

**Ex No: 1**

**Date:**

## **DEVELOP A LEXICAL ANALYZER TO RECOGNIZE TOKENS IN C**

**AIM:**

To implement the program to identify C keywords, identifiers, operators, end statements like [ ], { } using C tool.

**ALGORITHM:**

- We identify the basic tokens in c such as keywords, numbers, variables, etc.
- Declare the required header files.
- Get the input from the user as a string and it is passed to a function for processing.
- The functions are written separately for each token and the result is returned in the form of bool either true or false to the main computation function.
- Functions are issymbol() for checking basic symbols such as () etc , isoperator() to check for operators like +, -, \*, / , isidentifier() to check for variables like a,b, iskeyword() to check the 32 keywords like while etc., isInteger() to check for numbers in combinations of 0-9, isnumber() to check for digits and substring().
- Declare a function detecttokens() that is used for string manipulation and iteration then the result is returned from the functions to the main. If it's an invalid identifier error must be printed.
- Declare main function get the input from the user and pass to detecttokens() function.

**PROGRAM:**

```
#include<stdio.h>

#include<ctype.h>

#include<stdlib.h>

#include<string.h>

#include<math.h>

void main()

{#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("Symbol \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++;
    } } }
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("Symbol \t addr \t type");
while(j<=n)
{

```

```

c=b[j];
if(isalpha(toascii(c)))
{
p=malloc(c);
add[x]=p;#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("Symbol \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++;
}}}}
printf("\n %c \t %d \t operator\n",ch,p);
x++;
j++;
}}}}

```

**OUTPUT:**

```
C:\Users\WELCOME\Documents>gcc string.c -o string
```

```
C:\Users\WELCOME\Documents>string.exe
```

```
190701018 5:11:10.1040
```

```
Expression terminated by $:a+b=c$
```

```
Given Expression:a+b=c
```

```
Symbol Table
```

Symbol	addr	type
a	12719088	identifier
+	12719200	operator
b	12719264	identifier
=	12719376	operator
c	12719456	identifier

```
C:\Users\WELCOME\Documents>
```

## RESULT:

Thus, the program was implemented using LEX to identify the keywords, operators, identifiers and symbols like { } , [ ] ; through C Program.

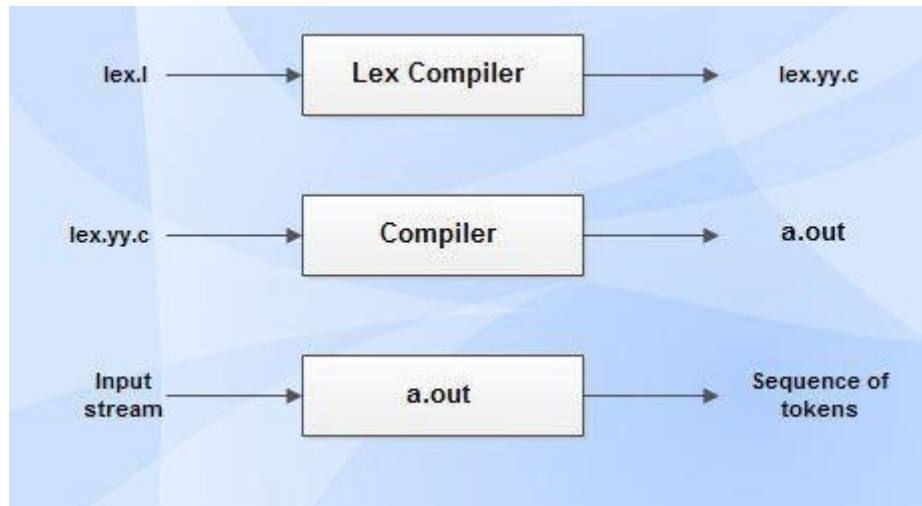
**Ex No: 2**

**Date:**

## **STUDY OF LEX TOOL**

### **LEX:**

Lex is a tool in lexical analysis phase to recognize tokens using regular expression. Lex tool itself is a lex compiler.



- lex.l is an input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
- lex.yy.c is compiled by the C compiler to a file called a.out.
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
- yylval is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.
- The attribute value can be numeric code, pointer to symbol table or nothing.
- Another tool for lexical analyzer generation is Flex.

### **STRUCTURE OF LEX PROGRAMS:**

Lex program will be in following form

declarations

%%

translation rules

%%

auxiliary functions

**Declarations:** This section includes declaration of variables, constants and regular definitions.

**Translation rules:** It contains regular expressions and code segments.

Form : Pattern { Action }

Pattern is a regular expression or regular definition.

Action refers to segments of code.

### Patterns for tokens in the grammar

- *digit* → [0-9]
- digits* → *digit*<sup>+</sup>
- number* → *digits* ( . *digits* )? ( E [ + - ]? *digits* )?
- letter* → [A-Za-z]
- id* → *letter* ( *letter* | *digit* )<sup>\*</sup>
- if* → if
- then* → then
- else* → else
- relop* → < | > | <= | >= | = | <>
- *ws* → (blank | tab | newline)<sup>+</sup>

```
%{ LT, LE, EQ, NE, GT, GE, IF,
    THEN, ELSE, ID, NUMBER, RELOP
}%
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter} | {digit})*
number   {digit}+(\.{digit})?(E[+-]?{digit})+
%%
{ws}     {}
if       {return(IF); }
then     {return(THEN); }
else     {return(ELSE); }
```

**Auxiliary functions:** This section holds additional functions which are used in actions.

These functions are compiled separately and loaded with lexical analyzer. Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found. Once a match is found, the associated action takes place to produce token. The token is then given to parser for further processing.

