MACHINE LEARNING AND COMPILER DESIGN LAB ASSIGNMENT

SUBMITTED BY -

NAME :- ACHAL RAJESH MATE
ROLL NO :- 2203541
ENROLL NO :-MITU20BTCSD001
CLASS :- TY CSE IS- 3

ASSIGNMENT I

Problem Statement

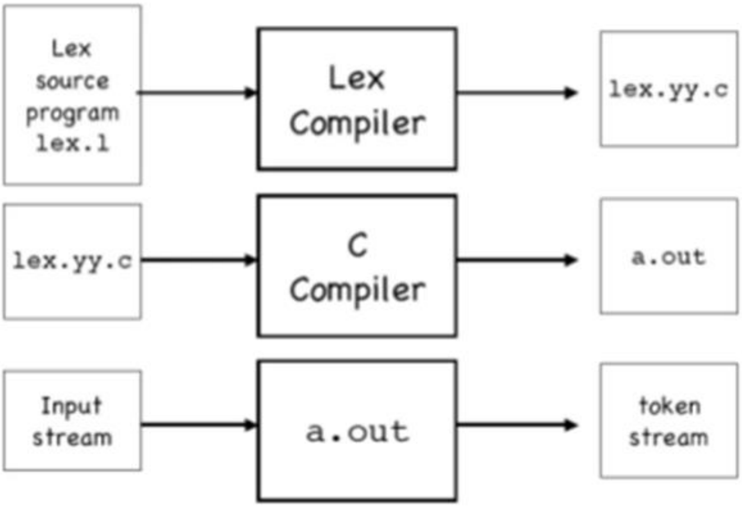
Assignment to understand basic syntax of LEX Specifications, built-in functions and Variables.

Objective

To understand the LEX syntax.

Theory

LEX format:



Specification:

```
%{
        C declarations and includes
%}

declarations
%%
translation rules
%%
user subroutines
```

1. Write a program to find out whether given input is a letter or digit.

```
%{


%}
letter [a-zA-Z]
digit [0-9]
id2 {letter}({letter}|{digit})*
num {digit}("."({digit})+)?

%%
"if"|"else"|"while"|"for"  {printf("keyword");}
{num} {printf("num");}
{id2}    { printf("id2 "); }
%%

int main()
{
yylex();
return 0;
}

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

Output :

Flex letter_or_digit.l

Gcc lex.yy.c

a.exe

Achal Mate

Id2 id2

2203541

Nnnnnnn

MITADTUD001

2. Write a program to find out whether given input is a noun, pronoun, verb, adverb, adjective or preposition

```
%{

%}

%%
" "  |
"\t" ;

go?                    |
runs?                  |
likes?         |
am?                    |
eats?          {printf("%s: is a verb\n",yytext); }
[a-zA-Z]+  {printf("%s: is not a  verb\n",yytext); }
"\n"  ;
%%

int main()
{
yylex();
return 0;
}


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

Output :

Flex verb_not.l

Gcc lex.yy.c

a.exe

I eats food and then likes to run for digestion

I: is not a verb

Eats : is a verb

Foods: is not a verb

And: is not a verb

Then : is not a verb:

Likes : is a verb

To: is not a verb

Run: is a verb

For: is not a verb

Digestion :is not a verb

ASSIGNMENT II

Problem Statement

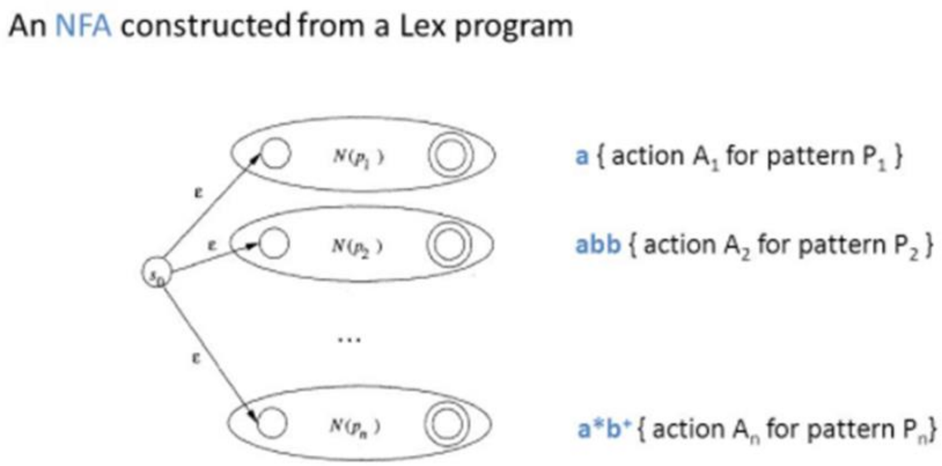
Implement Lexical analyser for sample language using LEX with error handling. (Subset of C).

