ILAHIA COLLEGE OF ENGINEERING AND TECHNOLOGY

Mulavoor P.O, Muvattupuzha-686673

(Approved by AICTE, New Delhi and Affiliated to APJ Abdul Kalam Technological University, Thiruvananthapuram)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

ON

CS431 COMPILER DESIGN LAB

VISION

To become a global technical institution of high repute by nurturing spirit of academic excellence and pursuit of advance technical research by imparting timeless core values to the learners to serve the humankind.

MISSION

To transform the learners into exceptional technocrats and entrepreneurs capable of meeting ever challenges in the global society, by continually imparting high quality outcome based technical education by:

- Incorporating best possible innovative instructional techniques.
- Providing a strong academic foundation, technical skills and promoting research and development activities.
- Developing leadership qualities, soft skills and building spirit for teamwork.
- Inculcating professional ethics, critical thinking mind and positive attitude of lifelong learning.

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

VISION

*"*To mould learners as excellent technocrats in the field of Computer Science and Engineering through quality education, lifelong learning, research and development with human values.*"*

MISSION

- To impart quality education through theory and practical sessions, value added courses and timely update of knowledge.
- Focus on innovation in research with recent trends by inculcating industry institute Interaction.
- Promote learners in Innovation and Entrepreneurship activities, placement and higher studies.
- To thrive socially committed professionals with human values and ethics

PROGRAM EDUCATIONAL OBJECTIVES (PEO)

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

1. Establish the graduates to solve the real world challenges through use of Computer knowledge and skills.
2. Graduates will have the ability to foster wide range of multi disciplinary projects with professional ethics for the welfare of humankind.
3. Make the graduates as good professionals by shaping their hard and soft skills.

PROGRAM SPECIFIC OBJECTIVES (PSO)

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

1. Our students will accomplish the skill of entrepreneurship by providing easy and feasible solutions for the advanced technology requirements with the aid of proper priority planning, strategic thinking and issue forecasting.

2. Proficiency in multiple skill sets which enable the learners to excel in diverse platforms in industries.

PROGRAM 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 modelling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

SYLLABUS

Course code	Course Name	L-T-P-Credits	Year of Introduction
IT431	**Compiler Design Lab**	**0-0-3-1**	**2016**

Prerequisite: CS331 System Software Lab

Course Objectives

- To implement the different Phases of compiler.
- To implement and test simple optimization techniques
- To give exposure to compiler writing tools.

List of Exercises / Experiments (Minimum 8 are mandatory)

1. Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.
2. Implementation of Lexical Analyzer using Lex Tool
3. Generate YACC specification for a few syntactic categories.
   a) Program to recognize a valid arithmetic expression that uses operator +, – , * and /.
   b) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
   c) Implementation of Calculator using LEX and YACC.
   d) Convert the BNF rules into YACC form and write code to generate abstract syntax tree.
4. Write program to find ε – closure of all states of any given NFA with ε transition.
5. Write program to convert NFA with ε transition to NFA without ε transition.
6. Write program to convert NFA to DFA.
7. Write program to minimize any given DFA.
8. Develop an operator precedence parser for a given language.
9. Write program to find Simulate First and Follow of any given grammar.
10. Construct a recursive descent parser for an expression.
11. Construct a Shift Reduce Parser for a given language.
12. Write a program to perform loop unrolling.
13. Write a program to perform constant propagation.
14. Implement Intermediate code generation for simple expressions.
15. Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc.

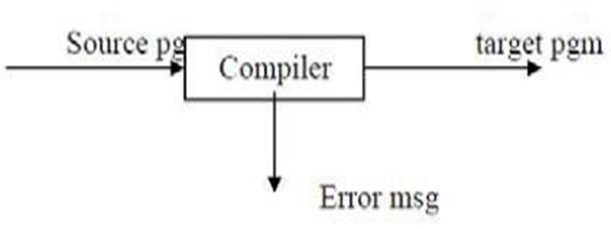
Expected Outcome

By the end of the course, the student will be able to:

i. Implement the techniques of Lexical Analysis and Syntax Analysis.
ii. Apply the knowledge of Lex & Yacc tools to develop programs.
iii. Generate intermediate code.
iv. Implement Optimization techniques and generate machine level code.

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.



STRUCTURE OF THE COMPILER DESIGN

Phases 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 phases of compilation.
a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)
Compilation process is partitioned into no-of-sub processes called **„phases**".

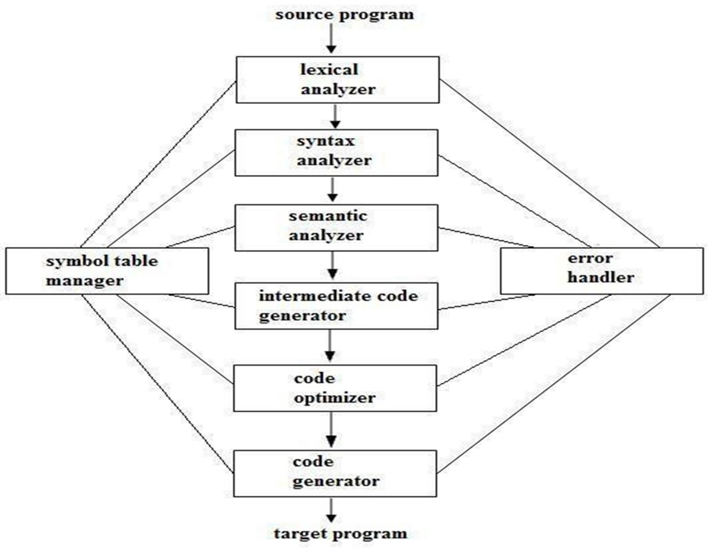


Fig 1.5 Phases of a compiler

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

Syntax Analysis:-
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.

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

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

Code Generation:-
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.

PROGRAM 1

Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.

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

FILE *fp; int
lineno=0;
char c;
char lexbuf[50],symtab[50][20];
int i=0,x;
char kw[30][20]={"void","int","float","double","short","long","if","else","switch","case","break","ret
urn","main","static","goto"};
char delim[]={'(',')','{','}','[',']',';',','};
char oper[]={'+','=','-','*','/','<','>'};
int isdelim(char);
int isoper(char);
int iskw(char[]);
void main()
{
fp=fopen("sample.c","r");
c=getc(fp);
while(c!=EOF)
{
if(c=='
'||c=='\t'); else
if(c=='\n')
{
lineno++;
}
else if((x=isdelim(c))!=-1)
{
printf("%c\t\tDelimiter\n",c);
}
else if((x=isoper(c))!=-1)
{
printf("%c\t\tOperator\n",c);
}
else if(isdigit(c))
{
int b=0;
while(isdigit(c))
{
```

```
lexbuf[b++]=c;
c=getc(fp);
}
ungetc(c,fp);
lexbuf[b]='\0';
printf("%s\t\tDigit\n",lexbuf)
;
}
else if(isalpha(c))
{
int b=0,k;
while(isalpha(c)||isdigit(c)||c=='_
')
{
lexbuf[b++]=c;
c=getc(fp);
}
ungetc(c,fp);
lexbuf[b]='\0'; if((!(lookup(lexbuf)))&&(!iskw(lexbuf)))
{
strcpy(symtab[i++],lexbuf);

}
if((k=iskw(lexbuf))!=0)
{
printf("%s\t\t Keyword\n",lexbuf);
}
else
{
printf("%s\t\tIdentifier\n",lexbuf);
}
}
c=getc(fp);
}
fclose(fp);
printf("\nNumber of lines=%d\n",lineno-1);
}
//Is delimiter
int isdelim(char d)
{
int k;
for(k=0;k<8;k++)
{
if(d==delim[k])
{
return k;
}
```

```
}
return -1;
}

//Is operator
int isoper(char op)
{
int k;
//printf("%c\n",op)
; for(k=0;k<7;k++)
{
if(op==oper[k])
{
return k;
}

