PROGRAM TO RECOGNIZE WHILE LOOP

Ex.No:3.c

Date:

AIM:

To write a yacc program to recognize while loop.

ALGORITHM:

Step1: Start the program.

Step2: Reading an expression .

Step3: Checking the validating of the given while loop according to the rule using yacc.

Step4: Print the result of the given while loop

Step5: Stop the program.

LEX program(wh.l):

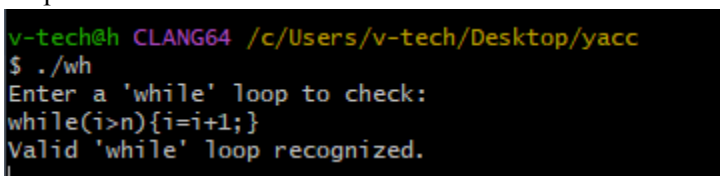
```
%{
#include "wh.tab.h"
%}

%option noyywrap
%option nounput

%%

"while"          { return WHILE; }
"("              { return LPAREN; }
")"              { return RPAREN; }
"{"              { return LBRACE; }
"}"              { return RBRACE; }
"="              { return ASSIGN; }
"<"              { return LT; }
">"              { return GT; }
"=="             { return EQ; }
"+"              { return PLUS; }
"-"              { return MINUS; }
"*"              { return MULT; }
"/"              { return DIV; }
";"              { return ';'; }  // Allow semicolon without error
[0-9]+           { yylval = atoi(yytext); return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* { return IDENTIFIER; }
[ \t\n]+         { /* skip whitespace */ }
.                { printf("Unexpected character: %s\n", yytext); return yytext[0]; }


<<EOF>>          { return 0; }
%%
```

YACC Program(wh.y):

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void yyerror(const char *s);
int yylex(void);
%}

/* Define tokens */
%token WHILE LPAREN RPAREN LBRACE RBRACE SEMICOLON IDENTIFIER NUMBER
%token ASSIGN LT GT EQ PLUS MINUS MULT DIV
%left PLUS MINUS
%left MULT DIV
%left LT GT EQ
%right ASSIGN
%%

program:
   statement
   ;

statement:
   WHILE LPAREN condition RPAREN LBRACE statements RBRACE
   {
      printf("Valid 'while' loop recognized.\n");
   }
   ;

statements:
   statement
   | assignment ';'
   | statements statement
   | /* empty */
   ;

assignment:
   IDENTIFIER ASSIGN expression
   ;

condition:
   expression LT expression
   | expression GT expression
   | expression EQ expression
   ;

expression:
    IDENTIFIER    | NUMBER
   | expression PLUS expression
   | expression MINUS expression
   | expression MULT expression
   | expression DIV expression
   ;
```

```
statement:
    IDENTIFIER '=' expression
    | IDENTIFIER
    | NUMBER
    ;

%%


void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main(void) {
    printf("Enter a 'while' loop to check:\n");
    yyparse();
    return 0;
}
```

Output:



CONCLUSION:

Thus a program to recognize while loop was executed successfully.

IMPLEMENTATION OF CALCULATOR USING LEX & YACC

Ex.No:3.d
Date:

AIM:

To write a program for implementing a calculator for computing the given expression using semantic rules of the YACC tool and LEX.

ALGORITHM:

Step 1: Start the program.
Step 2: Include the necessary header files.
Step 3: Use a function for printing the error message.
Step 4: Get the input from the user and parse it.
Step 5: Check the input is a valid expression or not.
Step 6: Write separate operations for addition, subtraction, multiplication and division using the expr and matching it with the operators in the in the input.
Step 7: Print the error messages for the invalid operators.
Step 8: Print the output of the expression. Step 9: Terminate the program.

LEX program(Calc.l):

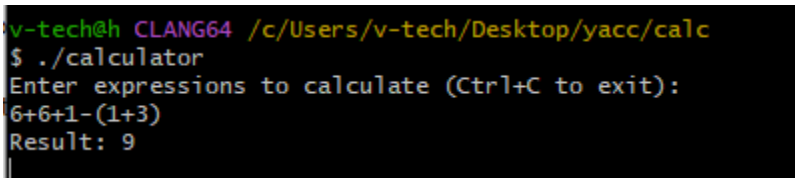
```
%{
#include "calc.tab.h"  // Include the parser's header for token definitions
%}

%%

[0-9]+          { yylval = atoi(yytext); return NUMBER; }
[ \t]+          { /* Ignore whitespace */ }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return MULT; }
"/"             { return DIV; }
"("             { return LPAREN; }
")"             { return RPAREN; }
\n              { return '\n'; }  // Recognize newline as a token
.               { printf("Unexpected character: %s\n", yytext); }

%%
```

YACC program(Calc.y):