### CONFLICT RESOLUTION IN LEX:

Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:

- Always prefer a longer prefix than a shorter prefix.



- If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred

**yylex()**

a function implementing the lexical analyzer and returning the token matched

**yytext**

a global pointer variable pointing to the lexeme matched

**yylen**

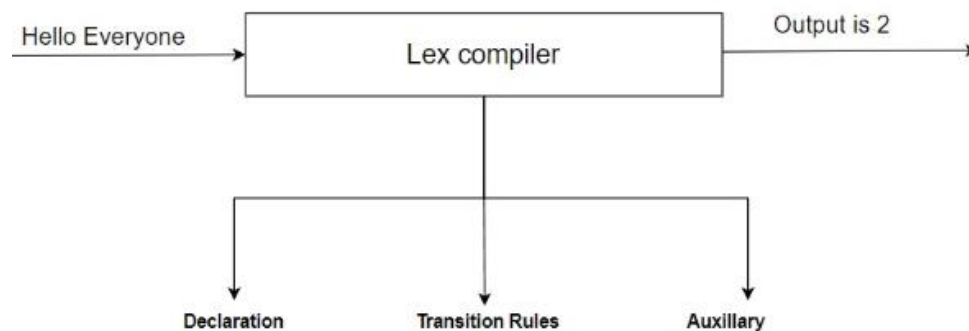
a global variable giving the length of the lexeme matched

**yyval**

an external global variable storing the attribute of the token

**yywrap:**

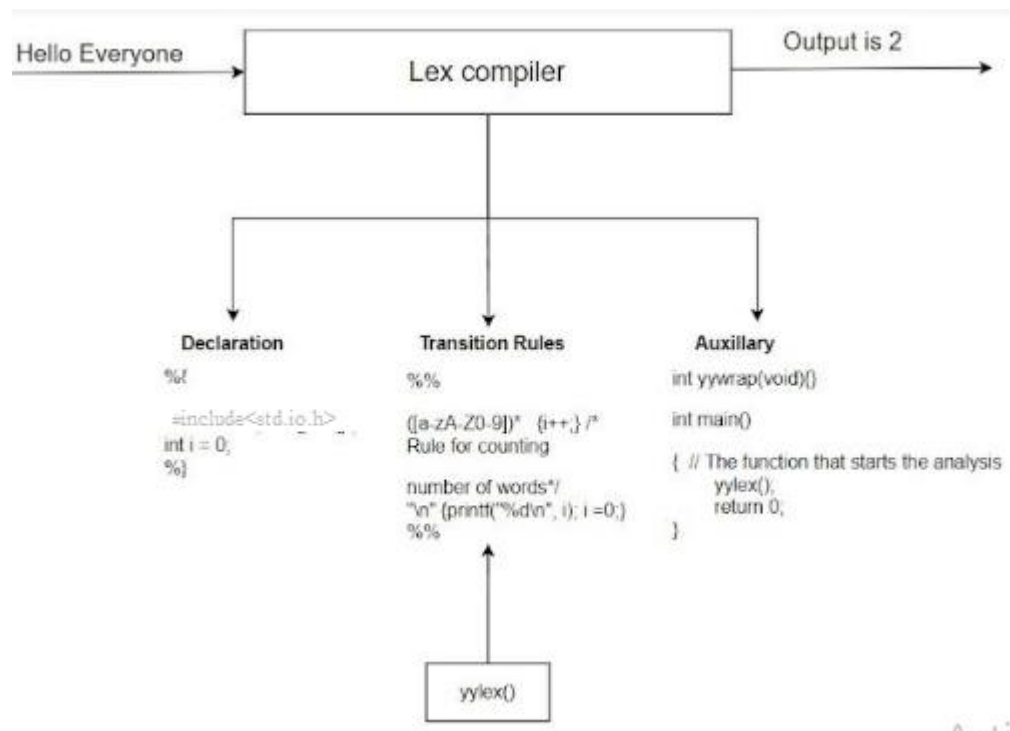
Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required.



Declaration-Has header files and initialization of variables

Translation rules-write the rules for counting the words

Auxillary- has yylex() that calls the translation rules



Simple lex program:

```
/*lex program to count number of words*/
```

```
% {
```

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

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

```
int i = 0;
```

```
% }
```

```
/* Rules Section*/
```

```
%%
```

```
([a-zA-Z0-9])* {i++;} /* Rule for counting
number of words*/
```

```
"\n" {printf("%d\n", i); i = 0;}
```

```
%%
```

```
int yywrap(void){}
```

```
int main()
```

```
{
```

```
// The function that starts the analysis
yylex();
```

```
return 0;
```

```
}
```

```
C:\Users\WELCOME\Documents>file1.exe
190701018 3 190701018 3
0rec hello world
 3 vikram rolex
 2
```

**Result:**

The lex program to find no. of words in a sentence is executed and verified.

**Ex No: 3**

**Date:**

## **DESIGN A DESK CALCULATOR USING LEX**

**AIM:**

**Problem statement:**

Create a calculator that performs addition , subtraction, multiplication and division using lex tool.

**ALGORITHM:**

- In the headers section declare the variables that is used in the program including header files if necessary.
- In the definitions section assign symbols to the function/computations we use along with REGEX expressions.
- In the rules section assign dig() function to the dig variable declared.
- In the definition section increment the values accordingly to the arithmetic functions respectively.
- In the user defined section convert the string into a number using atof() function.
- Define switch case for different computations.
- Define the main () and yywrap() function.