}
return -1;
}

int lookup(char s[])
{
int k;
for(k=0;k<i;k++)
{
if((strcmp(s,symtab[k]))==0)
{
return k+1;
}
return 0;
}
}
int iskw(char s[])
{
int k;
for(k=0;k<15;k++)
{
if(strcmp(s,kw[k])==0)
return k+1;
}
return 0;
}
```

PROGRAM 2

Implementation of Lexical Analyzer using Lex

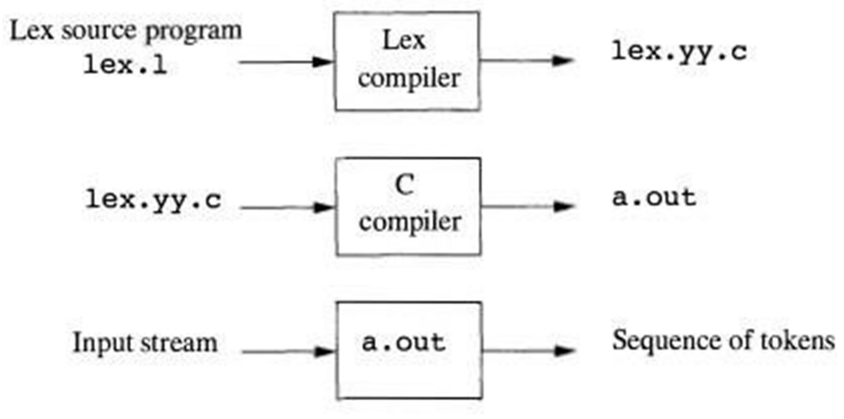
Tool Structure of Lex Programs
A lex program has the following form:
Declarations
%%
Translation rules
%%
Auxiliary functions
Each section is divided with %%

Declaration Section
It includes declaration of variables used in the code segment and regular definitions .Variable declarations are specified within %{ and %}.

Translation rules
Translation rules each have the form
{Pattern} Action
Each pattern is a regular expression, which may use the regular definitions of the declaration section.

Auxiliary functions
It holds whatever additional functions are used in the actions. The lexical analyzer created by Lex behaves in concert with the parser .When called by the parser; the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns Pi. It then executes the associated action Ai. The lexical analyzer returns a single value, t he token name to the parser, but uses the shared variable yylavl to pass additional information about the lexeme found, if needed.

LEX-Lexical Analyzer Generator
Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions .An input file, which we call lex.l, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c. The latter file is compiled by the C compiler into a file called a.out, as always. The C-compiler output is a working lexical analyzer



Lex program to count the number of characters in a file.

```
%{
int c=0;
%}
%%
[A-Za-z] c++;
. ;
%%
int main()
{
yyin=fopen("b.c","r");
yylex();
printf("count is %d\n",c);
}
int yywrap()
{
return 1;
}
```

Lex program to count the digits in a file.

```
%{
int c=0;
%}
digit [0-9]

%%
{digit} c++;




. ;
%%
int main()
{
yyin=fopen("b.c","r");
yylex();
printf("count is %d\n",c);
}
int yywrap()
{
return 1;
}
```

Lex program to count number of a"s in a program

```
%{
int c=0,d=0;
%}
digit [0-9]

%%
a c++;
. ;
%%
int main()
{
yyin=fopen("b.c","r");
yylex();
printf("count is %d\n",c);
}
int yywrap()
{
return 1;
}
```

```
%{
int negative=0;
int positive=0;
int positivefraction=0;
%}
%%
[-][0-9]+ {negative=negative+1;}
[+][0-9]+ {positive=positive+1;}
[+][0-9]+[.][0-9]+ {positivefraction=positivefraction+1;}
```

```
. ;
%%
int main()
{
yyin=fopen("b.c","r");
yylex();
printf("count of negative no is %d\n",negative);
printf("count of positive no is %d\n",positive);
printf("count of positive fraction is %d\n",positivefraction);
}
int yywrap()
{
return 1;
}
```

Lex program to remove comment line

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


%%
\/\/.* ;
%%
int main()
{
yyin=fopen("b.c","r");
yyout=fopen("c.c","w");
yylex();
}
int yywrap()
{
return 1;
}
```

Lex program to count the number of identifier.

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

%%
[a-z_][a-z_0-9]* {c=c+1;}
.|\n ;
%%
int main()
{
yyin=fopen("b.c","r");
yylex();
printf("count is %d\n",c);
}
```

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

Yacc Yet Another Compiler Compiler(Parser Generator)

Syntax

```
… definitions …
%%
… rules …
%%
… subroutines …
```

Input to yacc is divided into three sections. The definitions section consists of token declarations and C code bracketed by “ %{ “ and “ %} ”. The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.

```
%{
        //Header file
        //variable declarations
%}

%left           //for left - associative

%right // right associative
%token token_name //defining token
```

Eg.

```
%token NUMBER
%left    '+' '-'

%%
         // Grammar for example expr: expr '+' expr { $$ = $1 + $3; }
                                       |
                                       expr '*' expr { $$ = $1 * $3; }
                                       ;
```

// “$1” for the first term on the right - hand side of the production, “$2 ” for the second, and so on. “$$” designates the top of the stack after reduction has taken place.

%%

```
E -> E + E
E -> E * E
```

E -> id

Three productions have been specified. Terms that appear on the left - hand side (LHS) of a production, such as E (expression) are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right - hand side (RHS) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier.

Lex includes this file and utilizes the definitions for token values. To obtain tokens yacc calls yylex. Function yylex has a return type of int that returns a token. Values associated with the token are returned by lex in variable yylval.
For example,

```
[0-9]+ {yylval = atoi(yytext);
            return INTEGER;
            }
```

would store the value of the integer in yylval, and return token INTEGER to yacc.

Internally yacc maintains two stacks in memory; a parse stack and a value stack. Theparse stack contains terminals and nonterminals that represent the current parsing state. The value stack is an array of YYSTYPE elements and associates a value yylval with each element in the parse stack. For example when lex returns an INTEGER token yacc shifts this token to the parse stack. At the same time the corresponding yylval is shifted to the value stack. The parse and value stacks are always synchronized so finding a value related to a token on the stack is easily accomplished.

When we apply the rule **expr: expr '+' expr { $$ = $1 + $3; }.**We replace the right-hand side of the production in the parse stack with the lef-hand side of the same production. In this case pop “expr '+' expr” and push “expr”. We have reduced the stack by popping three terms off the stack nd pushing back one term. We may reference positions in the value stack in our C code by specifying “$1” for the first term on the right - hand side of the production, “$2 ” for the second, and so on. “$$” designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. As a conse quence the parse and value stacks remain synchronized.

PROGRAM 3

Generate YACC specification for a few syntactic categories.
a) Implementation of Calculator using LEX and YACC
calculator.y

```
%{
#include <stdio.h>
//extern FILE *yyin;
%}
%token NUMBER
%start S
%%

S      : E                    { printf("Expression_value= %d\n", $1); }
         ;
E      : E '+' NUMBER            { $$ = $1 + $3;
                         printf ("Recognized '+' expression.\n");
                         }
         | E '-' NUMBER          { $$ = $1 - $3;
                         printf ("Recognized '-' expression.\n");
                         }
         | E '*' NUMBER { $$ = $1 * $3;
                         printf ("Recognized '*' expression.\n");
                         }
         | E '/' NUMBER { if($3==0)
                         {
                         printf("Cannot divide by 0");
                         break;
                         }
                         else
                         $$ = $1 / $3;
                         printf ("Recognized '/' expression.\n");

                      }
         | NUMBER     { $$ = $1;
                         printf ("Recognized a number.\n");
                         }
         ;
%%
int main ()
         {
         //yyin=fopen("s.txt","r");
         do
         {printf("Enter the expression\n");
          yyparse();
```

```
            }while(1);
            return 1;

            }

int yyerror (char *msg)
            {
                        printf("Invalid Expression\n");
             }
 yywrap()
{
  return(1);
}
```

calculator.l

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ { yylval = atoi (yytext);
            printf ("scanned the number %d\n",
            yylval); return NUMBER; }
[ \t]       { printf ("skipped whitespace\n"); }
\n          { printf ("reached end of line\n");
            return 0;
            }
.           { printf ("found other data \"%s\"\n",
            yytext); return yytext[0];
            /* so yacc can see things like '+', '-', and '=' */
            }
%%
```

