EX.NO.1 a) STUDY OF LEX TOOL

AIM: To study about lexical analyser generator [LEX TOOL].

1.Introduction: The unix utility lex parses a file of characters. It uses regular expression matching; typically it is used to 'tokenize' the contents of the file. In that context, it is often used together with the yacc utility.

2 Structure of a lex file

A lex file looks like

...definitions...

%%

...rules...

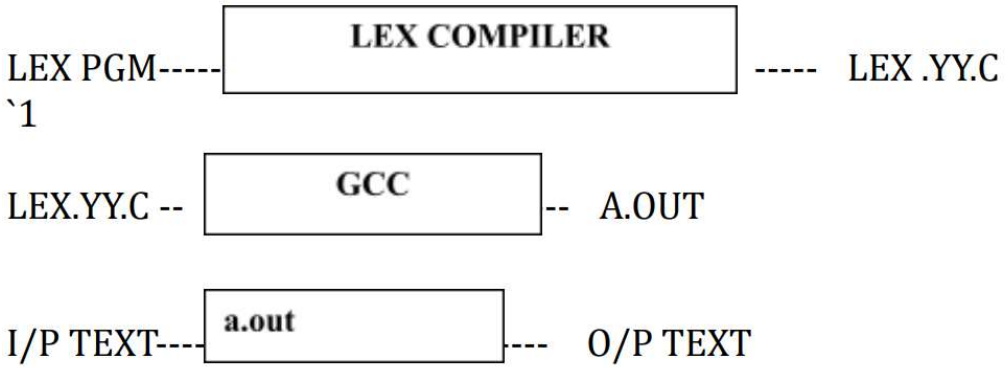
%%

...code...

Here is a simple example:

```
%{
int charcount=0,linecount=0;
%}
%%
. charcount++; \n {linecount++; charcount++;}
%%
int main()
{
yylex();
printf("There were %d characters in %d lines\n",
charcount,linecount);
return 0;
}
```

2.1 Block Diagram:



Definitions All code between %{ and %} is copied to the beginning of the resulting C file.

Rules A number of combinations of pattern and action: if the action is more than a single command it needs to be in braces.

Code This can be very elaborate, but the main ingredient is the call to yylex, the lexical analyser. If the code segment is left out, a default main is used which only calls yylex.

3 Definitions section

There are three things that can go in the definitions section:

C code Any indented code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

Definitions A definition is very much like a #define cpp directive.

For example

```
letter [a-zA-Z]
digit [0-9]
punct [,.:;!?]
nonblank [^ \t]
```

These definitions can be used in the rules section: one could start a rule {letter}+ {...
State definitions If a rule depends on context, it's possible to introduce states and incorporate those in the rules. A state definition looks like %s STATE, and by default a state INITIAL is already given

4. Rules section

The rules section has a number of pattern-action pairs.

4.1 Matched text

When a rule matches part of the input, the matched text is available to the programmer as a variable char* yytext of length int yyleng.

```
%{
int charcount=0,linecount=0,wordcount=0;
%}
letter [^ \t\n]
%% {letter}+ {wordcount++; charcount+=yyleng;}
. charcount++;
\n {linecount++; charcount++;}
```

5 Regular expressions

```
/* Regular expressions */
White       [\t\n ]+
Letter      [A-Za-z]
digit10     [0-9] /* base 10 */
digit16     [0-9A-Fa-f] /* base 16 */
identifier  {letter}(_|{letter}|{digit10})*
int10       {digit10}+
```

The example by itself is, I hope, easy to understand, but let's have a deeper look into regular expressions.

Symbol	Meaning
x	The "x" character
.	Any character except \n
[xyz]	Either x, either y, either z
[^bz]	Any character, EXCEPT b and z
[a-z]	Any character between a and z
[^a-z]	Any character EXCEPT those between a and z
R*	Zero R or more; R can be any regular expression
R+	One R or more
R?	One or zero R (that is an optionnal R)
R{2,5}	Two to 5 R
R{2,}	Two R or more
R{2}	Exactly two R
"[xyz\"foo"	The string "[xyz"foo"
{NOTION}	Expansion of NOTION, that as been defined above in the file
\X	If X is a "a", "b", "f", "n", "r", "t", or "v", this represent the ANSI-C interpretation of \X
\0	ASCII 0 character
\123	ASCII character which ASCII code is 123 IN OCTAL
\x2A	ASCII character which ASCII code is 2A in hexadecimal
RS	R followed by S
R|S	R or S
R/S	R, only if followed by S
^R	R, only at the beginning of a line
R$	R, only at the end of a line
<>	End of file

Conclusion:

With the help of lexical analyser generator tool ,we can separate tokens auttomatically.