**PROGRAM:**

**Declaration section**

```
% {  
  
    int a,b,flag=0;  
  
% }
```

```
dig [0-9]*  
add "+"  
sub "-"  
mul "*"  
div "/"
```

**Rule section**

```
%%  
  
{ dig } { dig();}  
  
{ add } { flag=1;}
```

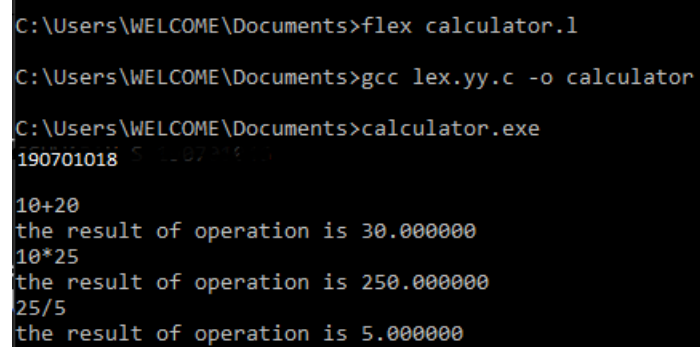
```
{sub} {flag=2;}
{mul} {flag=3;}
{div} {flag=4;}
\n {printf("The answer is:%d\n",a);}
%%
dig()
{
    if(flag==0)
    {
        a=atof(yytext);
    }
    else
    {
        b=atof(yytext);
        switch(flag)
        {
            case 1:
                a=a+b;
                break;
            case 2:
                a=a-b;
                break;
            case 3:
                a=a*b;
                break;
            case 4:
                a=a/b;
                break;
        }
    }
}

int main()
```

```
{    // The function that starts the analysis
    yylex();
    return 0;
}

int yywrap(void) {}
```

### OUTPUT:



```
C:\Users\WELCOME\Documents>flex calculator.l
C:\Users\WELCOME\Documents>gcc lex.yy.c -o calculator
C:\Users\WELCOME\Documents>calculator.exe
190701018 5.000000
10+20
the result of operation is 30.000000
10*25
the result of operation is 250.000000
25/5
the result of operation is 5.000000
```

### RESULT:

Thus, the calculator was implemented using LEX tool.

**Ex No: 4**

**Date:**

## **STUDY OF YACC TOOL**

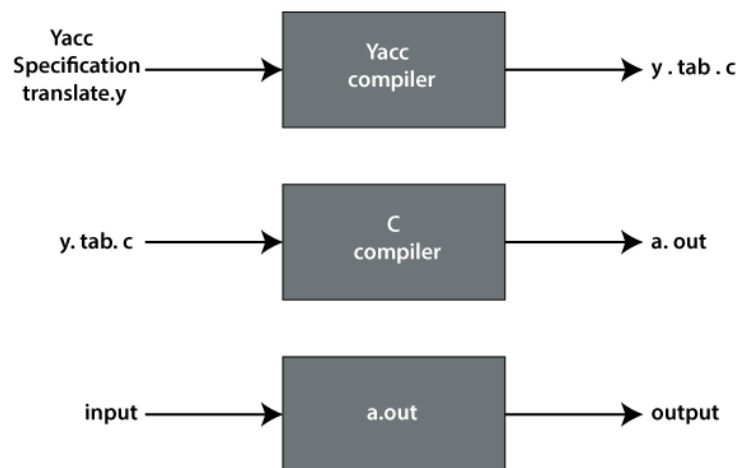
### **YACC TOOL:**

YACC is known as Yet Another Compiler Compiler. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar, and the output is a C program. If we have a file translate.y that consists of YACC specification, then the UNIX system command is:

### **YACC translate.y**

This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser prepared in C with some other user's prepared C routines. By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.

The construction of translation using YACC is illustrated in the figure below:



### **STRUCTURE OF YACC:**

Declarations

%%

Translation rules

%%

Supporting C rules

C declarations	<code>%{\n#include &lt;stdio.h&gt;\n%}</code>
yacc declarations	<code>%token NAME NUMBER</code>
Grammar rules	<code>%%\n\nstatement: NAME '=' expression\n          expression\n        ;\n\nexpression: expression '+' NUMBER { \$\$ = \$1 + \$3; }\n          expression '-' NUMBER { \$\$ = \$1 - \$3; }\n          NUMBER { \$\$ = \$1; }\n        ;\n\n%%</code>
Additional C code	<code>int yyerror(char *s)\n{\n    fprintf(stderr, "%s\\n", s);\n    return 0;\n}\n\nint main(void)\n{\n    yyparse();\n    return 0;\n}</code>

**Declarations Part:** This part of YACC has two sections; both are optional. The first section has ordinary C declarations, which is delimited by `%{` and `%}`. Any temporary variable used by the second and third sections will be kept in this part. Declaration of grammar tokens also comes in the declaration part. This part defined the tokens that can be used in the later parts of a YACC specification.

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

Terminal

Start Symbol

**Translation Rule Part:** After the first `%%` pair in the YACC specification part, we place the translation rules. Every rule has a grammar production and the associated semantic action. A set of productions:

$$\langle \text{head} \rangle \Rightarrow \langle \text{body} \rangle_1 \mid \langle \text{body} \rangle_2 \mid \dots \mid \langle \text{body} \rangle_n$$

would be written in YACC as

```
<head> : <body>1    {<semantic action>1}
      | <body>2    {<semantic action>2}
      | .....
      | <body>n    {<semantic action>n}
      ;
```



Grammar rule section

```
expr  : expr '+' term
      | term
      ;
term   : term '*' factor
      | factor
      ;
factor : '(' expr ')'
      | ID
      | NUM
```

In a YACC production, an unquoted string of letters and digits that are not considered tokens is treated as non-terminals.