How to run

```
yacc -d
calculator.y lex
calculator.l
gcc y.tab.c lex.yy.c
./a.out
```

b.Program to recognize a valid variable which starts with a letter followed by any number of letters or digits

```
valid.y
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token DIGIT LETTER
%start S
%%

S         : variable    { printf("Valid Variable\n"); }
    ;
variable           : LETTER alphanumeric
            ;

alphanumeric :LETTER alphanumeric
            |DIGIT alphanumeric
            |LETTER
            |DIGIT
            ;
%%
int main ()
    {
    do
    {printf("Enter the expression\n");
     yyparse();
    }while(1);
    return 1;

    }

int yyerror (char *msg)
    {
            printf("Invalid Expression\n");
    }
yywrap()
{
  return(1);
}
```

```
valid.l

%{
#include "y.tab.h"
extern int yylval;
%}
%%
[a-zA-Z] {return LETTER;}
[0-9] {return DIGIT;}
\n    { printf ("reached end of
   line\n"); return 0;
   }
.   { printf ("found other data \"%s\"\n",
   yytext); return yytext[0];
   /* so yacc can see things like '+', '-', and '=' */
   }
```

c.Program to recognize a valid arithmetic expression that uses operator +,-,*and /

arithmetic.y

```
%{
#include<stdio.h>
%}
%token ID NUMBER
%left '+' '-'
%left '*' '/'
%%
stmt:expr {printf("valid Expression\n");}
         ;
expr: expr '+' expr
         | expr '-' expr
         | expr '*' expr
         | expr '/' expr
         | '(' expr ')'
         | NUMBER
         | ID
         ;
%%
int main ()
   {
   do
   {printf("Enter the expression\n");
    yyparse();
   }while(1);
   return 1;

   }

int yyerror (char *msg)
   {
```

```
            printf("Invalid Expression\n");
    }
yywrap()
{
  return(1);
}
```

```
arithmetic.l
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[a-zA-Z] {return ID;}
[0-9] {return NUMBER;}
\n     { printf ("reached end of
    line\n"); return 0;
    }
.    { printf ("found other data \"%s\"\n",
    yytext); return yytext[0];
    /* so yacc can see things like '+', '-', and '=' */
    }
```

PROGRAM 4

Find ε – closure of all states of any given NFA with ε transition.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LEN 100

char NFA_FILE[MAX_LEN];
char buffer[MAX_LEN];
int zz = 0;

// Structure to store DFA states and their
// status ( i.e new entry or already
present) struct DFA {
  char
  *states; int
  count;
} dfa;

int last_index = 0;
FILE *fp;
int symbols;

/* reset the hash map*/
void reset(int ar[], int size) {
  int i;

  // reset all the values of
  // the mapping array to
  zero for (i = 0; i < size; i++)
  { ar[i] = 0;
  }
}

// Check which States are present in the e-closure

/* map the states of NFA to a hash
set*/ void check(int ar[], char S[]) {
  int i, j;

  // To parse the individual states of
  NFA int len = strlen(S);
  for (i = 0; i < len; i++) {

    // Set hash map for the position
    // of the states which is found
    j = ((int)(S[i]) - 65);
    ar[j]++;
  }
}

// To find new Closure States
void state(int ar[], int size, char S[]) {
  int j, k = 0;

  // Combine multiple states of NFA
  // to create new states of
  DFA for (j = 0; j < size; j++) {
    if (ar[j] != 0)
      S[k++] = (char)(65 + j);
  }

  // mark the end of the
  state S[k] = '\0';
}

// To pick the next closure from closure
set int closure(int ar[], int size) {
  int i;

  // check new closure is present or
  not for (i = 0; i < size; i++) {
    if (ar[i] == 1)
      return i;
  }
  return (100);
}

// Check new DFA states can be
// entered in DFA table or not
int indexing(struct DFA *dfa) {
int i;

  for (i = 0; i < last_index; i++) {
    if (dfa[i].count == 0)
      return 1;
  }
  return -1;
}

/* To Display epsilon closure*/
void Display_closure(int states, int closure_ar[],
                  char *closure_table[],
                  char *NFA_TABLE[][symbols + 1],
                  char *DFA_TABLE[][symbols]) {
  int i;
  for (i = 0; i < states; i++) {
    reset(closure_ar, states);
    closure_ar[i] = 2;

    // to neglect blank entry
    if (strcmp(&NFA_TABLE[i][symbols], "-") != 0) {

      // copy the NFA transition state to buffer
      strcpy(buffer, &NFA_TABLE[i][symbols]);
      check(closure_ar, buffer);
      int z = closure(closure_ar, states);

      // till closure get completely
      saturated while (z != 100)
      {
        if (strcmp(&NFA_TABLE[z][symbols], "-") != 0) {
          strcpy(buffer, &NFA_TABLE[z][symbols]);


          // call the check
          function
          check(closure_ar, buffer);
        }
        closure_ar[z]++;
        z = closure(closure_ar, states);
      }
    }

    // print the e closure for every states of NFA
    printf("\n e-Closure (%c) :\t", (char)(65 + i));


    bzero((void *)buffer, MAX_LEN);
    state(closure_ar, states, buffer);
    strcpy(&closure_table[i], buffer);
    printf("%s\n", &closure_table[i]);
  }
}


/* To check New States in DFA */
int new_states(struct DFA *dfa, char S[]) {

  int i;

  // To check the current state is already
  // being used as a DFA state or not in
  // DFA transition table
  for (i = 0; i < last_index; i++) {
    if (strcmp(&dfa[i].states, S) == 0)
      return 0;
  }

  // push the new
  strcpy(&dfa[last_index++].states, S);

  // set the count for new states entered
  // to zero
  dfa[last_index - 1].count = 0;
  return 1;
}

// Transition function from NFA to DFA
// (generally union of closure operation )
void trans(char S[], int M, char *clsr_t[], int st,
               char *NFT[][symbols + 1], char TB[]) {
  int len = strlen(S);
  int i, j, k, g;
  int arr[st];
  int sz;
  reset(arr,
  st);
  char temp[MAX_LEN], temp2[MAX_LEN]; char
  *buff;

  // Transition function from NFA to
  DFA for (i = 0; i < len; i++) {

    j = ((int)(S[i] - 65));
    strcpy(temp, &NFT[j][M]);

    if (strcmp(temp, "-") != 0) {
      sz = strlen(temp);
      g = 0;

      while (g < sz) {
        k = ((int)(temp[g] - 65));
        strcpy(temp2, &clsr_t[k]);
        check(arr, temp2);
        g++;
      }
    }
  }
  bzero((void *)temp, MAX_LEN);
  state(arr, st, temp);
  if (temp[0] != '\0') {
    strcpy(TB, temp);
  } else strcpy(TB,
    "-");
}

/* Display DFA transition state table*/
void Display_DFA(int last_index, struct DFA *dfa_states,
                 char *DFA_TABLE[][symbols]) {
  int i, j; printf("\n\n*******************************************************\n\n");
  printf("\t\t DFA TRANSITION STATE TABLE \t\t \n\n");
  printf("\n STATES OF DFA :\t\t");

  for (i = 1; i < last_index; i++) printf("%s,
  ", &dfa_states[i].states); printf("\n");
  printf("\n GIVEN SYMBOLS FOR DFA: \t");

  for (i = 0; i < symbols; i++)
    printf("%d, ", i);
  printf("\n\n");
  printf("STATES\t");

  for (i = 0; i < symbols; i++)
    printf("|%d\t", i);
  printf("\n");

  // display the DFA transition state
  table  printf("--------------------------------+
  -----------------------------------------------\n")
  ; for (i = 0; i < zz; i++) {
    printf("%s\t", &dfa_states[i + 1].states);
    for (j = 0; j < symbols; j++) { printf("|%s
    \t", &DFA_TABLE[i][j]);
    }
    printf("\n");
  }
}

// Driver
Code int
main() { int i,
j, states;
  char T_buf[MAX_LEN];
```

```
// creating an array dfa structures
struct DFA *dfa_states = malloc(MAX_LEN * (sizeof(dfa)));
states = 6, symbols = 2;

printf("\n STATES OF NFA :\t\t");
for (i = 0; i < states; i++)

  printf("%c, ", (char)(65 + i));
printf("\n");
printf("\n GIVEN SYMBOLS FOR NFA: \t");

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

  printf("%d, ", i);
printf("eps");
printf("\n\n");
char *NFA_TABLE[states][symbols + 1];

// Hard coded input for NFA table
char *DFA_TABLE[MAX_LEN][symbols];
strcpy(&NFA_TABLE[0][0], "FC");
strcpy(&NFA_TABLE[0][1], "-");
strcpy(&NFA_TABLE[0][2], "BF");
strcpy(&NFA_TABLE[1][0], "-");
strcpy(&NFA_TABLE[1][1], "C");
strcpy(&NFA_TABLE[1][2], "-");
strcpy(&NFA_TABLE[2][0], "-");
strcpy(&NFA_TABLE[2][1], "-");
strcpy(&NFA_TABLE[2][2], "D");
strcpy(&NFA_TABLE[3][0], "E");
strcpy(&NFA_TABLE[3][1], "A");
strcpy(&NFA_TABLE[3][2], "-");
strcpy(&NFA_TABLE[4][0], "A");
strcpy(&NFA_TABLE[4][1], "-");
strcpy(&NFA_TABLE[4][2], "BF");
strcpy(&NFA_TABLE[5][0], "-");
strcpy(&NFA_TABLE[5][1], "-");
strcpy(&NFA_TABLE[5][2], "-");
printf("\n NFA STATE TRANSITION TABLE \n\n\n");
printf("STATES\t");

for (i = 0; i < symbols; i++)
  printf("|%d\t", i);
printf("eps\n");
```

```
  // Displaying the matrix of NFA transition table
  printf("__________+_______________________________________\n");
  for (i = 0; i < states; i++) {
    printf("%c\t", (char)(65 + i));

    for (j = 0; j <= symbols; j++) {
      printf("|%s \t", &NFA_TABLE[i][j]);
    }
    printf("\n");
  }
  int closure_ar[states];
  char *closure_table[states];