EX.NO .1 b) Token Separation using LEX

AIM: To implement Token Separation using LEX TOOL.

Algorithm:

1.Declare the function yy error which is laid and taking char as input.
2.For letter given[a-z A-Z], for digit[0-9] and operation op[- + *].
3.For keyword, it will print it is a keyword.
4.For character as name, it will print it is an identifier.
5.For operators, it will function as intmain()/ print it is an operator.
6.Declare the main function as intmain().
7.In the main function, call the yylex(); function.
8.In the last return(0);
9.Stop the program.

program:

```
%option noyywrap
letter [A-Za-z]
digit [0-9]
operator [+-*]
%%
void |
main |
if |
do |
float |
int |
printf |
char |
for |
while {printf("%s is a keyword\n",yytext);}
%s |
%d |
%c |
%f {printf("%s is a data type\n",yytext);}
{digit}({digit})* {printf("%s is a number\n",yytext);}
{letter}({letter}|{digit})* {printf("%s is an identifier\n",yytext);}
\( {printf("%s is open paranthesis\n",yytext);}
\) {printf("%s is close paranthesis\n",yytext);}
\; {printf("%s is semi colon\n",yytext);}
\. {printf("%s is a dot operator\n",yytext);}
\= {printf("%s is assignment operator\n",yytext);}
\{ {printf("%s is open braces\n",yytext);}
\} {printf("%s is close braces\n",yytext);}
\/ {printf("%s is a back slash\n",yytext);}
\, {printf("%s is a comma\n",yytext);}
\" {printf("%s is double qoute\n",yytext);}
%%
int main(int argc,char* argv[])
{
FILE *fp;
if((fp=fopen(argv[1],"r"))==NULL)
{
printf("FILE doesn't exist");
}
yyin=fp;
```

```
yylex();
return 0;
}
```

Inputfile :

```
Void main()
{
int a=10;
int b=20;
float c=a/b;
print(“%f”,c);
}
```

Output :

Void is a keyword
Main is a keyword
(is a open parenthesis
) is a close parenthesis
{ is a curly bracket
int is a keyword
a= is a equal symbol
10 is a number
; is a double quotes
int is a keyword
b= is a equal symbol
20 is a number
; is a double quotes
float is a keyword
c= is a equal symbol
a/ is slash symbol
(is a open parenthesis
“ is a double quotes
%f is a identifier datatype
, is a comma
} is a close curly bracket

Result: The program of implementation of lexial analyser using LEX hasbeen executed successfully.

EX.NO .2 Evaluation of Arithmetic expression using Ambiguous Grammar(Use Lex and Yacc Tool)

Aim: To create program using LEX and YACC parsing Arithmetic expression using Ambiguous Grammar.

Algorithm:

1.Start the program.
2.Opening the lex compiler and declare the necessary variables.
3.Write the corresponding patterns and actions in rule section and return to YACC.
4.Save the document with “.L” extensions.
5.Open YACC compiler, declare the token and associatively the string of the input operator.
6.In the declaration part, write the production rules as patterns and call YY parser() in the main function to do parsing .compile and build the YACC and output shows in the “execute exe” directly.
7.End the program.

Lex Program:

```
%Option noyywrap
%{
#include<stdio.h>
#include"y.tab.h"
void yyerror(char *);
extern int yylval;
%}
```

```
%%
[0-9]+ {yylval=atoi(yytext); return INT;}
[-*+/\n] {return *yytext;}
[/)/(]{return *yytext;}
. {yyerror("syntax error");}
%%
int yywrap()
{
return 1;
}
```

Ambiguous.yacc

```
%{
#include<stdio.h>
extern int yylex(void);
void yyerror(char *);
%}
%token INT
%%
program:
program expr '\n'  {printf("%d\n",$2);}
|
;
expr:
INT {$$=$1;}
|expr '+' expr {$$=$1+$3;}
|expr '-' expr {$$=$1-$3;}
|expr '*' expr {$$=$1*$3;}
|expr '/' expr {$$=$1/$3;}
|'(' expr ')' {$$=$2;}
%%
void yyerror(cha*s)
{
printf("%s",s);
}
int main()
{
yyparse();
return 0;
}
```

Output :

2+3
5

Result : The program of implementation of Ambiguous using YACC and LEX hasbeen executed successfully.