Objective

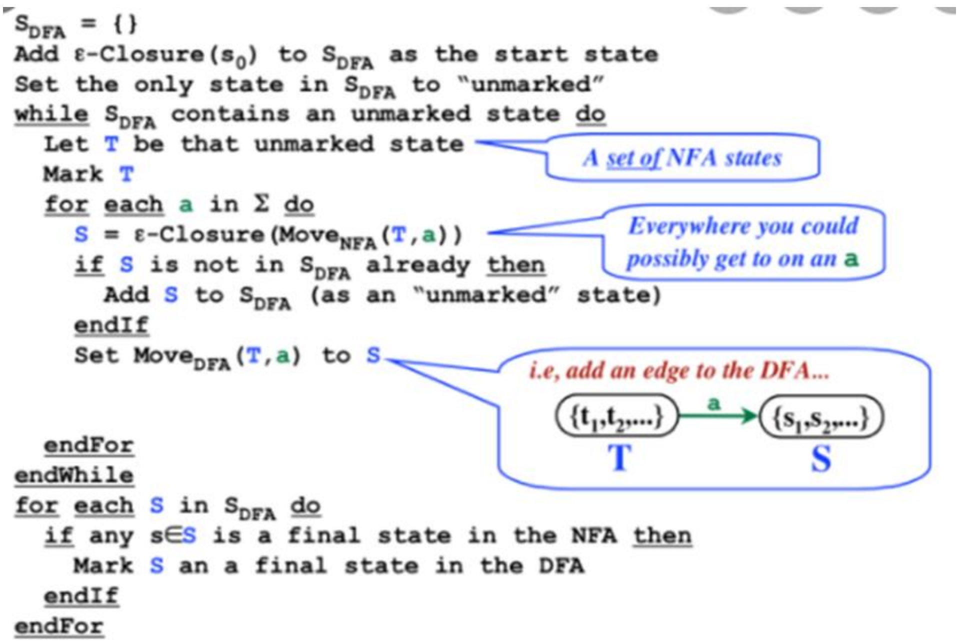
To understand how to build a Lexical Analyser.

Theory

Step 1: Construct ε-NFA from the Regular Expressions.



Convert ε-NFA to DFA using Subset Construction.




```
SDFA = {}
Add ε-Closure(s0) to SDFA as the start state
Set the only state in SDFA to "unmarked"
while SDFA contains an unmarked state do
  Let T be that unmarked state
  Mark T
  for each a in Σ do
    S = ε-Closure(MoveNFA(T,a))
    if S is not in SDFA already then
      Add S to SDFA (as an "unmarked" state)
    endIf
    Set MoveDFA(T,a) to S

  endFor
endWhile
for each S in SDFA do
  if any s∈S is a final state in the NFA then
    Mark S an a final state in the DFA
  endIf
endFor
```

```
%{

%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}

{identifier}\( {printf("\nFUNCTION \n\t%s",yytext);}

{identifier}(\[[0-9]*\])? {printf("\n %s IDENTIFIER",yytext);}
\".*\" {printf("\n\t %s is a STRING",yytext);}
[0-9]+ {printf("\n %s is a NUMBER ",yytext);}
```

```
%%
int main(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}
```

Output :

```
C:\Windows\System32\cmd.exe
C:\Users\Pushkar\OneDrive\Desktop\LEX>a.exe

#include<stdio.h> is a preprocessor directive

#include<conio.h> is a preprocessor directive

        void is a keyword
FUNCTION
        main()
{

        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

 a IDENTIFIER=
 1 is a NUMBER ;

 b IDENTIFIER=
 2 is a NUMBER ;

 c IDENTIFIER=
 a IDENTIFIER+
 b IDENTIFIER;

FUNCTION
        printf(
         "Sum:%d" is a STRING,
 c IDENTIFIER);
}
```

