CSI THOOTHUKUDI-NAZARETH DIOCESE

DR.G.U.POPE COLLEGE OF ENGINEERING

POPE NAGAR SAWYERPURAM-628 251

Register No :

Certified that this is the bonafide record of work done by

Selvan/Selvi ………………………………………………………………………………….…....

of ………. Semester ……….. branch for the lab ……………………………………….....

During the academic year ……………………………

Staff In-charge **H. O .D**

Submitted for the university practical Examination held on ……………………….

Internal Examiner **External Examiner**

S.NO	TOPIC	PAGE.NO	MARKS	SIGNATURE
1.				
2.				
3a.				
3b.				
3c.				
3d.				
4.				
5.				
6.				
7.				

EX NO :01	DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C
DATE :	

AIM:

To develop a lexical analyzer to identify identifiers,constants,comments,operators etc using C program

ALGORITHM:

STEP 1 : Initialize input string and pointer.
STEP 2 : Loop through the input string.
STEP 3 : Skip whitespace, identify if the current character is an identifier, number, or operator/unknown.
STEP 4 : Print the identified token type.
STEP 5 : Repeat until the end of the string is reached.

PROGRAM:

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

void printToken(char *type, char *value){
        printf("%s: %s\n", type, value);
 }

void getNextToken(char **input) {
   char token[100];
   int i = 0;
   while (isspace(**input)) (*input)++;
   if (isalpha(**input)) {
   while (isalpha(**input) || isdigit(**input)) token[i++] = *(*input)++;
        printToken("IDENTIFIER", token);
   }
   else if (isdigit(**input)) { while (isdigit(**input)) token[i++] = *(*input)++;
           printToken("NUMBER",token);
   }
   else if (**input != '\0') { token[i++] = *(*input)++;
           printToken("OPERATOR/UNKNOWN", token);
   }
   token[i] = '\0';
}

int main() {
   char input[] = "int a = 5 + 3;", *ptr = input;
   while (*ptr != '\0') getNextToken(&ptr);
   return 0;
}
```

OUTPUT :

```
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ gcc -o  EXN01 EXN01.c
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ ./EXN01
IDENTIFIER: int
IDENTIFIER: ant
OPERATOR/UNKNOWN: =
NUMBER: 5
OPERATOR/UNKNOWN: +
NUMBER: 3
OPERATOR/UNKNOWN: ;
```

RESULT:

Thus the program for lexical analyzer to recognize a few patterns program in C has been successfully executed.

EX NO :02	IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL
DATE :	

AIM:

To develop the lex to extract tokens from a given source code.

ALGORITHM :

STEP 1 : Check for the lexical tool version before working.
STEP 2 : If not installed , install it using the prefered commands.
STEP 3 : Feed your program to the code editor , covert it into lexical program using lex command.
STEP 4 : Compile your program using special lexical compiler which would be installed with your lex analyzer.
STEP 5 : Run the program, execute by the lexical tool and observe your output.

PROGRAM:

Installation - sudo apt-get install flex
Version - yacc --version
Flex file - <file name>.l
Flex Generate - flex <filename.l>
Compilation - gcc lex.yy.c
Run - ./a.out

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

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

%%
[0-9]+                    { printf("NUMBER: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]*    { printf("IDENTIFIER: %s\n", yytext); }
"+"                       { printf("PLUS\n"); }
"-"                       { printf("MINUS\n"); }
"*"                       { printf("MULTIPLY\n"); }
"/"                       { printf("DIVIDE\n"); }
\n                        { /* Ignore newline */ }
[ \t]+                    { /* Ignore whitespace */ }
.                         { printf("UNKNOWN CHARACTER: %c\n", yytext[0]); }
%%

int main(void) {
   yylex();
   return 0;
}
void yyerror(const char *s)
   {fprintf(stderr, "Error: %s\n", s);

}
```

OUTPUT :

```
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ flex EXNO2.l
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ gcc lex.yy.c -o lexer -l
fl
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ ./lexer
int A,B;float X,Y;
IDENTIFIER: int
IDENTIFIER: A
UNKNOWN CHARACTER: ,
IDENTIFIER: B
UNKNOWN CHARACTER: ;
IDENTIFIER: float
IDENTIFIER: X
UNKNOWN CHARACTER: ,
IDENTIFIER: Y
UNKNOWN CHARACTER: ;
```

RESULT:

Thus the execution of lex program is implemented successfully.