EX.NO .3 Evaluation of Arithmetic expression using Unmbiguous Grammar(Use Lex and Yacc Tool)

Aim: To create program using LEX and YACC parsing Arithmetic expression using Unmbiguous Grammar.

Algorithm:

1.Start the program.
2.Opening the lex compiler and declare the necessary variables.
3.Write the corresponding patterns and actions in rule section and return to YACC.
4.Save the document with ".L" extensions and compile LEX.
5.Open YACC compiler, declare the token and associatively the string of the input operators.
6.In the declaration part, write the production rules as patterns and call YY parser() in the main function for parsing.

7.Save the document with “.Y” extensions and compile YACC and build the LEX+YACC and output shows on the “execute exe” directly.
8.End the program.

Program:

```
%{
#include<stdio.h>
#include"y.tab.h"
void yyerror(char *);
extern int yylval;
%}
%%
[0-9]+ {yylval=atoi(yytext); return INT;}
[-*+/\n\(\)] {return *yytext;}
. {yyerror("syntax error");}
%%
int yywrap()
{
return 1;
}
```

Unambiguous.yacc

```
%{
#include<stdio.h>
extern int yylex(void);
void yyerror(char *);
%}
%token INT
%%
program:
program expr '\n'  {printf("%d\n",$2);}
|
;
expr:
T {$$=$1;}
|expr '+' T {$$=$1+$3;}
|expr '-' T {$$=$1-$3;}
;
T:
F {$$=$1;}
|T '*' F {$$=$1*$3;}
|T '/' F {$$=$1/$3;}
;
F:
INT {$$=$1;}
|'(' expr ')' {$$=$2;}
%%
void yyerror(char *s)
{
printf("%s",s);
}
int main()
{
yyparse();
return 0;
```

}

Output :

2+5-1

6

Result : The program of implementation of Unambiguous using YACC and LEX hasbeen executed successfully.

EX.NO .4 Use LEX and YACC tool to implement Desktop Calculator.

Aim: To create a desktop calculator using LEX and YACC.

Algorithm:

1.Using the file tool create LEX and YACC files.

2.In the c include section define the header file sequence.

3.Declare the race files inside it in the C definition section declare the header file.

4.Requried along with a variable valid with value assigned as L

5.In the YACC declaration declare the format token Num id.

6.In the grammar rule section if the output string is valid Unambiguous expression then identifier & section the value.

7.In the user define section if the valid is a string as invalid expression YYerror() and define msin function.

Lex File:

```
%option noyywrap
%{
#include<stdio.h>
#include"y.tab.h"
void yyerror(char *s);
extern int yylval;
%}
digit [0-9]
%%
{digit}+     {yylval=atoi(yytext);return NUM;}
[a-z]        {yylval=toascii(*yytext)-97;return ID;}
[A-Z]        {yylval=toascii(*yytext)-65;return ID;}
[-+*/=\n]  {return *yytext;}
\(           {return *yytext;}
\)           {return *yytext;}
.            {yyerror("syntax error");}
%%
```

Calculator.yacc

```
%{
#include<stdio.h>
void yyerror(char*);
extern int yylex(void);
int val[26];
%}
%token NUM ID
%%
S:
S E '\n'            {printf("%d\n",$2);}
| S ID '=' E '\n' {val[$2]=$4;}
|
;
E:
E '+' T {$$=$1+$3;}
|E '-' T {$$=$1-$3;}
|T {$$=$1;}
T:
```

```
T '*' F {$$=$1*$3;}
|T '/' F {$$=$1/$3;}
|F {$$=$1;}
F:
'(' E ')' {$$=$2;}
|NUM {$$=$1;}
|ID {$$=val[$1];}
%%
void yyerror(char *s)
{
printf("%s",s);
}
int main()
{
yyparse();
return 0;
}
```

Output :

```
1+3
4
a=2
b=4
a+b
6
```

Result : The program of implementation of Simple Calculator using YACC and LEX has been executed successfully.

