INTRODUCTION

Every software has to be translated to its equivalent machine instruction form so that it will be executed by underlying machine. There are many different types of system softwares that help during this translation process. Compiler is one such an important system software that converts high level language programs to its low level language. It is impossible to learn and use machine language in software development for the users. Therefore we use high level computer languages to write programs and then convert such programs and into machine understandable form with the help of mediator software such as compilers, interpreters, assemblers etc. Thus compiler bridges the gap between the user and machine is computer.

To build compiler software we must understand the principles, tools and techniques used in its working. The compiler goes through the following sequence of steps called phases of compiler.

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation

The objective is to understand and implement the principles, techniques and also available tools in compiler construction process.

LEX

Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyser is a program that transforms an input stream into sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in C program. The functions of Lex is as follows:

> Firstly, lexical analyzer creates a program lex.l in Lex language. Then lex compiler runs lex.l program and produces a C program lex.yy.c
> Finally C compiler runs lex.yy.c program and produces an object program a.out.
> a.out is a lexical analyzer that transforms an input stream into sequence of tokens.

A lex program is separated into three sections by %% %% delimeters. The format of lex source is as follows;

1. { definitions }
2. %% %%
3. { rules }
4. %% %%
5. { user subroutines }

Definitions include declaration of constants, variables and regular definitions.
Rules define the statement of form p1 {action}, p2 {action}, ... Pn {action}.
User Subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled seperately.

yywrap() function : A lex library routine that you can redefine is yywrap() which is called whenever the scanner reaches the end of a file. If yywrap() returns 1, the scanner continues with normal wrap up on end of input.

yyin() : It is written a variable of the type FILE * and points to input file. yyin is defined by LEX automatically. If the programmer assigns an input file to yyin in the auxillary function section, then yyin is set to point to that file.

yylex() : It is written by lex reads characters from a FILE* file pointer called yyin. If you don't set yyin, it defaults to standard input. A value of 0 is returned when end of file is reached, else yylex() returns a value indicating what kind of token was found.

YACC :

YACC [Yet Another Compiler . Compiler] is an LALR(1) parser generator . YACC was originally designed for being complemented by LEX.

Input file : YACC input file is divided into three parts ;

```
/* definitions */
...
%% %%
/* rules */
...
%% %%
/* auxillary routines */
...
```

Input file : Definition part :

The definition part includes information about tokens used in syntax definition

```
%% token NUMBER
%% token ID
```

YACC automatically assign numbers for token but it can be overridden by

```
%% token NUMBER 621
```

YACC also recognizes single characters as tokens. Therefore, assigned token numbers should not overlap ASCII codes. The definition part can include C code external to definition of parser and variable declarations within %{ and %}

in first column. It also include specification of starting symbol in the grammar

% start nonterminal

Input file : Rule part :

The rule part contains grammar definition in a modified BNF form. Actions is C code in {} and can be embedded inside.

Input file : Auxillary part :

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

Input file :

If yylex() is not defined in auxilliary routines section, then it should be included:

#include "lex.yy.c"

YACC input file generally finishes with .y

Output files :

The output of YACC is a file named y.tab.c. If it contains the main() definition it must be compiled to be executable.

Otherwise, the code can be an external function definition for the function int yyparse(). If called with -d option in command line, YACC produces as output a header file y.tab.h with all its specific definition. If called -v option yacc produces as output file y.output containing textual description of LALR(1) parsing table used by parser. It is useful for tracking how parser solves conflicts.

```
//PROGRAM

#include<string.h>
#include<ctype.h>
#include<stdio.h>
void main()
{
  FILE *f1;
  char c,str[10];
  int lineno=1,num=0,i=0;
  printf("\nEnter the c program\n");
  f1=fopen("input.txt","w");
  while((c=getchar())!=EOF)
  putc(c,f1);
  fclose(f1);
  f1=fopen("input.txt","r");
  while((c=getc(f1))!=EOF)
  {
    if(isdigit(c))
    {
      num=c-48;
      c=getc(f1);
      while(isdigit(c))
      {
        num=num*10+(c-48);
        c=getc(f1);
      }
      printf("%d is a number \n",num);
      ungetc(c,f1);
    }
    else if(isalpha(c))
    {
      str[i++]=c;
      c=getc(f1);
      while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
      {
        str[i++]=c;
        c=getc(f1);
      }
      str[i++]='\0';
      if(strcmp("for",str)==0||strcmp("while",str)==0||  strcmp("do",str)==0||
strcmp("int",str)==0||strcmp("float",str)==0|
      strcmp("char",str)==0|| strcmp("double",str)==0||strcmp("static",str)==0||
      strcmp("switch",str)==0||strcmp("case",str)==0)
      printf("%s is a keyword\n",str);
      else
      printf("%s is a identifier\n",str);
      ungetc(c,f1);
      i=0;
    }
    else if(c==' '||c=='\t')
    printf("\n");
    else if(c=='\n')
    lineno++;
    else
    printf("%c is a special symbol\n",c);
  }
  printf("Total no. of lines are: %d\n",lineno);
  fclose(f1);
}
```

22 / 7 / 21

EXPERIMENT NO : 1

RECOGNIZE THE TOKENS DEFINED BY GRAMMAR BY LEXICAL ANALYZER

AIM :

A c program to design a lexical analyzer to recognize the tokens defined by given grammar.

