                                   }//while
                              }//else
                         }//for
                    }//while
               }//main
```

Input: AB+

Output: L B
A A

Annexure – I

List of programs according to syllabus(Osmania University).

1. Scanner programs using C.
2. Scanner programs using LEX.
3. Finding FIRST set and FOLLOW set of the production.
4. Top Down Parsers.
5. Bottom up Parsers
6. Parser programs using YACC.
7. Intermediate code generation
8. code optimization.