  Display_closure(states, closure_ar, closure_table, NFA_TABLE, DFA_TABLE);
  strcpy(&dfa_states[last_index++].states, "-");

  dfa_states[last_index - 1].count =
  1; bzero((void *)buffer, MAX_LEN);

  strcpy(buffer, &closure_table[0]);
  strcpy(&dfa_states[last_index++].states, buffer);

  int Sm = 1, ind =
  1; int start_index =
  1;

}
```

OUTPUT

```
STATES OF NFA :             A, B, C, D, E, F,

 GIVEN SYMBOLS FOR NFA: 0, 1, eps


 NFA STATE TRANSITION TABLE


STATES       |0      |1      eps
__________+_______________________________________
A       |FC     |-      |B
                        F
B       |-      |C      |-
C       |-      |-      |D
D       |E      |A      |-
E       |A      |-      |B
                        F
F       |-      |-      |-
```

e-Closure (A) :	ABF
e-Closure (B) :	B
e-Closure (C) :	CD
e-Closure (D) :	D
e-Closure (E) :	BEF
e-Closure (F) :	F

PROGRAM 5

Develop a program to convert NFA to DFA

PROGRAM LOGIC:

Step 1 : Take ∈ closure for the beginning state of NFA as beginning state of DFA.

Step 2 : Find the states that can be traversed from the present for each input symbol (union of transition value and their closures for each states of NFA present in current state of DFA).

Step 3 : If any new state is found take it as current state and repeat step 2.

Step 4 : Do repeat Step 2 and Step 3 until no new state present in DFA transition table.

Step 5 : Mark the states of DFA which contains final state of NFA as final states of DFA.

PROGRAM

```
// C Program to illustrate how to convert e-nfa to DFA

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LEN 100

char NFA_FILE[MAX_LEN];
char buffer[MAX_LEN];
int zz = 0;
// Structure to store DFA states and their
// status ( i.e new entry or already
present) struct DFA {
char
*states; int
count;
} dfa;
int last_index = 0;
FILE *fp;
```

```
int symbols;

/* reset the hash map*/
void reset(int ar[], int size) {
int i;
// reset all the values of
// the mapping array to
zero for (i = 0; i < size; i++)
{
        ar[i] = 0;
}
}
// Check which States are present in the e-closure
/* map the states of NFA to a hash
set*/ void check(int ar[], char S[]) {
int i, j;
// To parse the individual states of NFA
int len = strlen(S);
for (i = 0; i < len; i++) {
        // Set hash map for the position
        // of the states which is
        found j = ((int)(S[i]) - 65);
        ar[j]++;
}
}
// To find new Closure States
void state(int ar[], int size, char S[]) {
int j, k = 0;
// Combine multiple states of NFA
// to create new states of
DFA for (j = 0; j < size; j++) {
        if (ar[j] != 0)
```

```
            S[k++] = (char)(65 + j);
}
// mark the end of the
state S[k] = '\0';
}
// To pick the next closure from closure
set int closure(int ar[], int size) {
int i;
// check new closure is present or
not for (i = 0; i < size; i++) {
            if (ar[i] == 1)
            return i;
}
return (100);
}
// Check new DFA states can be
// entered in DFA table or not
int indexing(struct DFA *dfa) {
int i;
for (i = 0; i < last_index; i++) {
            if (dfa[i].count == 0)
            return 1;
}
return -1;
}
/* To Display epsilon closure*/
void Display_closure(int states, int closure_ar[],
                                                char *closure_table[],
                                                char *NFA_TABLE[][symbols + 1],
                                                char *DFA_TABLE[][symbols]) {
int i;
```

```
for (i = 0; i < states; i++) {
        reset(closure_ar, states);
        closure_ar[i] = 2;
        // to neglect blank entry
        if (strcmp(&NFA_TABLE[i][symbols], "-") != 0) {
        // copy the NFA transition state to buffer
        strcpy(buffer, &NFA_TABLE[i][symbols]);
        check(closure_ar, buffer);
        int z = closure(closure_ar, states);

        // till closure get completely
        saturated while (z != 100)
        {
                if (strcmp(&NFA_TABLE[z][symbols], "-") != 0) {
                strcpy(buffer, &NFA_TABLE[z][symbols]);
                // call the check
                function
                check(closure_ar, buffer);
                }
                closure_ar[z]++;
                z = closure(closure_ar, states);
        }
        }
        // print the e closure for every states of
        NFA printf("\n e-Closure (%c) :\t", (char)(65
        + i)); bzero((void *)buffer, MAX_LEN);
        state(closure_ar, states, buffer);
        strcpy(&closure_table[i], buffer);
        printf("%s\n", &closure_table[i]);
}
}
/* To check New States in DFA */
```

```
int new_states(struct DFA *dfa, char S[]) {
int i;
// To check the current state is already
// being used as a DFA state or not in
// DFA transition table
for (i = 0; i < last_index; i++) {
        if (strcmp(&dfa[i].states, S) == 0)
        return 0;
}
// push the new
strcpy(&dfa[last_index++].states, S);
// set the count for new states entered
// to zero
dfa[last_index - 1].count = 0;
return 1;
}
// Transition function from NFA to DFA
// (generally union of closure operation )
void trans(char S[], int M, char *clsr_t[], int st,
                        char *NFT[][symbols + 1], char TB[]) {
int len = strlen(S);
int i, j, k, g;
int arr[st];
int sz;
reset(arr,
st);
char temp[MAX_LEN], temp2[MAX_LEN]; char
*buff;
// Transition function from NFA to
DFA for (i = 0; i < len; i++) {
        j = ((int)(S[i] - 65));
        strcpy(temp, &NFT[j][M]);
```

```
            if (strcmp(temp, "-") != 0) {
            sz = strlen(temp);
            g = 0;
            while (g < sz) {
                        k = ((int)(temp[g] - 65));
                        strcpy(temp2, &clsr_t[k]);
                        check(arr, temp2);
                        g++;
            }
            }
}
bzero((void *)temp, MAX_LEN);
state(arr, st, temp);
if (temp[0] != '\0') {
            strcpy(TB, temp);
} else
            strcpy(TB, "-");
}
/* Display DFA transition state table*/
void Display_DFA(int last_index, struct DFA *dfa_states,
                                                char *DFA_TABLE[][symbols]) {
int i, j; printf("\n\n*******************************************************\n\n");
printf("\t\t DFA TRANSITION STATE TABLE \t\t \n\n");
printf("\n STATES OF DFA :\t\t");
for (i = 1; i < last_index; i++)
            printf("%s, ", &dfa_states[i].states);
printf("\n");
printf("\n GIVEN SYMBOLS FOR DFA: \t");
for (i = 0; i < symbols; i++)
            printf("%d, ", i);
printf("\n\n");
printf("STATES\t");
for (i = 0; i < symbols; i++)
        printf("|%d\t", i);
printf("\n");
// display the DFA transition state
table printf("------------------------------------+
-------------------------------------------------------\n")
; for (i = 0; i < zz; i++) {
        printf("%s\t", &dfa_states[i + 1].states);
        for (j = 0; j < symbols; j++) { printf("|%s
        \t", &DFA_TABLE[i][j]);
        }
        printf("\n");
}
}
// Driver
Code int
main() { int i,
j, states;
char T_buf[MAX_LEN];
// creating an array dfa structures
struct DFA *dfa_states = malloc(MAX_LEN * (sizeof(dfa)));
states = 6, symbols = 2;
printf("\n STATES OF NFA :\t\t");
for (i = 0; i < states; i++)
        printf("%c, ", (char)(65 + i));
printf("\n");
printf("\n GIVEN SYMBOLS FOR NFA: \t");
for (i = 0; i < symbols; i++)
        printf("%d, ", i);
printf("eps");
printf("\n\n");
```

