```
int main(){
  printf("Enter a name for an identifier: ");
  yyparse();
  if(v){
    printf("It is a valid variable name\n");
  }
}
```

4. Aim:-

Program to implement calculator using LEX and YACC.

Algorithm:-

Step 1: start
Step 2: Get input from user.
Step 3: Perform operation according to the operator.
Step 4: Print the result.
Step 5: stop

Program:-

4.l

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

return 0;

%%

4.y

```
%{
#include<stdio.h>
int f=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

%%
Arithmetic Expression : E {
        Printf ("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')' { $$ = $2; }
  | NUMBER { $$ = $1; }
  ;

%%
int yyerror () {
        Printf ("Entered arithmetic expression is invalid
\n\n");
}       f=1;
```

```
void main()
{
    printf("Enter Arithmetic Expression :\n");
    yyparse();
    if (f==0)
        printf("\n");
}
```

Result:-

Yacc programs run successfully and output obtained.

Experiment - 6

NFA to DFA conversion

Aim:- Using c program, simulate to DFA conversion.

Algorithm:-

Step 1 : Start

Step 2 : Input the required array, ie, set of alphabets, set of states initial state, set of final states, transitions.

Step 3 : Initiary $Q' = \phi$

Step 4 : Add q_0 of NFA to Q'. Then find the transition from this start state.

Step 5 : In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.

Step 6 : In DFA, the final state will be all the states which contain f (final states of NFA)

Step 7 : Stop

Result :- Program compiled successfully and output obtained.

Experiment-1

DFA Minimization

Aim:- Using a C program, simulate DFA state minimization.

Algorithm:-

Step 1: Start

Step 2: Get the start state, final state as input and also the transitions.

Step 3: Maintain a stack required for transition from one state to other state.

Step 4: Using pop or push function perform the insertion and deletion of elements when required.

Step 5: Finally conversion has been made to change from regular expression to minimized DFA and the output is displayed as DFA transition table.

Step 6: Stop

Result:- Program compiled successfully and output obtained.

Experiment-8

Shift-Reduce Parser

Aim:- Construct a shift-Reduce parser for a given string in C programming language. Assume a gram... map for the same.

Algorithm:-

Step1:- Start

Step 2: Assume the grammar be:

$E \rightarrow E+E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Step 3: Read input variables

Step 4: Stack symbols.

Step 5: Loop forever.

For top-of-stack symbol, s, and next input symbol, a case action of T [s, a]

Step 6: shift x: [x is a STATE number]

Push a,

then x on the top of the stack and advance IP to point to the next input symbol.

Step 7: Reduce y: [y is a PRODUCTION number]

Assume that the production is of the form A -> beta

Step 8: Pop 2 * |β| 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.

step 9: Output the production A -> beta

step 10: Accept : return -- a successful parse.

step 11: Default: error -- the input string is not in the language.

step 12: stop

Result:- Program compiled successfully and output obtained.

Experiment - 9

Recursive Descent Parser for an Expression

Aim:- Construct a Recursive Descent parser for an expression in a programming language. Assume a grammer for the same.

Program:-

Step 1: Start

Step 2: Assume the grammer be

$E \rightarrow TE'$

$E' \rightarrow +TE \mid @$ "@ represents null character"

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid @$

$F \rightarrow (E) \mid ID$

Step 3: Read the input string.

Step 4: Write procedures for the non terminals.

Step 5: Verify the next token equals to non terminals if it satisfies, match the non terminal.

Step 6: If the input string does not match print errors.

Step 7: Else print string accepted.

Step 8: Stop

Result:- Program compiled successfully and output obtained.

Experiment - 10

Intermediate Code Generation

Aim:- Using lex and yacc, simulate three-address code generation of mathematical expressions.

Algorithm:-

Lex program

Step 1: Start

Step 2: In Rule Section, compare yytext with regular expression if yytext match [0-9]+

return DIGIT

If yytext match [A-Z a-z] [A-Z a-z 0-9]*

return ID

If yytext match [+ - / * \n]

return *yytext

Step 3: Stop

Yacc Program

Step 1: Start

Step 2: Take Token generated in the lex program.

Step 3: Apply production rule to generate three-address code by storing them in temporary variables.

S: S ID '=' E '\n' | S E '\n' | S '\n' ;

E: DIGIT | ID | E '+' E | E '-' E | E '*' E | E '/' E | '-' E | '(' E ')';

Step 4: In main function call yyparse()

Step 5: Stop

Result:- Program compiled successfully and output obtained.

Experiment - 11

Implementing Backend of Compiler

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

Algorithm:-

Step 1 : start

Step 2: Read three-address code.

Step 3: Check the operation in it and copy to opr variable

Step 4: Print mov and variable R1.

Step 5: Print operation second variable Ri

Step 6: Print mov Ri, result variable.

Step 7: Stop

Result:- Program executed and output obtained successfully.