ALGORITHM :

1. Start
2. Declare an assay of characters an input file to store the input.
3. Read character from input file and put it into character type of variable, say 'c'.
4. If 'c' is blank then do nothing
5. If 'c' is newline character line = line + 1
6. If 'c' is digit, set tokens val, the value assigned, for a digit and return the 'NUMBER'
7. If 'c' is proper token then assign token value.
8. Print the complete table with;
9. Token entered by user
10. Associated token value
11. Stop.

```
}

/* OUTPUT

Enter the c program
int main()
{
printf("Hey");
}int is a keyword

main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
printf is a identifier
( is a special symbol
" is a special symbol
Hey is a identifier
" is a special symbol
) is a special symbol
; is a special symbol
} is a special symbol
Total no. of lines are: 4 */
```


RESULT:

The C program is executed successfully and output verified.

```
//PROGRAM

#include<string.h>
#include<ctype.h>
#include<stdio.h>
#define epsilon 238

int flag=0,len,pt=0;
char st[25];
void E();
void T();
void F();
void EPRIME();
void TPRIME();
void ADVANCE();

int main()
{
 printf("\nRecursive Descent parser\n");
 printf("enter the string:");
 printf("\nE->TE'\nE'->+TE'/%c\nT->FT'\nT'->*FT/%c\nF->(E)/a\
n",epsilon,epsilon);
 printf("enter the string:");
 scanf("%s",st);
 len=strlen(st);
 E();
 if((flag==0)&&(len==pt))
 printf("\n\nString is accepted\n\n");
 else
 printf("\n\nString is rejected\n\n");
 return 0;
}
void E()
{
 T();
 EPRIME();
}
void T()
{
 F();
 TPRIME();
}
void EPRIME()
{
 if(st[pt]=='+')
 {
  ADVANCE();
  T();
  EPRIME();
 }
}
void TPRIME()
{
 if(st[pt]=='*')
 {
  ADVANCE();
  F();
  TPRIME();
 }
}
void F()
{
 if(st[pt]=='(')
 {
```

29 / 7 / 21

EXPERIMENT NO : 2

RECURSIVE DESCENT PARSER FOR GIVEN GRAMMAR

AIM :

To implement the recursive descent parser for a given grammar / language.

ALGORITHM :

1. Start
2. Read the input string
3. Write procedures for non terminals
4. Verify next token equals to non-terminals if it satisfies match the non terminal.
5. If input string does match print string accepted.
6. Else print string not accepted
7. Stop.

```
 ADVANCE();
 E();
 if(st[pt]==')')
 ADVANCE();
 else
 flag=1;
 }
 else if(st[pt]=='a')
 ADVANCE();
 else
 flag=1;
}
void ADVANCE()
{
 pt++;
}


/*OUTPUT

Recursive Descent parser
enter the string:
E->TE'
E'->+TE'/@
T->FT'
T'->*FT/@
F->(E)/a
enter the string:a*a

String is accepted

Recursive Descent parser
enter the string:
E->TE'
E'->+TE'/@
T->FT'
T'->*FT/@
F->(E)/a
enter the string:a*

String is rejected
*/
```

/ /

RESULT :

Any the program is executed successfully and the output is verified.

```
//PROGRAM


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

char ip_sym[15],stack[15];
int ip_ptr=0,st_ptr=0,len,i;
char temp[2],temp2[2];
char act[15];
void check();

int main()
{
 printf("\n\t\tSHIFT REDUCE PARSER\n");
 printf("\nGRAMMER\n");
 printf("\nE->E+E\nE->E/E");
 printf("\nE->E*E\nE->a/b");
 printf("\nEnter the input symbol:\t");
 gets(ip_sym);
 printf("\n\t stack implementation table");
 printf("\nstack\\tinput symbol\t\taction");
 printf("\n\t\t\t\n");
 printf("\n$\t\t%s$\t\t\t--",ip_sym);
 strcpy(act,"shift");
 temp[0]=ip_sym[ip_ptr];temp[1]='\0';
 strcat(act,temp);len=strlen(ip_sym);
 for(i=0;i<=len-1;i++)
 {
  stack[st_ptr]=ip_sym[ip_ptr];
  stack[st_ptr+1]='\0';ip_sym[ip_ptr]=' ';
  ip_ptr++;
  printf("\n$%s\t\t%s$\t\t\t%s",stack,ip_sym,act);
  strcpy(act,"shift");
  temp[0]=ip_sym[ip_ptr];
  temp[1]='\0';
  strcat(act,temp);
  check();
  st_ptr++;
 }
 st_ptr++;
 check();
}

void check()
{
 int flag=0;temp2[0]=stack[st_ptr];temp2[1]='\0';
 if((!strcmp(temp2,"a"))||(!strcmp(temp2,"b")))
 {
  stack[st_ptr]='E';
  if(!strcmp(temp2,"a"))
  printf("\n$%s\t\t%s$\t\t\tE->a",stack,ip_sym);
  else
  printf("\n$%s\t\t%s$\t\t\tE->b",stack,ip_sym);
  flag=1;
 }
 if((!strcmp(temp2,"+"))||(strcmp(temp2,"*"))||(!strcmp(temp2,"/")))
 {
  flag=1;
 }
 if((!strcmp(stack,"E+E"))||(!strcmp(stack,"E\E"))||(!strcmp(stack,"E*E")))
 {
  strcpy(stack,"E");
```