```
%{
#include <stdio.h>
#include <stdlib.h>

void yyerror(const char *s);
int yylex();
%}

%token NUMBER
%token PLUS MINUS MULT DIV LPAREN RPAREN

%left PLUS MINUS
%left MULT DIV
```

```
%%

input:
   /* empty */
   | input line
   ;

line:
   '\n'
   | expression '\n'   { printf("Result: %d\n", $1); }
   ;

expression:
   NUMBER              { $$ = $1; }
   | expression PLUS expression    { $$ = $1 + $3; }
   | expression MINUS expression   { $$ = $1 - $3; }
   | expression MULT expression    { $$ = $1 * $3; }
   | expression DIV expression     { $$ = $1 / $3; }
   | LPAREN expression RPAREN      { $$ = $2; }
   ;

%%

int main(void) {
   printf("Enter expressions to calculate (Ctrl+C to exit):\n");
   yyparse();
   return 0;
}

void yyerror(const char *s) {
   fprintf(stderr, "Error: %s\n", s);
}
```

Output:



CONCLUSION:

Thus a program to implement the calculator using lex & yacc was executed successfully

Implementation of Three Address Code using LEX and YACC

Ex.No:4
Date:

AIM:

To write a program for implementing Three Address Code using LEX and YACC.

ALGORITHM:

Step1: A Yacc source program has three parts as follows
Declarations %% translation rules %% supporting C routines
Step2: Declarations Section: This section contains entries that:
i. Include standard I/O header file.
ii. Define global variables.
iii. Define the list rule as the place to start processing.
iv. Define the tokens used by the parser. v. Define the operators and their precedence.
Step3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.
Step4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.
Step5: Main- The required main program that calls the yyparse subroutine to start the program.
Step6: yyerror(s) -This error-handling subroutine only prints a syntax error message.
20
Step7: yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmar file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.
Step8: calc.lex contains the rules to generate these tokens from the input stream.

PROGRAM:
LEX part (tac.l):

```
%{
#include "tac.tab.h"
#include <stdio.h>
#include <stdlib.h>
%}

%%


[0-9]+        { yylval.symbol = atoi(yytext); return NUMBER; }
[a-zA-Z]     { yylval.symbol = yytext[0]; return LETTER; }
"+"          { return '+'; }
"-"          { return '-'; }
"*"          { return '*'; }
"/"          { return '/'; }
"="          { return '='; }
";"          { return ';'; }
"("          { return '('; }
")"          { return ')'; }
[ \t\n]      { /* skip whitespace */ }
.            { printf("Unexpected character: %s\n", yytext); }
```

```
%%
int yywrap() {
   return 1; // Indicate no more input
}

Yacc program(tac.y):
%{
#include <stdio.h>
#include <stdlib.h>

int yylex(void); // Declaration of yylex
struct expr {
   char operand1;
   char operand2;
   char operator;
   char result;
};

char addtotable(char a, char b, char o);
void threeAdd();
void yyerror(const char *s);

// Global variables
struct expr arr[20];
int index1 = 0;
char temp = 'A' - 1;
%}

%union {
   char symbol;
}

%left '+' '-'
%left '*' '/'
%token <symbol> LETTER NUMBER
%type <symbol> exp

%%

// Grammar rules
statement:
   LETTER '=' exp ';' { addtotable($1, $3, '='); }
   ;

exp:
   exp '+' exp { $$ = addtotable($1, $3, '+'); }
   | exp '-' exp { $$ = addtotable($1, $3, '-'); }
   | exp '*' exp { $$ = addtotable($1, $3, '*'); }
   | exp '/' exp { $$ = addtotable($1, $3, '/'); }
   | '(' exp ')' { $$ = $2; }
   | NUMBER { $$ = $1; }
   | LETTER { $$ = $1; }
   ;
```

```
%%

// Error handling function
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

// Function to add a record to the table
char addtotable(char a, char b, char o) {
    temp++;
    arr[index1].operand1 = a;
    arr[index1].operand2 = b;
    arr[index1].operator = o;
    arr[index1].result = temp;
    index1++;
    return temp;
}

// Function to display the three-address code
void threeAdd() {
    for (int i = 0; i < index1; i++) {
        printf("%c := %c %c %c\n", arr[i].result, arr[i].operand1, arr[i].operator, arr[i].operand2);
    }
}

// Main function
int main() {
    printf("Enter the expression: ");
    yyparse();
    threeAdd();
    return 0;
}
```

Output:

```
v-tech@h CLANG32 /c/Users/v-tech/Desktop/yacc/tac
$ flex tac.l

v-tech@h CLANG32 /c/Users/v-tech/Desktop/yacc/tac
$ bison -d tac.y

v-tech@h CLANG32 /c/Users/v-tech/Desktop/yacc/tac
$ gcc -o tac tac.tab.c lex.yy.c -lfl

v-tech@h CLANG32 /c/Users/v-tech/Desktop/yacc/tac
$ ./tac
Enter the expression:
a=b+c*d-e
Error: syntax error
A := c * d
B := b + A
C := B - e
```

Conclusion:

Thus the program for implementing Three Address Code using LEX and YACC was executed successfully.