ASSIGNMENT III

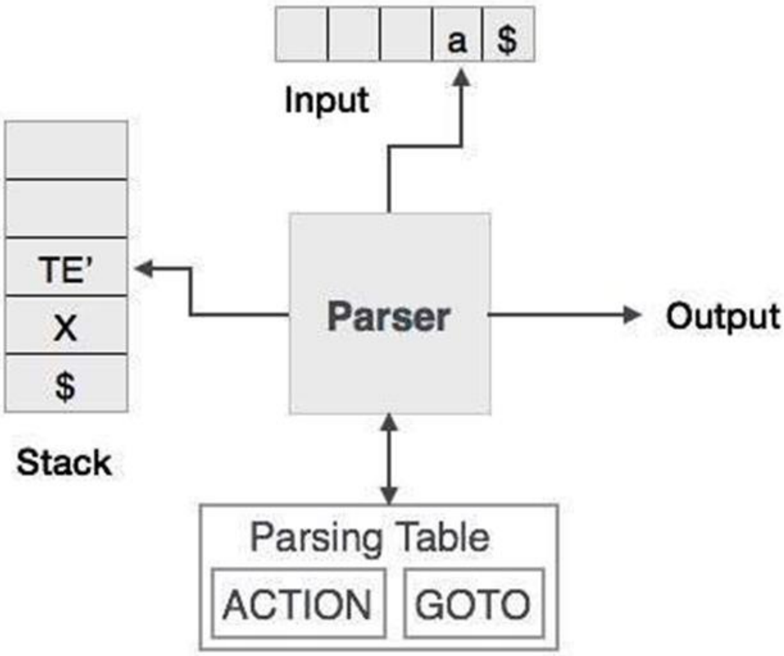
Problem Statement

Implement Recursive Descent Parser for Sample Language

Objective

To understand the working of Predictive Parser. Parse the given string using the Predictive Parser.

Theory:



Solution: Take any CFG as a input and perform the following steps.

Step 1: Remove Left Recursion from the grammar if any.

Step 2: Left Factorize the grammar if required.

Step 3: Construct FIRST set of items for every Non Terminal.

Step 4: Construct Follow set of items for every Non Terminal.

Step 5: Construct Predictive Parsing Table.

Step 6: Parse the given string using the Predictive Parsing Table.

Input: Any CFG as a input

Output: Successful / Unsuccessful Parsing

ASSIGNMENT IV

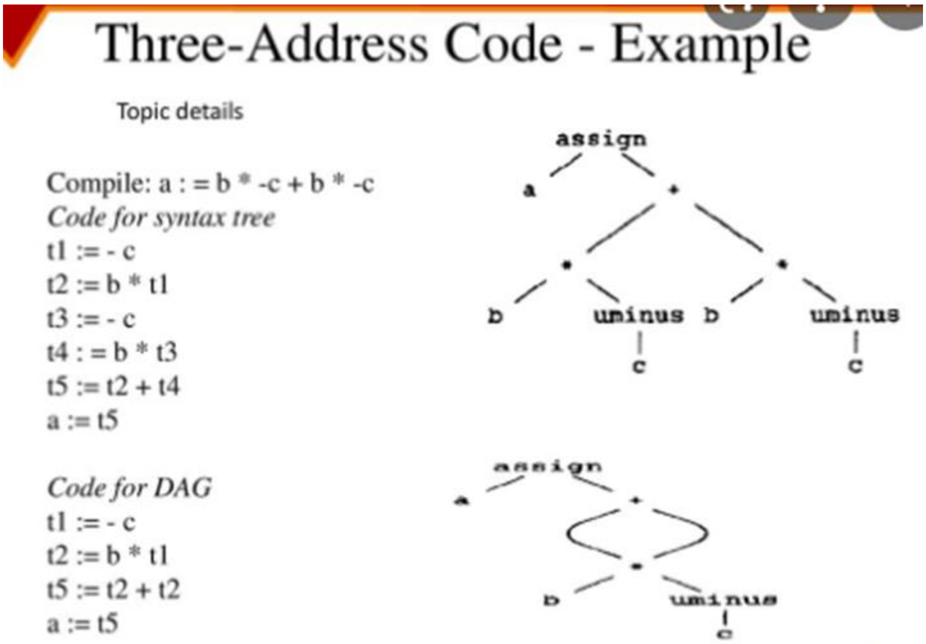
Problem Statement

Implement Intermediate Code generation

Objective

To learn the different forms of Intermediate Code generation such as Three address forms: Quadruples, Triples, Indirect Triples, Syntax Tree, Pseudo Code etc