12/8/21

12

EXPERIMENT NO: 3

SHIFT REDUCE PARSER FOR A GIVEN LANGUAGE

AIM:

A program to construct a shift reduce parser for a given language.

ALGORITHM:

loop forever:

for top-of-stack symbol, S and next input symbol, a case action of T[S, a]

shift x: (x is a STATE number)

push a, then x on top of stack and advance ip to point to next i/p symbol.

reduce y: (y is a PRODUCTION number)

assume that the production is of the form A ⇒ beta

pop 2*|beta| symbols of the stack. At this point the top of the stack should be a state number say S'.

Push A, then go to T[S', A] on the top of the stack

Output the production

A → beta

accept:

return -- a successful parse

default:

error -- the input string is not in language.

```
  st_ptr=0;
  if(!strcmp(stack,"E+E"))
  printf("\n$%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);
  else
  flag=1;
 }
 if((!strcmp(stack,"E+E"))||(!strcmp(stack,"E\E"))||(!strcmp(stack,"E*E")))
 {
  strcpy(stack,"E");
  st_ptr=0;
  if(!strcmp(stack,"E+E"))
  printf("\n$%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);
  else
  if(!strcmp(stack,"E\E"))
  printf("\n$%s\t\t%s$\t\t\tE->E/E",stack,ip_sym);
  else
  if(!strcmp(stack,"E*E"))
  printf("\n$%s\t\t%s$\t\t\tE->E*E",stack,ip_sym);
  else
  printf("\n$%s\t\t%s$\t\t\tE->E*E",stack,ip_sym);
  flag=1;
 }
 if(!strcmp(stack,"E")&&ip_ptr==len)
 {
  printf("\n$%s\t\t%s$\t\t\tACCEPT",stack,ip_sym);
  exit(0);
 }
 if(flag==0)
 {
  printf("\n%s\t\t\t%s\t\reject",stack,ip_sym);
  exit(0);
 }
}

/* OUTPUT


      SHIFT REDUCE PARSER

GRAMMER

E->E+E
E->E/E
E->E*E
E->a/b
Enter the input symbol: aa*

       stack implementation table
stack\tinput symbol          action


$           aa*$             --
$a           a*$             shifta
$E           a*$             E->a
$Ea           *$             shifta
$EE           *$             E->a
$EE*           $             shift*

*/
```

```
//PROGRAM

#include<iostream>
#include<stdio.h>
#include<string.h>
using namespace std;
int ister(char);
void first(char);
void follow(char);
char grammar[20][20]={"E->TX","X->+TX","X->i","T->FY","Y->*FY","Y->i","F->(E)","F-
>i"};
char ter[]={'+','*','(',')','i'};
char nonter[]={'E','X','T','Y','F'};
char FIRST[10],FOLLOW[10];
int fi,fo;
int main()
{
 int i,j;
 cout<<"\n\n\n\t\t\tGRAMMAR\n\t\t\t----\n";
 cout<<"\n\t\t\tE->TX"<<"\n\n\t\t\tX->+TX"<<"\n\n\t\t\tX=>i";
 cout<<"\n\n\t\t\tT->FY"<<"\n\n\t\t\tY->*FY"<<"\n\n\t\t\tY->i";
 cout<<"\n\n\t\t\tF->(E)"<<"\n\n\t\t\tF->i";
 printf("\n\n\t\tFIRST\n\t\t---\n");
 for(i=0;i<5;i++)
 {
  fi=0;
  for(j=0;j<10;j++)
  FIRST[j]='\0';
  printf("\n\tFIRST(%c)={",nonter[i]);
  first(nonter[i]);
  for(j=0;j<strlen(FIRST);j++)
  printf("%c,",FIRST[j]);
  printf("\b}");
 }
 printf("\n\t\tFOLLOW\n\t\t------\n");
 for(i=0;i<5;i++)
 {
  fo=0;
  for(j=0;j<10;j++)
  FOLLOW[j]='\0';
  printf("\n\tFOLLOW(%c)={",nonter[i]);
  follow(nonter[i]);
  for(j=0;j<strlen(FOLLOW);j++)
  printf("%c,",FOLLOW[j]);
  printf("\b}");
 }
 return 0;
}
void first(char x)
{
 int i;
 if(ister(x))
 {
  FIRST[fi]=x;
  fi++;
 }
 else
 for(i=0;i<8;i++)
 if(grammar[i][0]==x)
```

19/8/24

15.