```
char *NFA_TABLE[states][symbols + 1];
// Hard coded input for NFA table
char *DFA_TABLE[MAX_LEN][symbols];
strcpy(&NFA_TABLE[0][0], "FC");
strcpy(&NFA_TABLE[0][1], "-");
strcpy(&NFA_TABLE[0][2], "BF");
strcpy(&NFA_TABLE[1][0], "-");
strcpy(&NFA_TABLE[1][1], "C");
strcpy(&NFA_TABLE[1][2], "-");
strcpy(&NFA_TABLE[2][0], "-");
strcpy(&NFA_TABLE[2][1], "-");
strcpy(&NFA_TABLE[2][2], "D");
strcpy(&NFA_TABLE[3][0], "E");
strcpy(&NFA_TABLE[3][1], "A");
strcpy(&NFA_TABLE[3][2], "-");
strcpy(&NFA_TABLE[4][0], "A");
strcpy(&NFA_TABLE[4][1], "-");
strcpy(&NFA_TABLE[4][2], "BF");
strcpy(&NFA_TABLE[5][0], "-");
strcpy(&NFA_TABLE[5][1], "-");
strcpy(&NFA_TABLE[5][2], "-");
printf("\n NFA STATE TRANSITION TABLE \n\n\n");
printf("STATES\t");
for (i = 0; i < symbols; i++)
        printf("|%d\t", i);
printf("eps\n");
// Displaying the matrix of NFA transition table
printf("--------+-----------------------------------\n");
for (i = 0; i < states; i++) {
        printf("%c\t", (char)(65 + i)); for
        (j = 0; j <= symbols; j++) {
            printf("|%s \t", &NFA_TABLE[i][j]);
            }
            printf("\n");
}
int closure_ar[states];
char *closure_table[states];
Display_closure(states, closure_ar, closure_table, NFA_TABLE, DFA_TABLE);
strcpy(&dfa_states[last_index++].states, "-");
dfa_states[last_index - 1].count =
1; bzero((void *)buffer, MAX_LEN);
strcpy(buffer, &closure_table[0]);
strcpy(&dfa_states[last_index++].states, buffer);
int Sm = 1, ind = 1;
int start_index = 1;
// Filling up the DFA table with transition values
// Till new states can be entered in DFA table
while (ind != -1) {
            dfa_states[start_index].count = 1;
            Sm = 0;
            for (i = 0; i < symbols; i++) {
            trans(buffer, i, closure_table, states, NFA_TABLE, T_buf);
            // storing the new DFA state in
            buffer strcpy(&DFA_TABLE[zz][i],
            T_buf);
            // parameter to control new states
            Sm = Sm + new_states(dfa_states, T_buf);
            }
            ind = indexing(dfa_states);
            if (ind != -1)
            strcpy(buffer, &dfa_states[++start_index].states);
            zz++;
}
```

```
// display the DFA TABLE Display_DFA(last_index,
dfa_states, DFA_TABLE); return 0;
}
```

OUTPUT

```
STATES OF NFA :          A, B, C, D, E, F,
 GIVEN SYMBOLS FOR NFA:          0, 1,
 eps NFA STATE TRANSITION TABLE


STATES      |0   |1   eps
..........+.............................................
A     |FC    |-    |BF
B    |-    |C    |-
C    |-    |-    |D
D    |E    |A    |-
E    |A    |-
     |BF F     |-
     |-    |-
 e-Closure (A) :     ABF
e-Closure (B) :     B
 e-Closure (C) :     CD
 e-Closure (D) :     D
 ꞯ    losure (E) :    BEF
 e-Closure (F) :    F
******************************************************* DFA TRANSITION
        STATE TABLE
 STATES OF DFA :          ABF, CDF, CD, BEF,

GIVEN SYMBOLS FOR DFA:          0, 1,

STATES      |0   |1

-----------+-------------------------------

ABF     |CDF      |CD

CDF     |BEF      |ABF

CD     |BEF     |ABF

BEF     |ABF      |CD
```

PROGRAM 6

Develop a program to minimize any given DFA.

PROGRAM LOGIC:

Given a deterministic finite state machine A = (T, Q, R, q0, F), this program constructs an equivalent reduced deterministic finite state machine A' = (T, Q', R', q'0, F') as follows:

- Remove all unreachable states from Q (using DFS).
- Construct the states Q' as a partition of states of Q by successive refinement. Initially let there be two groups, F and Q - F. Sets are represents are bitsets in C for efficiency.
- For each group G in Q', partition G into subgroups such that two states q, r of G are in the same subgroup if and only if for all t in T, states q and r have transitions on t to states in the same group of Q', or both don't have transitions on t.
- Replace group G with its subgroups and repeat last step until no group is further refined. PROGRAM

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

int **transitionMap; // 2D array which is used to store state transitions. transitionMap[i][j] is
the state reached when state i is given symbol j
int **partitionTransitionMap; // same as transitionMap, except row indices represent partition
numbers, not state numbers
int startState; // The starting state. This is used as the root for DFS to eliminate unreachable
states long int reachable; // A bitset to represent states that are reachable
long int allStates; // A bitset to represent all states in the FSM
long int finalStates; // A bitset to represent final states in the
FSM
long int nonFinalStates; // A bitset to represent non-final states in the
FSM long int *P; // array of partitions. Each partition is a bitset of states
void dfs(int v)
{
        reachable |= (1 << v);
        // Try exploring all paths..
        for(int i=0; i<26; i++)
                if((transitionMap[v][i] != -1) && ((reachable & (1 << transitionMap[v][i])) ==
0))
                {
                        dfs(transitionMap[v][i]);
                }
}
int main(){
        // We start off with no
        states finalStates = 0;
        allStates = 0;
// Initialize our transition maps. We set transition[i][j] to be -1 in order to indicate that
state/partition i does not transition when given symbol j
        transitionMap = (int**)malloc(64*sizeof(int*)); for
        (int i = 0; i < 64; i++){
                transitionMap[i] = (int*) malloc(26*sizeof(int)); for
                (int j = 0; j < 26; j++){
                        transitionMap[i][j] = -1;
                }
        }
        partitionTransitionMap = (int**)malloc(64*sizeof(int*));
        for (int i = 0; i < 64; i++){
                partitionTransitionMap[i] = (int*) malloc(26*sizeof(int)); for
                (int j = 0; j < 26; j++){
                        partitionTransitionMap[i][j] = -1;
                }
        }
        // read start
        state char
        buff[125];
```

```
fgets(buff, sizeof(buff), stdin);
char *p = strtok(buff, " ");
startState = atoi(p);
// read final states
fgets(buff, sizeof(buff), stdin);
p = strtok(buff, " ");
while (p != NULL)
{
            int state = atoi(p);
            finalStates |= 1 <<
            (state); p = strtok(NULL, "
            ");
}
// read
transitions int
from;
char symbol;
int to;
while (fscanf(stdin, "%d %c %d", &from, &symbol, &to) != EOF) {
            transitionMap[from][symbol-'a'] = to; // add transition
            // add from and to states to the allStates
            bitset allStates |= (1 << from);
            allStates |= (1 << to);
}
// initialize reachable bitset to 0 and run dfs to determine reachable states
reachable = 0;
dfs(startState);
// filter unreachable
states allStates &=
reachable; finalStates &=
reachable;
// initialize array of partitions to include empty
bitsets P = (long int*) malloc(64*sizeof(long int));
for (int i = 0; i < 64 ; i++){
```

```
                P[i] = 0; // no partition exists
        }
        // P should include two partitions to start: final states and non-final
        states nonFinalStates = allStates & ~finalStates;
        P[0] = finalStates;
        P[1] = nonFinalStates;
        int nextPartitionIndex = 2; // Store how many partitions have been added already
// There will be at most 64 partitions. At each iteration, we operate on a partition and add at
most 1 more partition
        for (int i = 0; i < 64; i++){
// A bitset for a new partition. This partition will include all states that are distinct from the
state corresponding to the leftmost bit in P[i]
                long int newPartition = 0;
                // Done
                partitioning if (P[i]
                == 0){
                        break;
                }

// Try to find leftmost bit in the bitset. This loop will only run to its entirety once when that bit
is found
                for (int j = 63; j >= 0; j--) {
                        // Potential leftmost bit. If found, this bit will remain in the bit
                        set. long int staticState = (long int) 1 << j;
                        // Check if this state is in the current
                        bitset if ((P[i] & (staticState)) != 0){
// The lestmost bit state will be associated with this partition. Therefore, we must copy over the
transitions for this state to the transitions for
                                // the corresponding partition
                                partitionTransitionMap[i] = transitionMap[j];
                                // Check for states that should be removed from this partition. All
states will be bits right of the staticState bit
```

```
                    for (int k = j - 1; k >= 0; k -- ){
                        // Potential state to remove
                        long int otherState = (long int) 1 << k;
                        // Check if this state is in the current
                        bitset if ((P[i] & (otherState)) != 0){
                            // Iterate across the entire alphabet and check if
staticState and otherState can transition to different partitions.
                            for (int l = 0; l < 26; l++){
                                int staticNext = -1; // next partition for
                                static int otherNext = -1; // next partition
                                for other for (int m = 0; m <
                                nextPartitionIndex;
m++){
                                    if((P[m]&(1<<transitionMap[j][l])) != 0){
                                    staticNext = m;         //found static
next
                                    }
                                if ((P[m] & (1 << transitionMap[k][l])) !=
0){
                                    otherNext = m; // found other next
                                    }
                                }

// If partitions differ, remove the other state and add it to the new partition. Then break, since
we are done with this partition
        if (transitionMap[j][l] != transitionMap[k][l] && (staticNext != otherNext)){
                                    P[i] &= ~(1 << k);
                                    newPartition |= (1 << k);
                                    break;
                                }
                            }
                        }
                        }
                        break;
                    }
                }
                // New partition exists. Add it to P and increment
                nextPartitionIndex if (newPartition != 0){
                    P[nextPartitionIndex] = newPartition;
                    nextPartitionIndex++;
                }
            }
            // find and print start
            partition int startPartition =
            0;
            for (int i = 0; i < nextPartitionIndex; i ++){
                if ((P[i] & (1 << startState)) != 0 ){
                    startPartition =
                    i; break;
                }
            }
            printf("%d \n", startPartition);
            // find and print final partitions
            for (int i = 0; i < nextPartitionIndex; i++){
                if ((P[i] & finalStates) != 0){
                    printf("%d ", i);
                }
            }
            printf("\n");
            // find and print all transitions
            for (int i = 0; i < nextPartitionIndex; i++){
                for (int j = 0; j < 26; j++) {
                    if (partitionTransitionMap[i][j] != -1){
                        for (int k = 0; k < nextPartitionIndex; k++){
```

```
                                        if ((P[k] & (1 << partitionTransitionMap[i][j])) != 0){
                                                printf("%d %c %d\n", i, j + 'a', k);
                                        }
                                }
                        }
                }
        }
        return 0;
}
```