The semantic action of YACC is a set of C statements. In a semantic action, the symbol \$\$ is considered to be an attribute value associated with the head's non-terminal. While \$i is considered as the value associated with the grammar production of the body. If we have left only with associated production, the semantic action will be performed. The value of \$\$ is computed in terms of \$i's by semantic action.

**Supporting C-Rules:** It is the last part of the YACC specification and should provide a lexical analyzer named **yylex()**. These produced tokens have the token's name and are associated with its attribute value. Whenever any token like DIGIT is returned, the returned token name should have been declared in the first part of the YACC specification.

The attribute value which is associated with a token will communicate to the parser through a variable called **yylval**. This variable is defined by a YACC.

Whenever YACC reports that there is a conflict in parsing-action, we should have to create and consult the file **y.output** to see why this conflict in the parsing-action has arisen and to see whether the conflict has been resolved smoothly or not.

Instructed YACC can reduce all parsing action conflict with the help of two rules that are mentioned below:

- A reduce/reduce conflict can be removed by choosing the production which has conflict mentioned earlier in the YACC specification.
- A shift/reduce conflict is reduced in favor of shift. A shift/reduce conflict that arises from the dangling-else ambiguity can be solved correctly using this rule.

## OUTPUT

```
C:\Users\eshwaran\Documents>a.exe
190701018 190701018
type the expression
3+2
valid expression
```

## RESULT:

The YAAC program to check validity of arithmetic expression is executed and verified.

**Ex No: 5**

**Date:**

## **Recognize an arithmetic expression using LEX and YACC**

**AIM:**

To check whether the arithmetic expression using lex and yacc.

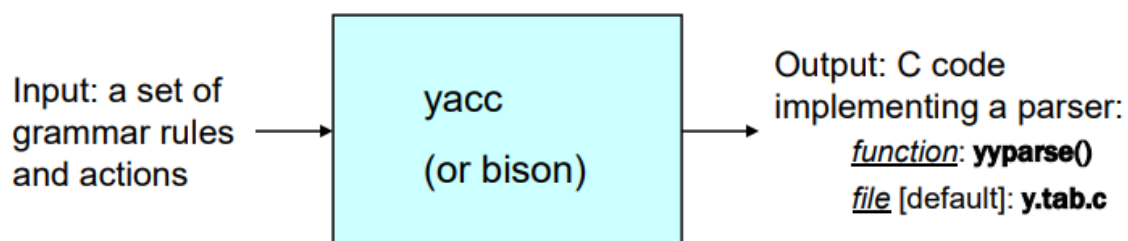
**ALGORITHM:**

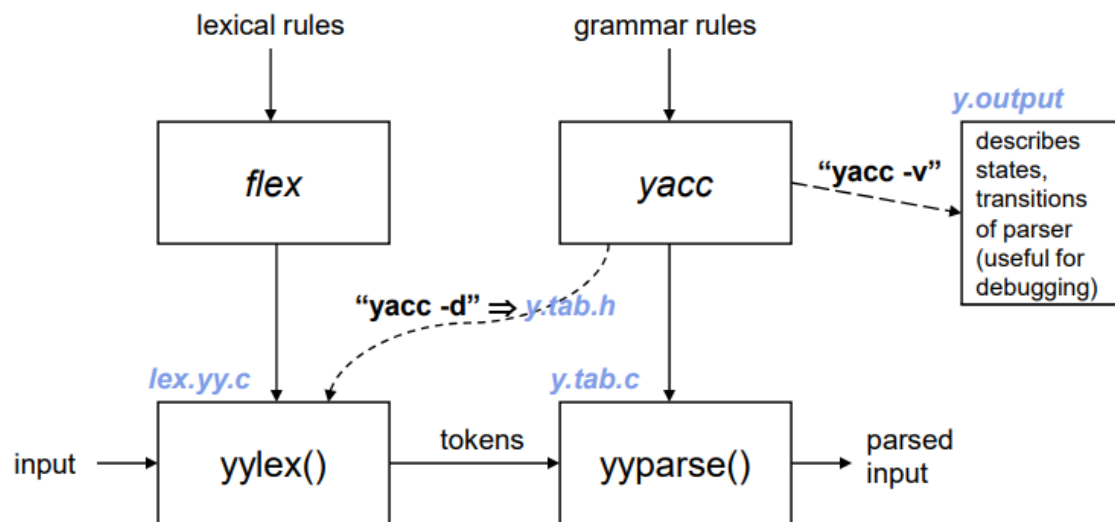
- Using the flex tool, create lex and yacc files.
- In the C include section define the header files required.
- In the rules section define the REGEX expressions along with proper definitions.
- In the user defined section define yywrap() function.
- Declare the yacc file inside it in the C definitions section declare the header files required along with an integer variable valid with value assigned as 1.
- In the Yacc declarations declare the format token num id op.
- In the grammar rules section if the starting string is followed by assigning operator or identifier or number or operator followed by a number or open parenthesis followed by an identifier. The x could be an operator followed by an identifier or operator or no operator then declare that as valid expressions by making the valid stay in 1 itself.
- In the user definition section if the valid is 0 print as Invalid expression in yyerror() and define the main function.

## **LEX AND YACC WORKING**

Parser generator:

- Takes a specification for a context-free grammar.
- Produces code for a parser.