EXPERIMENT NO : 4

FIRST AND FOLLOW OF ANY GIVEN TERMINAL OR NON-TERMINAL

AIM :

A program to compute first and follow of any given terminal or non-terminal.

ALGORITHM :

To compute first (x) for all grammar symbol x, apply the following rules :

1) If the x is a terminal, then FIRST(x) = {x}
2) If $x \to \epsilon$ is a production rule, then add ϵ to FIRST (x)
3) If $x \to y_1, y_2, y_3 \ldots y_n$ is a production
   - FIRST (x) = FIRST (y_1)
   - If FIRST (y_1) contain ϵ then FIRST(x) = {FIRST (y_1) - ϵ} ∪ {FIRST (y_2)}
   - If FIRST (y_i) contains ϵ for all i = 1 ton, then add ϵ to FIRST (x).

To compute follow (a) for nonterminal A, apply the following :

1) FOLLOW (S) = {$}
2) If $A \to pBq$ is a production where p, B, q are any grammar symbols, then everything in first (q) except ϵ is in follow (B)
3) If $A \to pB$ is a production then everything in follow (A) is in follow (B)
4) If $A \to pBq$ is a production and first (q) contain ϵ, then follow (B) contains {FIRST (q) - ϵ} ∪ FOLLOW (A).

```
 if(grammar[i][3]=='i')
 {
  FIRST[fi]='i';
  fi++;
 }
 else
 first(grammar[i][3]);
}

void follow(char x)
{
 int i,j,k,flag;
 if(x=='E')
 {
  FOLLOW[fo]='$';
  fo++;
 }
 for(i=0;i<8;i++)
 for(j=4;grammar[i][j]!='\0';j++)
 if(grammar[i][j]==x)
 if(grammar[i][j+1]=='\0')
 {
  follow(grammar[i][0]);
  return;
 }
 else
 {
  fi=flag=0;
  first(grammar[i][j+1]);
  for(k=0;k<fi;k++)
  if(FIRST[k]!='i')
  {
   FOLLOW[fo]=FIRST[k];
   fo++;
  }
  else
  flag=1;
  if(flag)
  {
   follow(grammar[i][0]);
   return;
  }
 }
}
int ister(char a)
{
 int i;
 for(i=0;i<5;i++)
 if(ter[i]==a)
 return 1;
 return 0;
}

/*OUTPUT
```

GRAMMAR

/ /

```
                ----

                E->TX

                X->+TX

                X=>i

                T->FY

                Y->*FY

          Y->i

                F->(E)

                F->i

           FIRST
           ---

      FIRST(E)={(,i}
      FIRST(X)={+,i}
      FIRST(T)={(,i}
      FIRST(Y)={*,i}
      FIRST(F)={(,i}
           FOLLOW
           ------

      FOLLOW(E)={$,)}
      FOLLOW(X)={$,)}
      FOLLOW(T)={+,$,)}
      FOLLOW(Y)={+,$,)}
*/
```


RESULT :

The program is executed successfully and the output is verified

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

char in[25],post[25],stack[25],st[25][25],three[10][10],u[5];
int i,s,p,r,item,top,b,th=0,tr=0;
int pre(char t)
        {
        int r;
        if((t=='+') || (t=='-'))
        r=1;
        if((t=='*') || (t=='/'))
        r=2;
        if((t=='^'))
        r=3;
        if((t=='(') || (t==')'))
        r=0;
        return(r);
        }
        int main()
        {
                int j,k,p,q,a,b;
                char *buff;
                r=1;
                printf("\t\tOUTPUT\n");
                printf("**********************************************\n");
                printf("Enter the expression: \n");
                scanf("%s",in);
                s=1;
                p=0;
                stack[0]='(';
                for(i=2; in[i]!='\0';i++)
        {
                if((in[i]!='+') && (in[i]!='*') && (in[i]!='/') && (in[i]!='^') && (in[i]!='(') && (in[i]!= ')'))
                {
                        post[p] = in[i];
                        p++;
                }
                else
                {
                if(in[i]=='(')
                        {
                        stack[s]=in[i];
                        s++;
                        }
                else
                        {
                if(in[i] == ')')
                        {
                        while(stack[s-1]!='(')
                        post[p] = stack[s-1];
```

2/9/21

EXPERIMENT NO : 5

INTERMEDIATE CODE GENERATION

AIM :

A program to implement three address codes for a set of tokens.

ALGORITHM :

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y'. If value of y is not already in L then generate the instruction MOV y', L to place a copy of y in L.
3. Generate the instruction Op z', L where z' is used to show the current location of z. If z is in both, then prefer a register to memory location. If x is in L then update its descriptor and remove x from all other description.
4. If current value of y or z have no next uses or not live on next from the block or in register then alter the register descriptor to the indicate that after execution of x:y op z those registers will no longer contain y or z