OUTPUT

```
3
0
1 b 0
2 a 1
3 a 1
3 b 2
```

PROGRAM 7

Develop an operator precedence parser for a given language.

PROGRAM LOGIC:

An operator precedence parser is a one of the bottom-up parser that interprets an operator-precedence grammar. This parser is only used for operator grammars. Ambiguous grammars are not allowed in case of any parser except operator precedence parser. The operator precedence parsers usually do not store the precedence table with the relations; rather they are implemented in a special way. Operator precedence parsers use precedence functions that map terminal symbols to integers, and so the precedence relations between the symbols are implemented by numerical comparison. The parsing table can be encoded by two precedence functions f and g that map terminal symbols to integers. We select f and g such that:

- f(a) < g(b) whenever a is precedence to b
- f(a) = g(b) whenever a and b having precedence
- f(a) > g(b) whenever a takes precedence over b

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void main(){

/*OPERATOR PRECEDENCE PARSER*/
char stack[20],ip[20],opt[10][10][1],ter[10];
int i,j,k,n,top=0,col,row;
clrscr();
for(i=0;i<10;i++)
{
 stack[i]=NULL;
 ip[i]=NULL;
 for(j=0;j<10;j++)
 { opt[i][j][1]=NULL;
 }
}
printf("Enter the no.of terminals
:\n"); scanf("%d",&n);
```

```
printf("\nEnter the terminals
:\n"); scanf("%s",&ter);
printf("\nEnter the table values
:\n"); for(i=0;i<n;i++)
{
 for(j=0;j<n;j++)
 {
 printf("Enter the value for %c %c:",ter[i],ter[j]);
 scanf("%s",opt[i][j]);
 }
}
printf("\n**** OPERATOR PRECEDENCE TABLE ****\n");
for(i=0;i<n;i++)
{
printf("\t%c",ter[i]);
}
printf("\n");
for(i=0;i<n;i++){printf("\n%c",ter[i]);
for(j=0;j<n;j++){printf("\t%c",opt[i][j][0]);}}
stack[top]='$';
printf("\nEnter the input
string:"); scanf("%s",ip);
i=0;
printf("\nSTACK\t\t\tINPUT
STRING\t\t\tACTION\n");
printf("\n%s\t\t\t%s\t\t\t",stack,ip);
while(i<=strlen(ip))
{
for(k=0;k<n;k++)
{
if(stack[top]==ter[k])
col=k; if(ip[i]==ter[k])
row=k;
}
if((stack[top]=='$')&&(ip[i]=='$')){
printf("String is accepted\n");
break;}
else if((opt[col][row][0]=='<') ||(opt[col][row][0]=='='))
{ stack[++top]=opt[col][row][0];
stack[++top]=ip[i];
```

```
 printf("Shift %c",ip[i]);
 i++;
  }
else{
 if(opt[col][row][0]=='>')
{
while(stack[top]!='<'){--top;}
top=top-1;
printf("Reduce");
}
else
{
printf("\nString is not accepted");
break;
}
}
printf("\n");
for(k=0;k<=top;k++)
{
printf("%c",stack[k]);
}
printf("\t\t\t");
for(k=i;k<strlen(ip);k++){
printf("%c",ip[k]);
}
printf("\t\t\t");
}
getch();
}
```

OUTPUT:

```
Enter the value for * *:>

Enter the value for * $:>

Enter the value for $ i:<

Enter the value for $ +:<

Enter the value for $ *:<

Enter the value for $

$:accept
```

```
**** OPERATOR PRECEDENCE TABLE ****

      i     +     *     $

i     e     >     >     >

+     <     >     <     >

*     <     >     >     >

$     <     <     <     a

*/

Enter the input string:

i*i

STACK              INPUT STRING              ACTION

$                  i*i                Shift i

$<i                  *i                Reduce

$                  *i                 Shift *

$<*                  i                 Shift i

$<*<i

String is not accepted
```

PROGRAM 8

To find First and Follow of any given grammar.

PROGRAM LOGIC:

The functions follow and followfirst are both involved in the calculation of the Follow Set of a given Non-Terminal. The follow set of the start symbol will always contain “$”. Now the calculation of Follow falls under three broad cases :

A. If a Non-Terminal on the R.H.S. of any production is followed immediately by a Terminal then it can immediately be included in the Follow set of that Non-Terminal.
B. If a Non-Terminal on the R.H.S. of any production is followed immediately by a Non-Terminal, then the First Set of that new Non-Terminal gets included on the follow set of our original Non-Terminal. In case encountered an epsilon i.e. ” # ” then, move on to the next symbol in the production.
   Note : “#” is never included in the Follow set of any Non-Terminal.
C. If reached the end of a production while calculating follow, then the Follow set of that non-teminal will include the Follow set of the Non-Terminal on the L.H.S. of that production. This can easily be implemented by recursion.

PROGRAM

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

// Functions to calculate
Follow void followfirst(char,
int, int); void follow(char c);

// Function to calculate
First void findfirst(char, int,
int);
```

```
int count, n = 0;

// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production
rules char
production[10][10]; char
f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
        int jm = 0;
        int km = 0;
        int i, choice;
        char c, ch;
        count = 8;

        // The Input grammar
        strcpy(production[0], "E=TR");
        strcpy(production[1], "R=+TR");
        strcpy(production[2], "R=#");
        strcpy(production[3], "T=FY");
        strcpy(production[4], "Y=*FY");
        strcpy(production[5], "Y=#");
        strcpy(production[6], "F=(E)");
        strcpy(production[7], "F=i");

        int kay;
        char
        done[count]; int
        ptr = -1;

        // Initializing the calc_first
        array for(k = 0; k < count; k++)
        {
                for(kay = 0; kay < 100; kay++) {
                        calc_first[k][kay] = '!';
                }
        }
        int point1 = 0, point2, xxx;

        for(k = 0; k < count; k++)
        {
                c = production[k][0];
                point2 = 0;
                xxx = 0;

                // Checking if First of c has
                // already been calculated
                for(kay = 0; kay <= ptr; kay++)
                        if(c == done[kay])
                                xxx = 1;

                if (xxx == 1)
                        continue;
```

```
// Function call
findfirst(c, 0, 0);
ptr += 1;

// Adding c to the calculated
list done[ptr] = c;
printf("\n First(%c) = { ", c);
calc_first[point1][point2++] = c;

// Printing the First Sets of the
grammar for(i = 0 + jm; i < n; i++) {
        int lark = 0, chk = 0;

        for(lark = 0; lark < point2; lark++) {

                if (first[i] == calc_first[point1][lark])
                {
                        chk = 1;
                        break;
                }
        }
        if(chk == 0)
        {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
        }
}
printf("}\n");
jm = n;
point1++;
```

```
}
printf("\n");
printf("
...................................................................................\n\n")
; char donee[count];
ptr = -1;

// Initializing the calc_follow
array for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
                calc_follow[k][kay] = '!';
        }
}
point1 = 0;
int land =
0;
for(e = 0; e < count; e++)
{
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

        // Checking if Follow of ck
        // has alredy been calculated
        for(kay = 0; kay <= ptr; kay++)
                if(ck == donee[kay])
                        xxx = 1;

        if (xxx == 1)
                continue
        ;     land += 1;

        // Function call
```

```
                follow(ck);
                ptr += 1;

                // Adding ck to the calculated
                list donee[ptr] = ck;
                printf(" Follow(%c) = { ", ck);
                calc_follow[point1][point2++] = ck;

                // Printing the Follow Sets of the
                grammar for(i = 0 + km; i < m; i++) {
                        int lark = 0, chk = 0;
                        for(lark = 0; lark < point2; lark++)
                        {
                                if (f[i] == calc_follow[point1][lark])
                                {
                                        chk = 1;
                                        break;
                                }
                        }
                        if(chk == 0)
                        {
                                printf("%c, ", f[i]);
                                calc_follow[point1][point2++] = f[i];
                        }
                }
                printf(" }\n\n");
                km = m;
                point1++;
        }
}
```

```
void follow(char c)
{
        int i, j;

        // Adding "$" to the follow
        // set of the start symbol
        if(production[0][0] == c) {
                f[m++] = '$';
        }
        for(i = 0; i < 10; i++)
        {
                for(j = 2;j < 10; j++)
                {
                        if(production[i][j] == c)
                        {
                                if(production[i][j+1] != '\0')
                                {
                                        // Calculate the first of the next
                                        // Non-Terminal in the production
                                        followfirst(production[i][j+1], i, (j+2));
                                }

                                if(production[i][j+1]=='\0' && c!=production[i][0])
                                {
                                        // Calculate the follow of the Non-Terminal
                                        // in the L.H.S. of the
                                        production
                                        follow(production[i][0]);
                                }
                        }
                }
        }
```

```
}

void findfirst(char c, int q1, int q2)
{
        int j;

        // The case where we
        // encounter a
        Terminal
        if(!(isupper(c))) {
                first[n++] = c;
        }
        for(j = 0; j < count; j++)
        {
                if(production[j][0] == c)
                {
                        if(production[j][2] == '#')
                        {
                                if(production[q1][q2] == '\0')
                                        first[n++] = '#';
                                else if(production[q1][q2] != '\0'
                                                && (q1 != 0 || q2 != 0))
                                {
                                        // Recursion to calculate First of New
                                        // Non-Terminal we encounter after epsilon
                                        findfirst(production[q1][q2], q1, (q2+1));
                                }
                                else
                                        first[n++] = '#';
                        }
                        else if(!isupper(production[j][2]))
                        {
```

```
                                        first[n++] = production[j][2];
                              }
                              else
                              {
                                        // Recursion to calculate First of
                                        // New Non-Terminal we encounter
                                        // at the beginning
                                        findfirst(production[j][2], j, 3);
                              }
                    }
          }
}

void followfirst(char c, int c1, int c2)
{
          int k;

          // The case where we encounter
          // a Terminal
          if(!(isupper(c)))
                    f[m++] = c;
          else
          {
                    int i = 0, j = 1;
                    for(i = 0; i < count; i++)
                    {
                              if(calc_first[i][0] == c)
                                        break;
                    }

                    //Including the First set of the
                    // Non-Terminal in the Follow of
                    // the original query
                    while(calc_first[i][j] != '!')
                    {
                              if(calc_first[i][j] != '#')
                              {
                                        f[m++] = calc_first[i][j];
                              }
                              else
                              {
                                        if(production[c1][c2] == '\0')
                                        {
                                                  // Case where we reach the
                                                  // end of a production
                                                  follow(production[c1][0]);
                                        }
                                        else
                                        {
                                                  // Recursion to the next symbol
                                                  // in case we encounter a "#"
                                                  followfirst(production[c1][c2], c1, c2+1);
                              }         }
                              j++;
                    }
          }
}
```

OUTPUT

```
First(E)= { (, i, }
 First(R)= { +, #, }
 First(T)= { (, i, }

 First(Y)= { *, #, }

 First(F)= { (, i, }


 -------------------------------


 Follow(E) = { $, ),  }

 Follow(R) = { $, ),  }

 Follow(T) = { +, $, ),  }

 Follow(Y) = { +, $, ),  }

 Follow(F) = { *, +, $, ), }
```

PROGRAM 9

Construct a recursive descent parser for an expression.

PROGRAM LOGIC:

A recursive descent parser is a top-down parser. This is one of the most simple forms of parsing. It is used to build a parse tree from top to bottom and reads the input from left to right. A form of recursive descent parsing that does not require backtracking algorithm is known as a predictive parser. The parsers that use backtracking may require exponential time. This parser is normally used for compiler designing purpose. The parser gets an input and reads it from left to right and checks it. If the source code fails to parse properly, then the parser exits by giving an error (flag) message. If it parses the source code properly then it exits without giving an error message.\

PROGRAM

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

void  Tprime();
void  Eprime();
void E();
void  check();
void T();



char expression[10];
int count, flag;

int main()
{
    count = 0;
    flag = 0;
    printf("\nEnter an Algebraic
    Expression:\t"); scanf("%s", expression);
    E();
    if((strlen(expression) == count) && (flag == 0))
    {
        printf("\nThe Expression %s is Valid\n", expression);
    }
    else
    {
        printf("\nThe Expression %s is Invalid\n", expression);
    }
}

void E()
{
    T();
    Eprime();
}

void T()
{
    check();
    Tprime();
}

void Tprime()
{
    if(expression[count] == '*')
    {
        count++;
        check();
        Tprime();
    }
}
```

```
void check()
{
    if(isalnum(expression[count]))
    {
        count++;
    }
    else if(expression[count] == '(')
    {
        count++;
        E();
        if(expression[count] == ')')
        {
            count++;
        }
        else
        {
            flag = 1;
        }
    }
    else
    {
        flag = 1;
    }
}

void Eprime()
{
    if(expression[count] == '+')
    {
        count++;
```

```
            T();
            Eprime();
        }
}
```

OUTPUT

```
Enter an algebraic expression: (a+b)*c
The expression (a+b)*c is Valid
```

PROGRAM 10

Construct a Shift Reduce Parser for a given language.

PROGRAM LOGIC:

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser.This parser requires some data structures i.e.

- A input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Basic Operations –

A. **Shift:** This involves moving of symbols from input buffer onto the stack.

B. **Reduce**: If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.

C. **Accept:** If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.

D. **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<string.h
>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
void main()
  {
    clrscr();
    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
```

```
       puts("enter input string ");
       gets(a);
       c=strlen(a);
       strcpy(act,"SHIFT->");
       puts("stack \t input \t
       action");
       for(k=0,i=0; j<c; k++,i++,j++)
        {
          if(a[j]=='i' && a[j+1]=='d')
            {
              stk[i]=a[j];
              stk[i+1]=a[j+1];
              stk[i+2]='\0';
              a[j]=' ';
              a[j+1]=' ';
              printf("\n$%s\t%s$\t%sid",stk,a,ac
              t); check();
            }
          else
            {
              stk[i]=a[j];
              stk[i+1]='\0';
              a[j]=' ';
              printf("\n$%s\t%s$\t%ssymbols",stk,a,ac
              t); check();
            }
        }
        getch();
   }
 void check()
   {
     strcpy(ac,"REDUCE TO E");
```

```
for(z=0; z<c; z++)
  if(stk[z]=='i' && stk[z+1]=='d')
    {
      stk[z]='E';
      stk[z+1]='\0';
      printf("\n$%s\t%s$\t%s",stk,a,a
      c); j++;
    }
for(z=0; z<c; z++)
  if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
    {
      stk[z]='E';
      stk[z+1]='\0';
      stk[z+2]='\0';
      printf("\n$%s\t%s$\t%s",stk,a,a
      c); i=i-2;
    }
for(z=0; z<c; z++)
  if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
    {
      stk[z]='E';
      stk[z+1]='\0';
      stk[z+1]='\0';
      printf("\n$%s\t%s$\t%s",stk,a,a
      c); i=i-2;
    }
for(z=0; z<c; z++)
  if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
    {
      stk[z]='E';
      stk[z+1]='\0';
```

```
          stk[z+1]='\0';
          printf("\n$%s\t%s$\t%s",stk,a,a
          c); i=i-2;
        }
  }
```