Yacc determines integer representations for tokens:

- Communicated to scanner in file y.tab.h
- use “yacc -d” to produce y.tab.h

Token encodings:

- “end of file” represented by ‘0’;
- A character literal: its ASCII value
- Other tokens: assigned numbers  $\geq 257$ .
- Parser assumes the existence of a function ‘int yylex()’ that implements the scanner.
- Scanner:
  - Return integer value indicates the type of token found
  - Values communicated to the parser using yytext, yylval
  - yytext determines lexeme of a token and yylval determines a integer assigned to a token
  - The token error is reserved for error handling

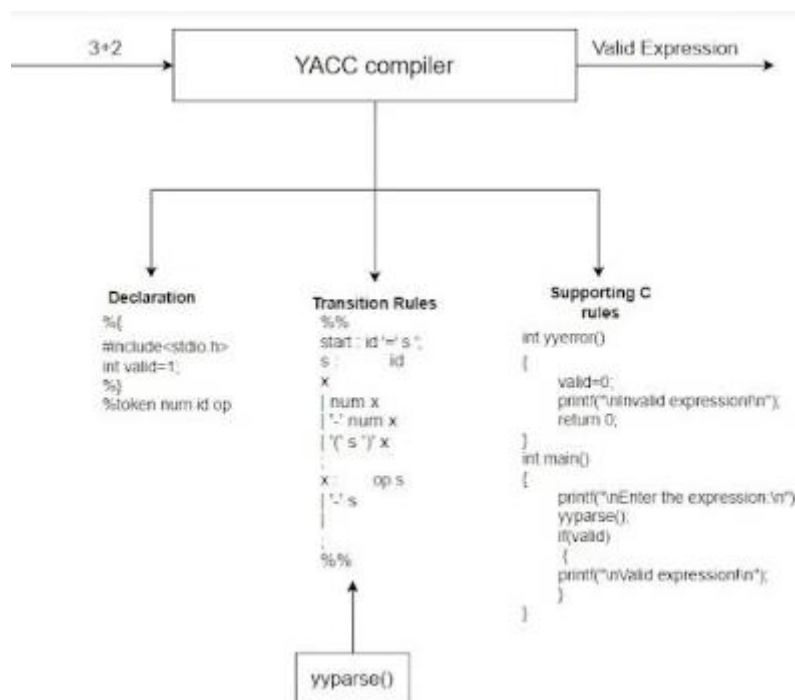
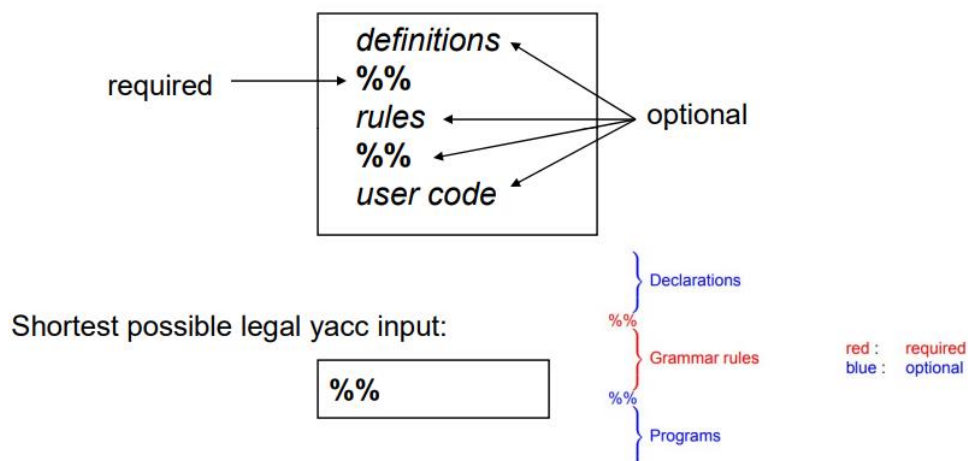
### Communication between Scanner and Parser :-

Suppose the grammar specification is in the file yacc.y then

- The command ‘yacc yacc.y’ would yield a file y.tab.c containing the parser constructed by yacc
- The command ‘yacc -d yacc.y’ constructs a file y.tab.h which is included in the declaration scanner section of lex file

The user needs to supply the main() function to the driver and a file yyerror() is generated by the parser if there is an error in the input.

Yacc Input File :-



## intyyparse()

Called once from main() [user-supplied]

Repeatedly calls yylex() until done:

- On syntax error, calls yyerror() [user-supplied]
- Returns 0 if all of the input was processed
- Returns 1 if aborting due to syntax error.

Example: `intmain() { return yyparse(); }`

## Yacc Grammar Rules :

### Information about tokens:

- Token names:

- Declared using ‘%token’ %token name1 name2 ...
- Any name not declared as a token is assumed to be a nonterminal.
- Start symbol of grammar, using ‘%start’ [optional] %start name
  - If not declared explicitly, defaults to the nonterminal on the LHS of the first grammar rule listed
- Stuff to be copied verbatim into the output (e.g., declarations, #includes): enclosed in % { ... } %

### Rules:

Grammar Production	Yacc Rule
A -> B1 B2...Bm	A: B1 B2 ... Bm;
B -> C1 C2,...Cn	B: C1 C2 ... Cn;
C -> D1 D2,...Dk	C: D1 D2 ... Dk;

- Rule RHS can have arbitrary C code embedded within { ... }.  
E.g. A : B1 { printf(“after B1\n”); x = 0; } B2 { x++; } B3;
- Left-recursion more efficient than right-recursion:–  
A : A x | ... rather than A : x A | ...