```
                stack[s-1]=' ';
                s--;
                    p++;
                    }
                    stack[])
    printf("%c=%s",in[0],st[0]);
    }s-1]=' ';
                s--;
                }
      else
                {
    a=pre(in[i]);
    b=pre(stack[s-1]);
    j=p;
    while(a<=b)
    {
    post[j] = stack[s-1];
    stack[s-1]=' ';
    s--;
    j++;
    p=j;
    b=pre(stack[s-1]);
    }
    stack[s]=in[i];
    s++;
    }}}}
    for(i=s-1;i>0;i--)
    if(stack[i]!='(')
    {
    post[p]=stack[i];
    p++;
    }
    printf("\n\nThe postfix notation for the given expression : \n\n ");
    for(p=0;post[p]!='\0';p++)
    printf("%c",post[p]);
    printf("\n\n Three address code \n\n ");
    top=0;
    for(p=0;post[p]!='\0';p++)
    {
    if((post[p]!='+') && (post[p]!='-') && (post[p]!='*') && (post[p]!='/') && (post[p]!=
'^'))

    {
    st[top][0]=post[p];
    st[top][1]='\0';
    top++;
    }
    else
    {
    printf("t%d=%s%c%s\n",tr,st[top-2],post[p],st[top-1]);
    top--;
    top--;
    snprintf(u,sizeof(u),"%d",tr);
    strcpy(st[top],"t");
    strcat(st[top],u);
    tr++;
    top++;
    }}
```

```
printf("%c=%s",in[0],st[0]);
}
```

O/P

Enter the expression

a = b + c * d

The postfix for given expression

bcd * +

Three address code

t0 = c * d

t1 = b + t0

a = t1

RESULT :

The program is executed successfully and the output is verified.

```
#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;
 int main()
  {
    input();
    Constant();
    output();
    return 0;
  }

 void input()
  {
    int i;
    printf("\n\n Enter the maximum number of expression : ");
    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;
   char 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)
         {
           sscanf(arr[i].op1,"%d",&op1);
           sscanf(arr[i].op1,"%d",&op2);
           op=arr[i].op[0];
           switch(op)
            {
                case '+' :
                 res=op1+op2;
                 break;
```

4/9/24

EXPERIMENT NO : 6

CONSTANT PROPAGATION

21

AIM :

A program to perform constant propagation

ALGORITHM :

Worklist = all statements in SSA
while worklist ≠ ϕ
remove some statement S from worklist
If S is $x = phi(c, c \dots c)$ for some constant c
replace S with $v = c$
If S is $x = c$ for some constant c
delete S from program
for each statement T that uses v, substitute
c for x in T
worklist = worklist union {T}

```
                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;
          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);
   }
 }
```

O/P

Enter maximum number of expression : 4

Enter input :

= 3 - a

+ a b t1

+ a c t2

+ t1 t2 t3

optimized code is

+ 3 b t1

+ 3 c t2

+ t1 t2 t3

RESULT :

The program is executed successfully and the output is verified.

9/9/24

```c
#include <stdio.h>
#include <string.h>
int main()
 {
  char icode[10][30],str[20],opr[10];
  int i=0;
  printf("\n Enter the sert of intermediate code (terminated by exit):\n");

do
 {
  scanf("%s",icode[i]);
 }while(strcmp(icode[i++],"exit")!=0);

printf("\n target code generation");
printf("*********************************");
i=0;

do
 {
  strcpy(str,icode[i]);
  switch(str[3])
  {
   case '+':
   strcpy(opr,"ADD");
   break;

   case '-':
   strcpy(opr,"SUB");
   break;

   case '*':
   strcpy(opr,"MUL");
   break;

   case'/':
   strcpy(opr,"DIV");
   break;

 }

printf("\n\tMov %c,R%d",str[2],i);
printf("\n\t %s%c,R%d",opr,str[4],i);
printf("\n\tMov R%d,%c",i,str[0]);
}while(strcmp(icode[++i],"exit")!=0);

  return 0;
}
```

9/9/24

EXPERIMENT No : 7

BACKENED OF A COMPILER

AIM :

To implement backened of a compiler which takes 3 address code and produces 8086 assembly language instruction that can be assembled and run using 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes used.

ALGORITHM :

1. Start
2. Open the source file and store the content as quadruples.
3. Check for operators, in quadruples if it is an arithmetic operator generator it or if assignment generates it, else perform unary minus on register C.
4. Write the generated code into output definition of file in outp.c
5. Print the output
6. Stop.

O/P

Enter set of intermediate code

a = 8 + 9

exit

Target code generation

mov 8, R0

~~add~~ 9, R0

mov R0, a

/ /

RESULT :

Thus program is executed successfully and the output is verified.

prgrm 9