Theory:



```
#include<stdio.h>
#include<string.h>
void pm();
void plus();
void div();
int i,ch,j,l,addr=100;
char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5];
void main()
{
clrscr();
while(1)
{
printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nEnter the expression with assignment operator:");
scanf("%s",exp);
l=strlen(exp);
exp2[0]='\0';
i=0;
while(exp[i]!='=')
{
i++;
}
strncat(exp2,exp,i);
strrev(exp);
exp1[0]='\0';
strncat(exp1,exp,l-(i+1));
strrev(exp1);
printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
break;

case 2:
printf("\nEnter the expression with arithmetic operator:");
```

```
scanf("%s",ex);
strcpy(exp,ex);
l=strlen(exp);
exp1[0]='\0';

for(i=0;i<l;i++)
{
if(exp[i]=='+'||exp[i]=='-')
{
if(exp[i+2]=='/'||exp[i+2]=='*')
{
pm();
break;
}
else
{
plus();
break;
}
}
else if(exp[i]=='/'||exp[i]=='*')
{
div();
break;
}
}
break;

case 3:
printf("Enter the expression with relational operator");
scanf("%s%s%s",&id1,&op,&id2);
if(((strcmp(op,"<")==0)||(strcmp(op,">")==0)||(strcmp(op,"<=")==0)||(strcmp(o
p,">=")==0)||(strcmp(op,"==")==0)||(strcmp(op,"!=")==0))==0)
printf("Expression is error");
else
{
```

```
printf("\n%d\tif %s%s%s goto %d",addr,id1,op,id2,addr+3);
addr++;
printf("\n%d\t T:=0",addr);
addr++;
printf("\n%d\t goto %d",addr,addr+2);
addr++;
printf("\n%d\t T:=1",addr);
}
break;
case 4:
exit(0);
}
}
}
void pm()
{
strrev(exp);
j=l-i-1;
strncat(exp1,exp,j);
strrev(exp1);
printf("Three address
code:\ntemp=%s\ntemp1=%c%ctemp\n",exp1,exp[j+1],exp[j]);
}
void div()
{
strncat(exp1,exp,i+2);
printf("Three address
code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
void plus()
{
strncat(exp1,exp,i+2);
printf("Three address
code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
```

INPUT:

A<=B

OUTPUT:

100 if a<=b goto 103
101 T:=0
102 goto 104
103 T:=1

ASSIGNMENT V

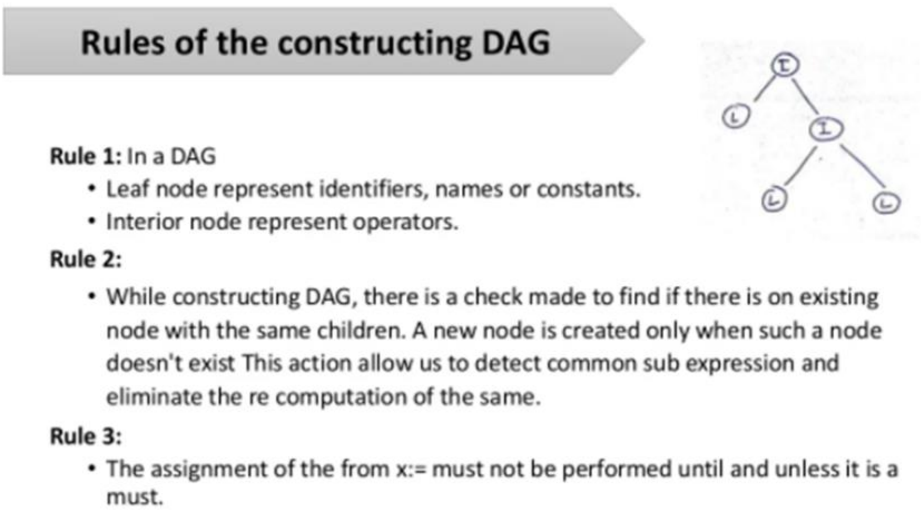
Problem Statement

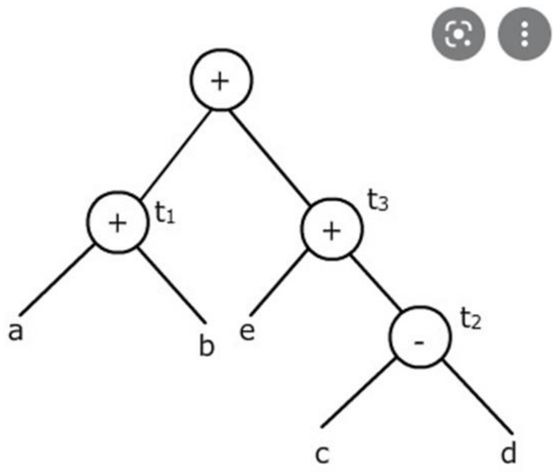
Implement Common Sub expression elimination Code optimization technique using DAG.

Objective

To learn the different techniques of Code Optimization such as Code Motion, Common Subexpression Elimination, Loop Jamming, Loop Unrolling etc. To study about DAG(Directed Acyclic Graph).

Theory :





DAG for *(a + b) + (e + (c - d))*