### Specifying Operator Properties

Binary operators: %left, %right, %nonassoc:

Associativity of operator

%left '+' '-'

%left '\*' '/'

%right '^'

Operators in the same group have the same precedence

Unary operators: %prec

- Changes the precedence of a rule to be that of the token specified. E.g.:

%left '+' '-'

%left '\*' '/'

Expr: expr '+' expr

| '-' expr %prec '\*'

| ...

## PROGRAM:

### Lex file:

```
% {
#include<stdio.h>
#include "y.tab.h"
extern int yyval;
% }

%%

[0-9]+ {
    yyval=atoi(yytext);
    return NUMBER;
}

[\t] ;
[\n] return 0;
. return yytext[0];
%%

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

### Yacc file:

#### Definition section

```
% {
    #include<stdio.h>
    int flag=0;

% }

%token NUMBER

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%

ArithmeticExpression: E{
    printf("\nResult=%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;}
|NUMBER {$$=$1;}
;
%%
```

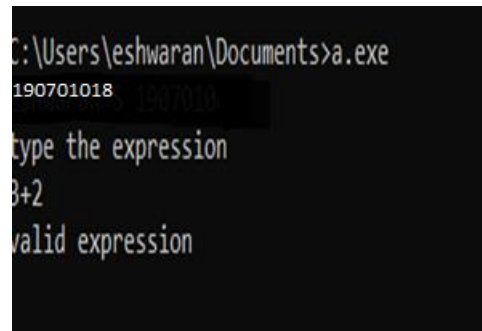
```

void main()
{
    yyparse();
    if(flag==0)
        printf("\nEnter arithmetic expression is Valid\n\n");

}
void yyerror()
{
    printf("\nEnter arithmetic expression is Invalid\n\n");
    flag=1;
}

```

### OUTPUT:



```

C:\Users\eshwaran\Documents>a.exe
190701018 1907010
type the expression
3+2
valid expression

```

### RESULT:

Thus the valid expression using LEX and YACC was verified.



**Ex No: 6**

**Date:**

## **EVALUATE EXPRESSION THAT TAKES DIGITS, \*, + USING YACC**

**AIM:**

To perform arithmetic operations using lex and yacc.

**ALGORITHM:**

- Using the flex tool, create lex and yacc files.
- In the definition section of the lex file, declare the required header files along with an external integer variable yylval.
- In the rule section, if the regex pertains to digit convert it into integer and store yylval. Return the number.
- In the user definition section, define the function yywrap()
- In the definition section of the yacc file, declare the required header files along with the flag variables set to zero. Then define a token as number along with left as '+', '-', 'or', '\*', '/', '%' or '(' )'
- In the rules section, create an arithmetic expression as E. Print the result and return zero.
- Define the following:
  - E: E '+' E (add)
  - E: E '-' E (sub)
  - E: E '\*' E (mul)
  - E: E '/' E (div)
  - If it is a single number return the number.
- In driver code, get the input through yyparse(); which is also called as main function.
- Declare yyerror() to handle invalid expressions and exceptions.
- Build lex and yacc files and compile.

**PROGRAM:**

**Lex File:**

```
% {  
  
#include<stdio.h>  
  
#include "y.tab.h"  
  
int yylval;  
  
% }  
  
%%  
  
[0-9]+ {  
    yylval=atoi(yytext);
```

```
return NUMBER;
```

```
}
```

```
[\t];
```

```
[\n] return 0;
```

```
. return yytext[0];
```

```
%%
```

```
int yywrap()
```

```
{
```

```
return 1;
```

```
}
```

### **Yacc File:**

```
% {
```

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

```
% }
```

```
%token NUMBER
```

```
%left '+'
```

```
%%
```

```
ArithmeticExpression: E {
```

```
    printf("\nResult=%d\n", $$);
```

```
return 0;
```

```
};
```

```
E: E '+' E { $$ = $1 + $3; }
```

```
| NUMBER { $$ = $1; }
```

```
;
```

```
%%
```

```
void main()
```

```
{
```

```
    printf("\nEnter Any Arithmetic expression with addition\n");
```

```
    yyparse();
```

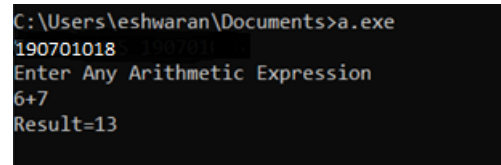
```
}
```

```
void yyerror()
```

```
{
```

```
printf("\nEntered arithmetic expression cannot be computed \n\n");  
}
```

### OUTPUT:



```
C:\Users\eshwaran\Documents>a.exe  
190701018 190701018  
Enter Any Arithmetic Expression  
6+7  
Result=13
```

### RESULT:

Thus, the arithmetic evaluation of expression was implemented using Lex and Yacc.

**Ex No: 7**

**Date :**

**Generate three address codes for a given expression (arithmetic expression, flow of control)**

**AIM:**

To generate the three address codes using the C program.

**ALGORITHM:**

- The expression is read from the file using a file pointer
- Each string is read and the total no. of strings in the file is calculated.
- Each string is compared with an operator; if any operator is seen then the previous string and next string are concatenated and stored in a first temporary value and the three address code expression is printed
- Suppose if another operand is seen then the first temporary value is concatenated to the next string using the operator and the expression is printed.
- The final temporary value is replaced to the left operand value.

**PROGRAM:**

```
#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(op,">=")==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\tT:=0",addr);
addr++;
printf("\n%d\tgoto %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%c\ntemp\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]);