```
#include <stdio.h>
#include <string.h>
char result[20][20],copy[3],states[20][20];

void add_state(char a[3],int i)
{
strcpy(result[i],a);
}

void display(int n)
{
 int k=0;
 printf("epsilon closure of %s = {",copy);
  while(k<n)
  {
     Printf("%s",result[k]);
     k++;
  }
 printf("}nnn");
}

int main()
 {
 FILE*INPUT;
 INPUT=fopen("input.dat","r");
 char state[3];
 int end,i=0,n,k=0;
 char state1[3],input[3],state2[3];
 printf("enter the umber of states");
 scanf("%d",&n);
 printf("enter the states");
 for( k =0; k<3;  k++)
  {
    scanf("%s",states[k]);
  }
 for(k=0 ; k<n ; k++)
  {
    i=0;
    strcpy(state,states[k]);
    strcpy(copy,state);
    add_state(state,i++);
    while(1)
    {
      end = fscanf(INPUT,"%s%s%s",state1,input,state2);
      if(end = EOF)
      {
        break;
      }
      if(strcmp(state,state1)==0)
      {
        if(strcmp(input,"e")==0)
        {
        add_state(state2,i++);
        strcpy(state,state2);
        }
```

23/9/24

26

EXPERIMENT NO : 8

∈ - CLOSURE OF A GIVEN NFA

AIM :

Program to find ∈ closure of all states of any given grammar NFA with ∈ transition

ALGORITHM :

1. Enter number of states and the states given. Get a transition table with ∈ moves
2. Read the current state
3. Check ∈ moves from each state to all other states.
4. Display output states.

```
    }
  }
  display(i);
  rewind(INPUT);
  }
 return 0;
}
```

o/p

Enter number of states - 3
enter the states : q_0 q_1 q_2

ϵ - closure of q_0 - $\{q_0\}$
ϵ - closure of q_1 = $\{q_1\}$
ϵ - closure of q_2 = $\{q_2\}$

RESULT :

Thus the program is executed successfully and output is verified.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
int ninputs;
int dfa[100][2][100] = {0};
int state[10000] = {0};
char ch[10],str[1000];
int go[10000][2] = {0};
int arr[10000] = {0};
int main()
{
int st,fin,in;
int f[10];
int i,j=3,s=0,final=0,flag=0,curr1,curr2,k,l,rel;
int c;
printf("\n Follow the one based indexing \n");
printf("\n Enter the number of states::");
scanf("%d",&st);
printf("Give state number from 0 t %d",st-1);
for(i=0;i<st;i++)
state[(int)(pow(2,i))]=1;
printf("enter the number of final state");
for(i=0;i<fin;i++)
{
scanf("%d",&f[i]);
}
int p,q,r,ref;
printf("\n enter number of rules according to NFA::");
scanf("%d",&rel);
printf("\n\nDefine transition rule as \"intial state input symbolfinal state\"\n");
for(i=0;i<rel;i++)
{
scanf("%d%d%d",&p,&q,&r);
if(q==0)
dfa[p][0][r] = 1;
else
dfa[p][1][r] = 1;
}
printf("\n enter intial state :: ");
scanf("%d",&in);
in = pow(2,in);
i=0;
printf("\n solving according to DFA");
int x = 0;
for(i=0; i<st; i++)
{
for(j=0; j<2; j++)
{
int stf = 0;
for(k=0;k<st;k++)
{
if(dfa[i][j][k] == 1)
stf = stf + pow(2,k);
```

30 / 9 / 24

EXPERIMENT NO : 9

28

CONVERSION OF NFA TO DFA

AIM :

Program to implement conversion of NFA to DFA

ALGORITHM :

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

```
}
go[(int)(pow(2,i))][j] = stf;
printf("%d-%d-->%d \n",(int)(pow(2,i)),j,stf);
if(state[stf] == 0)
arr[x++] = stf;
state[stf] = 1;
}
}
for(i=0; i<x; i++)
{
printf("for %d ------",arr[x]);
for(j=0;j<2;j++)
{
int new = 0;
for(k=0;k<st;k++)
{
if(arr[i] & ( 1 << k))
{
int h = pow(2,k);
if(new == 0)
new = go[h][j];
new = new | (go[h][j]);
}
}
if(state[new] == 0)
{
arr[x++] = new;
state[new] = 1;
}
}
}
printf("\n the total number of distinct states are: \n");
printf("STATE 0 1\n");
for(i =0; i<10000 ; i++)
{
if(state[i] == 1)
{
int y =0;
if(i==0)
printf("q0");
else
for(j = 0; j<st; j++)
{
int x =1 << j;
if(x & i)
{
printf("q%d",j);
y=y+pow(2,j);
}
}
printf(" %d %d \n",go[y][0],go[y][1]);
printf("\n");
}
}
j = 3;
while(j--)
{
```

```
printf("\n enter string");
scanf("%s",str);
l = strlen(str);
curr1 = in;
flag = 0;
printf("\n String takes the folloowing path --->\n");
printf("%d-",curr1);
for(i=0; i<l;i++)
{
curr1 = go[curr1][str[i] - '0'];
printf("%d-",curr1);
}
printf("\n Final state - %d \n",curr1);
for(i=0; i<fin; i++)
{
if(curr1 & (1 << f[i]))
{
flag = 1;
}
if(flag)
printf("\n String accepted");
else
printf("\n String rejected");
}
return 0;
}
}
```

O/P

Follow one based indexing
Enter number of states - 2
Enter no. of final states : 1
Enter final states : 1
enter number of rules according to NFA : 3
Define transition rules as initial state
input symbol final state

0 0 1
0 1 1
1 1 1

Enter initial state : 0
Solve according to DFA 1 → 0 → 2

1 - 1 → 2
2 - 0 → 0
2 → 1 → 2
for 0 . . . .