EX.NO.5 Recursive descent parsing

Aim: To implement the Recursive descent parser for the given grammar.

Algorithm:

1.Start the program.
2.Get the expression from the user and call the parser() function.
3.In lexer() get the input symbol and match with the look ahead pointer and then return the token accordingly.
4.In E(), check whether the look ahead pointer is „+‟ or „-„ else return syntax error.
5.In T(),check whether the look ahead pointer is „*‟ or „/‟ else return syntax error.
6.In F(),check whether the look ahead pointer is a member of any identifier.
7.In main(), check if the current look ahead points to the token in a given CFG it doesn‟t match the return syntax error.

Program:

```
#include<stdio.h>
#include<conio.h>
int i=0 ,f=0;
char str[30];
void E();
void Eprime();
void T();
void Tprime();
void F();
void E()
{
printf("\nE->TE'");
T();
Eprime();
}
```

```
void Eprime()
{
if(str[i]=='+')
{
printf("\n E'->+TE'");
i++;
T();
Eprime();
}
else if((str[i]==')') || (str[i]=='$'))
printf("\nE'->^");
}
void T()
{
printf("\nT->FT'");
F();
Tprime();
}
void Tprime()
{
if(str[i]=='*')
{
printf("\nT'->*FT'");
i++;
F();
Tprime();
}
else if((str[i]==')')||(str[i]=='+') || (str[i]=='$'))
printf("\nT'->^");
}
void F()
{
if(str[i]=='a')
{
printf("\nF->a");
i++;
}
else if(str[i]=='(')
{
printf("\nF->(E)");
i++;
E();
if(str[i]==')')
i++;
}
else
f=1;
}
void main()
{
int len;
clrscr();
printf("Enter the string: ");
```

```
scanf("%s",str);
len=strlen(str);
str[len]='$';
E();
If((str[i]=='$')&&(f==0))
printf("\nString sucessfully parsed!");
else
printf("\nSyntax Error!");
getch();
}
```

Output 1

```
Enter the string: a+a*a$
E->TE'
T->FT'
F->a
T'->^
E'->+TE'
T->FT'
F->a
T'->*FT'
F->a
T'->^
E'->^
String sucessfully parsed!
```

Output 2

```
Enter the string: a++
E->TE'
T->FT'
F->a
T'->^
E'->+TE'
T->FT'
T'->^
E'->+TE'
T->FT'
Syntax Error!
```

Result : The program of implementation of Recursive decent parsing hasbeen executed successfully.

EX.NO:6.Shift Reduce Parser

Aim: To write a program dor implementing shift reduce parsing using c.

Algorithm:

1.Start the program
2.read the variables , stack symbols
3. loop forever:for top-of-stack symbol, s, and next input symbol, a case action of T[s,a]
4. shift x: (x is a STATE number) push a, then x on the top of the stack and advance ip to point to next input symbol
5. reduce y: (y is a PRODUCTION number) Assume that the production is of the form A ==> beta
6. 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 goto of T[s',A] (a state number) on the top of the stack.
7. Output the production A ==> beta.
8. accept: return --- a successful parse.
9.default: error --- the input string is not in the language.
10.Stop the program.

program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int z,i,j,c;
char a[16],stk[15];
void reduce();
void main()
{ clrscr();
puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->a");
puts("enter input string ");
gets(a);
c=strlen(a);
a[c]='$';
stk[0]='$';
puts("stack \t input \t action");
for(i=1,j=0;j<c; i++,j++)
{
if(a[j]=='a')
{
stk[i]=a[j];
stk[i+1]='\0';
a[j]=' ';
printf("\n
%s\t%s\tshift a",stk,a);
reduce();}
else
{
stk[i]=a[j];
stk[i+1]='\0';
a[j]=' ';
printf("\n%s\t%s\tshift->%c",stk,a,stk[i]);
reduce();
}
}
if(a[j]=='$')
reduce();
if((stk[1]=='E')&&(stk[2]=='\0'))
printf("\n%s\t%s\tAccept",stk,a);
else
printf("\n%s\t%s\terror",stk,a);
getch();
}
void reduce()
{
for(z=1; z<=c; z++)
if(stk[z]=='a')
{
stk[z]='E';
stk[z+1]='\0';
printf("\n%s\t%s\tReduce by E->a",stk,a);
}
for(z=1; 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\tReduce by E->E+E",stk,a);
i=i-2;
}
for(z=1; 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\tReduce by E->E*E",stk,a);
i=i-2;
}
for(z=1; z<=c; z++)
if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
{
stk[z]='E';
stk[z+1]='\0';
stk[z+2]='\0';
printf("\n%s\t%s\tReduce by E->(E)",stk,a);
i=i-2;
}
}
```

Output:

```
GRAMMAR is E->E+E
E->E*E
E->(E)
E->a
enter input string
a*a+a
stack    input     action
$a       *a+a$    shift a
$E       *a+a$    Reduce by E->a
$E*       a+a$   shift->*
$E*a       +a$   shift a
$E*E      +a$    Reduce by E->a
$E        +a$    Reduce by E->E*E
$E+ a       $     shift->+
$E+a        $      shift a
$E+E        $     Reduce by E->a
$E          $     Reduce by E->E+E
$E          $      Accept
```

Result: The program of implementation of Shift Reduce parsing has been executed successfully.

Ex. No.7. OPERATOR PRECEDENCE PARSING

Aim: Write a c program to implement operator precedence parsing.

Algorithm:

1. Include the nessary header files.
2. Declare the nessary variables with the operators defined before.
3. Get the input from the user and compare the string for any operators.
4. Find the precedence of the operator in the expression from the predefined operator.

5. Set the operator with the maximum precedence accordingly and give the relational operators for them.
6. Parse the given expression with the operators and values.
7. Display the parsed expression.
8. Exit the program.

Algorithm:

```
#include<stdio.h>
#include<string.h>
int main()
{
char stack[20],ip[20],opt[10][10][1],ter[10];
int i,j,k,n,top=0,col,row;
for(i=0;i<5;i++)
{
stack[i]='\0';
ip[i]='\0';
for(j=0;j<5;j++)
{
opt[i][j][0]='\0';
}
}
printf("Enter the no.of terminals:");
scanf("%d",&n);
printf("\nEnter the terminals:");
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("\nOPERATOR 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])
row=k;
if(ip[i]==ter[k])
col=k;
}
if((stack[top]=='$')&&(ip[i]=='$'))
{
printf("String is accepted");
break;
}
else if((opt[row][col][0]=='<') ||(opt[row][col][0]=='='))
{
stack[++top]=opt[row][col][0];
stack[++top]=ip[i];
printf("Shift %c",ip[i]);
i++;
}
else
{
if(opt[row][col][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");
}
return 0;
}
```

OUTPUT:

```
Enter the no.of terminals:3
Enter the terminals:a+$
Enter the table values:
Enter the value for a a:e
Enter the value for a +:>
Enter the value for a $:>
Enter the value for + a:<
Enter the value for + +:>
Enter the value for + $:>
```

```
Enter the value for $ a:<
Enter the value for $ +:<
Enter the value for $ $:a
OPERATOR PRECEDENCE TABLE:
    a       +       $
a   e       >       >
+   <       >       >
$   <       <       a
Enter the input string:a+a$
STACK     INPUT STRING     ACTION
$         a+a$             Shift a
$<a        +a$             Reduce
$           +a$            Shift +
$<+        a$              Shift a
$<+<a       $              Reduce
$<+          $             Reduce
$             $            String is accepted
```

Result: The program of implementation of operator precedence parsing has been executed successfully

EX.NO:08. Implementation of 3-Address Code

Aim: To Implement of 3-Address code Generate by using c program.

Algorithm:

1.Start the program.
2.Open the input file.
3.Enter the intermediate code as an input to the program.
4.Apply conditions for checking the keywords in the intermediate code.
5.Analze each instruction in switch case.
6.After generating machine code, copy it to the output file.
7.Stop the program.

program:

```
#include<stdio.h>
#include<conio.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()
{
clrscr();
printf("\t\tINTERMEDIATE CODE GENERATION\n\n");
printf("Enter the Expression :");
scanf("%s",str);
printf("The intermediate code\n");
findopr();
explore();
getch();
```