}

```

## OUTPUT

```
C:\Users\eshwaran\Documents>a.exe
190701018
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2

Enter the expression with arithmetic operator:a*b+c
Three address code:
temp=a*b
temp1=temp+c

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:3
Enter the expression with relational operator a
<=
b

100    if a<=b goto 103
101    T:=0
102    goto 104
103    T:=1
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:
```

## RESULT:

Thus, the three address code was implemented successfully using the C programming language.

**Ex No: 8**

**Date:**

## **CODE OPTIMIZATION**

### **CODE OPTIMIZATION:**

The process of code optimization involves

- Eliminating the unwanted code lines
- Rearranging the statements of the code

### **CODE OPTIMIZATION TECHNIQUES:**

#### **1. Compile Time Evaluation**

Two techniques that falls under compile time evaluation are-

##### **A) Constant Folding**

In this technique,

- As the name suggests, it involves folding the constants.
- The expressions that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

**Example:**

$$\text{Circumference of Circle} = (22/7) \times \text{Diameter}$$

Here,

- This technique evaluates the expression  $22/7$  at compile time.
- The expression is then replaced with its result 3.14.
- This saves the time at run time.

##### **B) Constant Propagation**

In this technique,

- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of variable must not get alter in between.

**Example:**

$$\text{pi} = 3.14$$

$$\text{radius} = 10$$

$$\text{Area of circle} = \text{pi} \times \text{radius} \times \text{radius}$$

Here,

- This technique substitutes the value of variables 'pi' and 'radius' at compile time.



- It then evaluates the expression  $3.14 \times 10 \times 10$ .
- The expression is then replaced with its result 314.
- This saves the time at run time.

## 2. Common Sub-Expression Elimination

The expression that has been already computed before and appears again in the code for computation is called as **Common Sub-Expression**.

In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

**Example:**

Code Before Optimization	Code After Optimization
<pre> S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i // Redundant Expression S5 = n S6 = b[S4] + S5 </pre>	<pre> S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5 </pre>

## 3. Code Movement

In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

**Example:**

Code Before Optimization	Code After Optimization
<pre> for ( int j = 0 ; j &lt; n ; j ++ ) { x = y + z ; a[j] = 6 x j ; } </pre>	<pre> x = y + z ; for ( int j = 0 ; j &lt; n ; j ++ ) { a[j] = 6 x j ; } </pre>

## 4. Dead Code Elimination

In this technique,

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

**Example:**

Code Before Optimization	Code After Optimization
<pre>i = 0 ; if (i == 1) { a = x + 5 ; }</pre>	<pre>i = 0 ;</pre>

## 5. Strength Reduction

In this technique,

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

**Example:**

Code Before Optimization	Code After Optimization
<pre>B = A x 2</pre>	<pre>B = A + A</pre>

Here,

- The expression “A x 2” is replaced with the expression “A + A”.
- This is because the cost of multiplication operator is higher than that of addition operator.

**Ex No: 8.a**

**Date :**

## **IMPLEMENT CODE OPTIMIZATION TECHNIQUES LIKE DEAD CODE AND COMMON EXPRESSION ELIMINATION**

**AIM:**

To write a C program to implement the dead code elimination and common expression elimination (code optimization) techniques.

**ALGORITHM:**

- Start
- Create the input file which contains three address code.
- Open the file in read mode.
- If the file pointer returns NULL, exit the program else go to 5.
- Scan the input symbol from left to right.
- Store the first expression in a string.
- Compare the string with the other expressions in the file.
- If there is a match, remove the expression from the input file.
- Perform these steps 5-8 for all the input symbols in the file.
- Scan the input symbol from the file from left to right.
- Get the operand before the operator from the three address code.
- Check whether the operand is used in any other expression in the three address code.
- If the operand is not used, then eliminate the complete expression from the three address code else go to 14.
- Perform steps 11 to 13 for all the operands in the three address code till end of the file is reached.
- Stop.

**PROGRAM:**

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

struct op
{
    char l;
    char r[20];
}
op[10], pr[10];

void main()
{
    int a, i, k, j, n, z = 0, m, q;
    char * p, * l;
    char temp, t;
```

```

char * tem;
clrscr();
printf("enter no of values");
scanf("%d", & n);
for (i = 0; i < n; i++)
{
    printf("\tleft\t");
    op[i].l = getche();
    printf("\tright:\t");
    scanf("%s", op[i].r);
}
printf("intermediate Code\n");
for (i = 0; i < n; i++)
{
    printf("%c=", op[i].l);
    printf("%s\n", op[i].r);
}
for (i = 0; i < n - 1; i++)
{
    temp = op[i].l;
    for (j = 0; j < n; j++)
    {
        p = strchr(op[j].r, temp);
        if (p)
        {
            pr[z].l = op[i].l;
            strcpy(pr[z].r, op[i].r);
            z++;
        }
    }
}
pr[z].l = op[n - 1].l;
strcpy(pr[z].r, op[n - 1].r);
z++;
printf("\nafter dead code elimination\n");
for (k = 0; k < z; k++)
{
    printf("%c\t=", pr[k].l);
    printf("%s\n", pr[k].r);
}

//sub expression elimination
for (m = 0; m < z; m++)
{
    tem = pr[m].r;
    for (j = m + 1; j < z; j++)
    {
        p = strstr(tem, pr[j].r);
        if (p)