The total number of distinct states are :

State	0	1
q_0	0	0
q_1	2	2
q_2	0	2

Enter string 101
string takes the following path --> 1 - 2 - 2 -
final state - 2
string accepted.

RESULT :

The program is executed successfully and the output is verified.

```
math[+|-|*|/|^]
eq[=|>|<|]
sp[,|;]
%%
"if"|"else"|"int"|"char"|"double"|"float"
printf("\n%s\t is a keyword",yytext);
[a-z A-Z][a-zA-Z0-9]* printf("\n%s\t is an identifier",yytext);
[0-9] printf("\n%s\t is a constant",yytext);
{sp} printf("\n%s\t is a special character",yytext);
{math} printf("\n%s\t is an Arithmetic Operator",yytext);
{eq} printf("\n%s\t is an Equality Operator",yytext);
%%
int yywrap()
{
  return 1;
}
int main(int argc, char**argv)
{
  yyin=fopen(argv[1],"r");
  yylex();
  fclose(yyin);
  return 0;
}
```

30/9/24

EXPERIMENT NO : 10

IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL

AIM :

To implement lexical analyzer using lex tool.

ALGORITHM :

1. Start the program
2. Lex program consist of three parts.
3. Declaration %. %
4. Translation rules % %
5. Auxilliary procedure.
6. The declaration section includes declaration of variables, main test, constants and regular definitions
7. Translation rule of lex programs are statements of the form
8. P_1 {action}
9. P_2 {action}
   . . .
   . . .
10. P_n {action}
11. Write program in Vi editor and save with .l extension
12. Compile the lex program with lex compiler to produce output file as lex.yy.cc
13. y:. $ lexfilename.l
14. $ gcc lex.yy.c -ll
15. Compile that file with C compiler and verify output.

o/p

hello a + b
hello is identifier
a is identifier
+ is arithmetic operator
b is identifier

RESULT :

The program is executed successfully and output is verified.

```
%{
#include<stdio.h>
#include<string.h>
int wc=0, lc=0,ch=0,sc=0;
%}
%%
\n lc++;
[ ] sc++;
[^\t\n] ch++;
%%
void main()
 {
  printf("enter the program\n");
  yylex();
  printf("\n no of lines = %d ",lc);
  printf("\n no of characters = %d" ,ch);
  printf("\n no of words = %d",lc+sc);
 }
int yywrap()
 {
return 1;
 }
```

O/P

Enter the program:

```
int main()
{
printf("Hello");
}
```

No. of lines = 4

No. of characters = 10

No of words = 25

9 / 10 / 24

EXPERIMENT NO : 11

34

DISPLAY NUMBER OF LINES, WORDS OR CHARACTERS IN INPUT TEXT

AIM :

A lex program to display number of lines, words and characters in input text.

ALGORITHM :

1. Start the program.
2. Lex programs have declarations, translation rule and auxiliary procedure.
3. Do declaration section includes declaration of variables, maintext, consonants and regular definition.
4. Translation rule of the form P_1 {action 1}
5. Take the count of lines, words, characters as given in the translation rule.
6. Compile the lex program and verify the output.

RESULT :

The program is executed successfully and the output is verified.

16/10/24

```
%{
#include<ctype.h>
int i;
%}
%%
[a-z A-z]*
{
for(i=0; i<=yyleng; i++)
 {
  if((yytext[i]=='a') && (yytext[i+1]=='b') && (yytext[i+2]=='c')) {
          yytext[i]='A';
          yytext[i+1]='B';
          yytext[i+2]='C';
    }
 }
printf("%s",yytext);
}
[\t]* return;
. * {ECHO;}
\n {printf("%s",yytext);}
%%
main()
 {
  yylex();
}
int yywrap()
 {
  return 1;
 }
```

o/p

a b c

A B C

9 /10 /24

35

EXPERIMENT NO : 12

SUBSTRING CONVERSION

AIM :

A LEX program to convert a substring to uppercase

ALGORITHM :

1. Scan the input
2. Capture each word and check whether it contains substring abc
3. Convert it to uppercase with c function to upper()

RESULT :

The program is executed successfully and output is verified.

```
%{

int v=0;
int c=0;

%}

%%

[aeiouAEIOU] {v++;}
[a-zA-z] {c++;}
"\n" { printf("number of vowels are: %d\n",v);
v=0;
printf("number of consonants are : %d\n",c);
c=0;
}

%%

int main()
 {
  printf("enter the string of vowels and consonants :");
  yylex();
  return 0;
 }
  int yywrap()
 {
  return 1;
 }
```

O/P

Enter the string of vowels & consonants
Arjun
Number of vowels are: 2
Number of consonants are: 3

16 / 10 / 24

EXPERIMENT NO : 13

FIND NUMBER OF VOWELS AND CONSONANTS

AIM :

A lex program to find out total number of vowels and consonants from given input string

ALGORITHM :

1. Start the program
2. Lex program consists of three parts
3. Declaration %. %.
4. Translation rules %. %.
5. Auxilliary procedure
6. The declaration section includes declaration of variables, consonants main test and regular definitions
7. Translation rule of lex program are statements of the form :
8. P_1 {action 1}
9. P_2 {action 2}
   :
10. P_n {action}
11. Define the vowels & consonants in the translation rule.
12. Compile the lex program as lex . program name
13. Run as cc lex.yy.c
14. Verify the output.