OUTPUT

```
GRAMMAR IS E=E*E

E=E*E

E=(E)

E=id

input string is

(id*id)+id

stack               input                   action

$(                  id*id)+id$              SHIFT->symbols

$(id                *id)+id$                SHIFT->id

$(E                 *id)+id$                REDUCE TO E

$(E*                id)+id$                 SHIFT->symbols

$(E*id               )+id$                  SHIFT->id

$(E*E                  )+id$                      REDUCE TO E

$(E                     )+id$                     REDUCE TO E

$(E)                    +id$                       SHIFT->symbols

$E                      +id$                      REDUCE TO E

$E+                      id$                       SHIFT->symbols

$E+id                      $                      SHIFT->id

$E+E                       $                      REDUCE TO E

$E                         $                      REDUCE TO E
```

PROGRAM 11

Write a program to perform loop unrolling.

```
#include<stdio.h>
void main()
{
unsigned int
n; int x;
char ch;
printf("\nEnter
N\n");
scanf("%u",&n);
printf("\n1. Loop Roll\n2. Loop UnRoll\n");
printf("\nEnter ur choice\n");
scanf(" %c",&ch);
switch(ch)
{
case '1':
  x=countbit1(n);
  printf("\nLoop Roll: Count of  1's     : %d" ,x);
  break;
case '2':
  x=countbit2(n);
  printf("\nLoop UnRoll: Count of 1's : %d" ,x);
  break;
default:
  printf("\n Wrong Choice\n");

}
}
int countbit1(unsigned int n)
{
            int bits = 0,i=0;
            while (n != 0)
            {
 if (n & 1) bits++;
 n >>= 1;
 i++;
            }
            printf("\n no of iterations %d",i);
            return bits;
}
int countbit2(unsigned int n)
{
            int bits = 0,i=0;
            while (n != 0)
            {
 if (n & 1) bits++;
 if (n & 2) bits++;
 if (n & 4) bits++;
 if (n & 8) bits++;
 n >>= 4;
 i++;
            }
            printf("\n no of iterations %d",i);
            return bits;
}
```

OUTPUT

```
1.

2.

Enter ur choice :

1 Enter N:  48

No.of Intersections: 6
```

PROGRAM 12

Write a program to perform constant propagation.

```
#include<stdio.h>
#include<string.h
>
#include<ctype.h>
void input();
void output();
void change(int p,char *res);
void constant();
struct expr
{
char op[2],op1[5],op2[5],res[5];
int flag;
}arr[10];
int n;
void main()
{
input();
constant();
output();
}
void input()
{
int i;
printf("\n\nEnter the maximum number of expressions : ");
scanf("%d",&n);
printf("\nEnter the input :
\n"); for(i=0;i<n;i++)
{
scanf("%s",arr[i].op);
scanf("%s",arr[i].op1);
scanf("%s",arr[i].op2);
scanf("%s",arr[i].res);
arr[i].flag=0;
}
}
void constant()
{
int i;
int op1,op2,res;
char op,res1[5];
for(i=0;i<n;i++)
{
if(isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]) || strcmp(arr[i].op,"=")==0) /*if both digits,
store them in variables*/
```

```
{
op1=atoi(arr[i].op1);
op2=atoi(arr[i].op2);
op=arr[i].op[0];
switch(op)
{
case '+':
res=op1+op2;
break;
case '-':
res=op1-op2;
break;
case '*':
res=op1*op2;
break;
case '/':
res=op1/op2
; break;
case '=':
res=op1;
break;
}
sprintf(res1,"%d",res);
arr[i].flag=1; /*eliminate expr and replace any operand below that uses result of this expr
*/ change(i,res1);
}
}
}
void output()
{
int i=0;
printf("\nOptimized code is : ");
for(i=0;i<n;i++)
{
if(!arr[i].flag)
{
printf("\n%s %s %s %s",arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);
}
}
}
void change(int p,char *res)
{
int i;
for(i=p+1;i<n;i++)
{
if(strcmp(arr[p].res,arr[i].op1)==0)
```

```
strcpy(arr[i].op1,res);
else if(strcmp(arr[p].res,arr[i].op2)==0)
strcpy(arr[i].op2,res);
}
}
```

INPUT

```
Enter the input :
= 3 - a
+ a b t1
+ a c t2
+ t1 t2 t3
```

OUTPUT:

```
Optimized code is :
+ 3 b t1
+ 3 c t2
+ t1 t2 t3
```

PROGRAM 13

Implement Intermediate code generation for simple expressions.

```
#include"stdio.h"
#include"string.h"
int i=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void   explore();
void    fleft(int);
void  fright(int);
struct exp
{
 int pos;
 char op;
}k[15];
void main()
{
 printf("\t\tINTERMEDIATE CODE GENERATION\n\n");
 printf("Enter the Expression :");
 scanf("%s",str);
 printf("The intermediate
 code:\t\tExpression\n"); findopr();
 explore();

}
void findopr()
{
 for(i=0;str[i]!='\0';i++)
  if(str[i]==':')
  {
  k[j].pos=i;
  k[j++].op=':';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='/')
  {
  k[j].pos=i;
  k[j++].op='/';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='*')
  {
  k[j].pos=i;
  k[j++].op='*';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='+')
  {
  k[j].pos=i;
  k[j++].op='+';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='-')
  {
  k[j].pos=i;
  k[j++].op='-';
  }
}
void explore()
{
 i=1;
 while(k[i].op!='\0')
 {
  fleft(k[i].pos);
  fright(k[i].pos);
  str[k[i].pos]=tmpch--;
  printf("\t%c := %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);
  for(j=0;j <strlen(str);j++)
   if(str[j]!='$')
            printf("%c",str[j]);
  printf("\n");
  i++;
 }
 printf("\t%c:=  %c",str[0],str[k[--i].pos]);
}
void fleft(int x)
{
 int w=0,flag=0;
 x--;
 while(x!= -1 &&str[x]!= '+' &&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-
'&&str[x]!='/'&&str[x]!=':')
 {
  if(str[x]!='$'&& flag==0)
  {
  left[w++]=str[x];
  left[w]='\0';
  str[x]='$';
  flag=1;
  }
  x--;
 }
```

```
}
void fright(int x)
{
 int w=0,flag=0;
 x++;
 while(x!= -1 && str[x]!= '+'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-
'&&str[x]!='/')
 {
  if(str[x]!='$'&& flag==0)
  {
  right[w++]=str[x];
  right[w]='\0';
  str[x]='$';
  flag=1;
  }
  x++;
 }
}
```

INPUT

```
d:=a+b*c
The intermediate code:           Expression
    Z := b*c                    d:=a+Z
    Y := a+Z                     d:=Y
    d := Y
```

PROGRAM 14

Implement the back end of the compiler which takes the three address code and produces the 8086 Assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc.

```
#include<stdio.h>
#include<string.h
> struct code
{
char op[2],arg1[5],arg2[5],result[5];
}in[10];
void main()
{
int n,i;
printf("enter the number of
instructions\n"); scanf("%d",&n);
for(i=0;i<n;i++)
{
scanf("%s%s%s%s",in[i].op,in[i].arg1,in[i].arg2,in[i].result);
}
for(i=0;i<n;i++)
{
if(strcmp(in[i].op,"+")==0)
        {
        printf("\nMOV R0,%s",in[i].arg1);
        printf("\nADD R0,%s",in[i].arg2);
        printf("\nMOV %s,R0",in[i].result);
        }
        if(strcmp(in[i].op,"*")==0)
        {
        printf("\nMOV R0,%s",in[i].arg1);
        printf("\nMUL R0,%s",in[i].arg2);
        printf("\nMOV %s,R0",in[i].result);
        }
        if(strcmp(in[i].op,"-")==0)
        {
        printf("\nMOV R0,%s",in[i].arg1);
        printf("\nSUB R0,%s",in[i].arg2);
        printf("\nMOV %s,R0",in[i].result);
        }
        if(strcmp(in[i].op,"/")==0)
        {
        printf("\nMOV R0,%s",in[i].arg1);
        printf("\nDIV R0,%s",in[i].arg2);
        printf("\nMOV %s,R0",in[i].result);
        }
```

```
            if(strcmp(in[i].op,"=")==0)
            {
            printf("\nMOV R0,%s",in[i].arg1);
            printf("\nMOV %s,R0",in[i].result);
            }
            }
}
```

INPUT

```
+ a b t1
+ c d t2

MOV R0,a
ADD R0,b
MOV t1,R0
MOV R0,c
ADD R0,d
MOV t2,R0
```