```
}
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++;
}
fright(-1);
if(no==0)
{
fleft(strlen(str));
```

```
printf("\t%s = %s",right,left);
getch();
exit(0);
}
printf("\t%s = %c",right,str[k[--i].pos]);
getch();
}
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]!='/')
{
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++;
}
}
```

Output:

INTERMEDIATE CODE GENERATION

Enter the Expression :a=b+c*d/e

The intermediate code

Z = d/e a=b+c*Z

Y = c*Z a=b+Y

X = b+Y a=X

a = X

Result: The program of implementation of 3-Address code has been executed successfully

EX.N0:09.Symbol Table

Aim: To write a program for implementing symbol Table Using C program.

Algorithm:

1.Start the program.
2.Open the input file.
3.Enter the intermediate code as an input to the program.
4.Apply conditions for checking the keywords in the intermediate code.
5.Analze each instruction in switch case.
6.After generating machine code, copy it to the output file.
7.Stop the program.

Program:

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
void main()
{
int i=0,j=0,x=0,n;
void *p,*add[5];
char ch,srch,b[15],d[15],c;
printf("Expression terminated by $:");
while((c=getchar())!='$')
{
 b[i]=c;
 i++;
}
n=i-1;
printf("Given Expression:");
i=0;
while(i<=n)
{
 printf("%c",b[i]);
 i++;
}
printf("\n Symbol Table\n");
printf("\nSymbol \t addr \t type");
while(j<=n)
{
 c=b[j];
 if(isalpha(toascii(c)))
 {
 p=malloc(c);
 add[x]=p;
 d[x]=c;
 printf("\n%c \t %d \t identifier\n",c,p);
 x++;
 j++;
 }
 else
 {
 ch=c;
 if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
 {
 p=malloc(ch);
 add[x]=p;
```

```
d[x]=ch;
printf("\n %c \t %d \t operator\n",ch,p);
x++;
j++;
}}}}
```

Output:

```
Expression terminated by $:A=B+C$
Given Expression:A=B+C
 Symbol Table

Symbol    addr     type
A     -1706927424      identifier

 =    -1706927344      operator

B     -1706927264      identifier

 +    -1706927184      operator

C     -1706927120      identifier
```

EX.NO:10-Construction of NFA and DFA from a regular expression.

AIM:

EX.NO:11- Implementation of simple code optimization techniques.

AIM: To implement of simple code optimization techniques by using c.

Algorithm:

1.Start
2.Intialise value N
3.Initialize the count value
4.If perform loop unrolling then,
 i.Perform each operation upto count.
5.Else, loop rolling
 i.Check the condition
 ii.perfrom the operation
 iii.increment the count
6.Print the result
7.stop

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.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':
      printf("\nLoop Roll: Count of 1's : %d" ,x);
```

```
      break;
   case '2':
      printf("\nLoop UnRoll: Count of 1's : %d" ,x);
      break;
   default:
      printf("\n Wrong Choice\n");
   }
   getch();
}
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:

Result:To implement of simple code optimization techniques is implemented successfully

EX.NO:12- Construct a Simple Code Generator

AIM: To construct back end of a simple code generator by using c program.

Algorithm:

1.Start.
2.Get address code sequence.
3.Determine current location of 3 using address (for 1st operator)
4.If current location not already exist generate move (B,O)
5.Update address of A(for 2nd operand).
6.If current value of B and () is null, exist
7.If they generate operator () A,3 ADPR
8.Store the move instruction in memory
9.Stop

Program: (Back End of the Compiler)

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

```
void main()
  {
     char icode[10][30], str[20], opr[10];
     int i=0;
     printf("\nEnter the set of intermediate code (terminated by exit):\n");
     do{
          scanf("%s", icode[i]);
     }
     while(strcmp(icode[i++],"exit")!=0);
     printf("\nTarget code generation");
     printf("\n********************");
     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);
  }
```

Output:

```
Enter the set of intermediate code (terminated by exit):
a=b+c
b=c-d
c=d*e
exit
Target code generation
********************
     Mov b,R0
     ADD c,R0
     Mov R0,a
     Mov c,R1
     SUB d,R1
     Mov R1,b
     Mov d,R2
     MUL e,R2
     Mov R2,c
```

Result: Construction of a simple code generator is constructed successfully