RESULT :

The program is executed successfully and output is verified.

program 15

lex.l

```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z]+ {return ID;}
[0-9]+ {return NUMBER;}

[ \t]+ {;}
\n {return 0;}
. {return yytext[0]; }
%%
int yywrap()
{
return 1;
}
```

yacc.y

```
%{
#include "y.tab.h"
%}
%token ID NUMBER
%left '+''-'
%left '*''/'
%%
stmt:expr;
expr:expr'+'expr|expr'-
'expr|expr'*'expr|expr'/'expr|'('expr')'|ID|NUMBER;
%%
int main()
{
printf("\n Enter an expression : \n");
yyparse();
printf("\n Valid Expression \n");
exit(0);
}
int yyerror()
{
printf("\n Invalid expression \n");
exit(0);
}
```

16 / 10 / 24

EXPERIMENT NO : 14

YACC PROGRAM TO RECOGNIZE VALID ARITHMETIC EXPRESSION

AIM :

A YACC program to recognize valid arithmetic expression that uses operator +, -, *, /

ALGORITHM :

1. Start the program
2. Write code for the parser - 1 in the declaration part
3. Write code for the 'y' parser.
4. Also write code for different arithmetical operations
5. Write additional code to print result of computation
6. Execute and verify it.
7. Stop the program.

o/p

```
lex arithmetic.l
yacc -d arithmetic.y
cc lex.yy.c y.tab.c -w
./a.out
Enter an expression : a+b
valid expression
```


RESULT :

The program is executed successfully and the output is verified.

Any 3
Th 4
03/10/24

lex.l

```
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
yylval=atoi(yytext);
return NUMBER;
}
[ \t];
[ \n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

yacc.y

```
%{
#include <stdio.h>
int flag=0;
%}
%token NUMBER
%left '+''-'
%left '*''/''%'
%left '('')'
%%
ArithematicExpression: E{
printf("\n Result=%d\n", $$);
return 0;
};
E:E'+'E{$$=$1+$3;}|E'-'E{$$=$1-
$3;}|E'*'E{$$=$1*$3;}|E'/'E{$$=$1/$3;}|E'%'E{$$=$1%$3;}|'('E')'{$$=$2;}|N
UMBER{$$=$1;};
%%
void main()
{
printf("\n Enter any arithematic expression: \n");
yyparse();
if(flag==0)
printf("\n Entered arithematic expression is valid \n\n");
}
void yyerror()
{
printf("\n Entered arithematic expression is invalid \n\n");
flag=1;
}
```

22 / 10 / 24

EXPERIMENT NO : 15

CALCULATOR USING LEX AND YACC

AIM :

To implement a calculator using lex and YACC.

ALGORITHM :

1. Start the Program
2. Write code for the passer.l in declaration part
3. Write code for the 'y' passer.
4. Write code for different arithmetical operation.
5. Write additional code to print result of computation
6. Execute and verify it.
7. Stop the program.

o/p

Enter any arithmetic expression

3 + 5

Result : 8

entered arithmetic expression is valid

RESULT :

The program executed successfully and the output is verified.

```
program 17
lex.l
%{
#include "y.tab.h"
%}
%%
[a-zA-Z][a-zA-Z0-9]* return ALPHA;
[0-9] return DIGIT;
. return yytext[0];
\n return 0;
%%
int yywrap()
{
return 1;
}
yacc.y
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token DIGIT ALPHA
%%
var: ALPHA
|var ALPHA
|var DIGIT;
%%
int main(int argc,char*argv[])
{
printf("enter the variable name:");
yyparse();
printf("valid Variable! \n");
return 0;
}
int yyerror()
{
printf("invalid Variable! \n");
}
```

22 / 10 / 24

EXPERIMENT NO : 16

41

RECOGNIZE A VALID VARIABLE WHICH START WITH A LETTER

AIM :

To recognize a valid variable which start with a letter followed by any number of letters or digits.

ALGORITHM :

1. Start the program
2. Write code for parser.l in declaration part.
3. Write code for the 'y' parser
4. Also write code for validating input variable name.
5. Write additional code to print result of computation
6. Execute and verify it
7. Stop the program.

o/p

Enter a variable name: Arjun
valid variable !
Enter a variable name: 1 Arjun
Invalid variable !

RESULT :

The program is executed successfully and the output is verified.

1/11/24