```

```

{
    t = pr[j].l;
    pr[j].l = pr[m].l;
    for (i = 0; i < z; i++)
    {
        l = strchr(pr[i].r, t);
        if (l) {
            a = l - pr[i].r;
            //printf("pos: %d",a);
            pr[i].r[a] = pr[m].l;
        }
    }
}
}
}
}
printf("eliminate common expression\n");
for (i = 0; i < z; i++) {
    printf("%c\t", pr[i].l);
    printf("%s\n", pr[i].r);
}
// duplicate production elimination

for (i = 0; i < z; i++)
{
    for (j = i + 1; j < z; j++)
    {
        q = strcmp(pr[i].r, pr[j].r);
        if ((pr[i].l == pr[j].l) && !q)

        {
            pr[i].l = '\0';
            strcpy(pr[i].r, '\0');
        }
    }
}
printf("optimized code");
for (i = 0; i < z; i++)
{
    if (pr[i].l != '\0') {
        printf("%c=", pr[i].l);
        printf("%s\n", pr[i].r);
    } } getch();
}

```

**OUTPUT:**

```
C:\Users\eshwaran\Documents>a.exe
190701018
enter no of values5
      left  a      right: 9
      left  b      right: c+d
      left  e      right: c+d
      left  f      right: b+e
      left  r      right: f
intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f

after dead code elimination
b      =c+d
e      =c+d
f      =b+e
r      =f
eliminate common expression
b      =c+d
b      =c+d
f      =b+b
r      =f

C:\Users\eshwaran\Documents>
```

## RESULT:

Thus the above program was compiled and executed successfully and output is verified.

```
interrslt=(s[2]-48)+(s[4]-48);
```

```

                                break;
                        case '-':
                                interrslt=(s[2]-48)-(s[4]-48);
                                break;
                        case '*':
                                interrslt=(s[2]-48)*(s[4]-48);
                                break;
                        case '/':
                                interrslt=(s[2]-48)/(s[4]-48);
                                break;
                        default:
                                interrslt = 0;
                                break;
                }
                fprintf(fp2,"/*Constant Folding*\n");
                fprintf(fp2,"%c = %lf\n",result,interrslt);
                flag2 = 0;
        }
    } else {
        fprintf(fp2,"Not Optimized\n");
        fprintf(fp2,"%s\n",s);
    }
} else {
    fprintf(fp2,"%s\n",s);
}
}
fscanf(fp1,"%s",s);
}
fclose(fp1);
fclose(fp2);
}

```

**OUTPUT:**  
**OUTPUT.TXT**

```

#include<stdio.h>
int
main()
{
/*Constant Folding*/
a = 6.000000
b=a+10;
}

```



**RESULT:**

Thus the C program was written for copy propagation – A Code Optimization Technique.

**Ex No: 9**

**Date:**

**Generate Target Code (Assembly language) for the given set of Three Address Code**

**AIM:**

To generate assembly language for the three address code using C program.

**ALGORITHM:**

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

**PROGRAM:**

```
#include<stdio.h>

#include<string.h>

#include<ctype.h>

typedef struct
{
    char var[10];
    int alive;
}

regist;

regist preg[10];

void substring(char exp[],int st,int end)
{
    int i,j=0;
    char dup[10]="";
    for(i=st;i<end;i++)
        dup[j++]=exp[i];
    dup[j]='0';
```

```

strcpy(exp,dup);
}
int getregister(char var[])
{
int i;
for(i=0;i<10;i++)
{
if(preg[i].alive==0)
{
strcpy(preg[i].var,var);
break;
}
}
return(i);
}
void getvar(char exp[],char v[])
{
int i,j=0;
char var[10]="";
for(i=0;exp[i]!='\0';i++)
if(isalpha(exp[i]))
var[j++]=exp[i];
else
break;
strcpy(v,var);
}
void main()
{
char basic[10][10],var[10][10],fstr[10],op;

```

```

int i,j,k,reg,vc,flag=0;
printf("\nEnter the Three Address Code:\n");
for(i=0;;i++)
{
    gets(basic[i]);
    if(strcmp(basic[i],"exit")==0)
        break;
}
printf("\nThe Equivalent Assembly Code is:\n");
for(j=0;j<i;j++)
{
    getvar(basic[j],var[vc++]);
    strcpy(fstr,var[vc-1]);
    substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
    getvar(basic[j],var[vc++]);
    reg=getregister(var[vc-1]);
    if(preg[reg].alive==0)
    {
        printf("\nMov R%d,%s",reg,var[vc-1]);
        preg[reg].alive=1;
    }
    op=basic[j][strlen(var[vc-1])];
    substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
    getvar(basic[j],var[vc++]);
    switch(op)
    {
        case '+': printf("\nAdd"); break;
        case '-': printf("\nSub"); break;
        case '*': printf("\nMul"); break;
    }
}

```

```

case '/': printf("\nDiv"); break;

}

flag=1;

for(k=0;k<=reg;k++)

{

if(strcmp(preg[k].var,var[vc-1])==0)

{

printf("R%d, R%d",k,reg);

preg[k].alive=0;

flag=0;

break;

}

}

if(flag)

{

printf(" %s,R%d",var[vc-1],reg);

printf("\nMov %s,R%d",fstr,reg);

}

strcpy(preg[reg].var,var[vc-3]);

}

}

```

## OUTPUT:

```

C:\Users\eshwaran\Documents>a.exe
190701018
Enter the Three Address Code:
a=b+c
a=b-c
c=a*b
exit

The Equivalent Assembly Code is:

Mov R0,b
Add c,R0
Mov a,R0
Mov R1,b
Sub c,R1
Mov a,R1
Mov R2,a
Mul b,R2
Mov c,R2

```

**RESULT:**

Thus, the three address code was implemented successfully using the C programming language.