EX NO :03	GENERATE YACC SPECIFICATION FOR A FEW SYNTACTIC CATEGORIES
DATE :	

AIM:

To write a program to do exercise on syntax analysis using yacc

ALGORITHM:

STEP 1 : Read an expression from the user, allowing for multiple entries until the user decides to exit.
STEP 2 : Use the lexer to tokenize the input, recognizing operators, identifiers, and numbers.
STEP 3 : Pass the tokens to the parser to construct a syntax tree based on defined grammar rules.
STEP 4 : Perform actions based on the parsed structure and validate the expression.
STEP 5 : Catch and report any syntax or lexical errors during parsing and tokenization

PROGRAM:

Installation - sudo apt-get install bison
Version - yacc --version
Flex file - <file name>.l
Yacc file - <file name>.y
Flex Generate - flex <filename.l>
Yacc Generate - yacc -d <file name.y>
Compilation - gcc lex.yy.c y.tab.c
Run - ./a.out

A) PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATOR +,-,* ,/

LEX CODE: File name .l

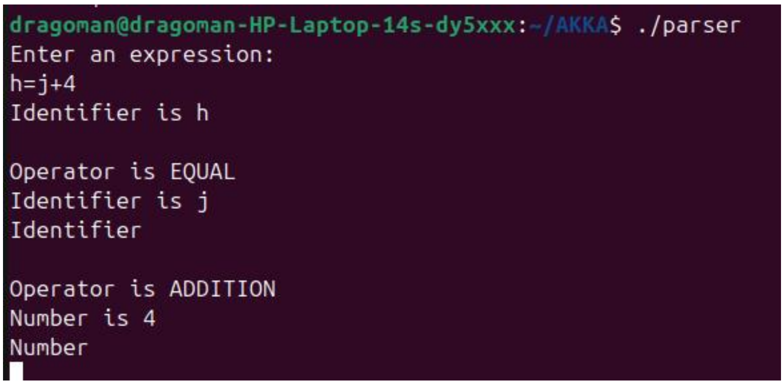
```
%{
#include "y.tab.h"
#include <stdio.h>
#include <stdlib.h>

void yyerror(const char *s);
%}

%%

"="             { printf("\nOperator is EQUAL\n"); return '='; }
"+"             { printf("\nOperator is ADDITION\n"); return '+'; }
"-"             { printf("\nOperator is SUBTRACTION\n"); return '-'; }
"*"             { printf("\nOperator is MULTIPLICATION\n"); return '*'; }
"/"             { printf("\nOperator is DIVISION\n"); return '/'; }
[0-9]+          { printf("Number is %s\n", yytext); return NUMBER; }
[a-zA-Z]+[0-9]* {
   printf("Identifier is %s\n", yytext);
   return ID;
```

```
}
\n          { /* ignore newlines */ }
[ \t]+      { /* ignore whitespace */ }
.           { fprintf(stderr, "Unexpected character: %c\n", yytext[0]); exit(1); }

%%

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

YACC CODE: **File name.y**

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

void yyerror(const char *s);
%}

%token ID NUMBER

%%

statement:
    ID '=' E        { printf("\nValid arithmetic expression\n"); }
    | E             { printf("\nValid arithmetic expression\n"); }
    ;

E:
    E '+' E                 { printf("Addition\n"); }
    | E '-' E               { printf("Subtraction\n"); }
    | E '*' E               { printf("Multiplication\n"); }
    | E '/' E               { printf("Division\n"); }
    | ID                    { printf("Identifier\n"); }
    | NUMBER                { printf("Number\n"); }
    ;

%%

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

int main() {
    printf("Enter an expression:\n");
    yyparse();
    return 0;
}
```

OUTPUT:



```
dragoman@dragoman-HP-Laptop-14s-dy5xxx:~/AKKA$ ./parser
Enter an expression:
h=j+4
Identifier is h

Operator is EQUAL
Identifier is j
Identifier

Operator is ADDITION
Number is 4
Number
```

B) PROGRAM TO RECOGNIZE A VALID WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS.

ALGORITHM:

STEP1: The lexer identifies keywords (int, float, double), identifiers, and handles whitespace while returning appropriate tokens.

STEP2: For identifiers, the lexer prints the recognized identifier name using yytext before returning the ID token.

STEP3: The parser processes a series of declarations, supporting multiple variable declarations of the same type with a comma-separated list.

STEP4: Each grammar rule defines how types are associated with identifiers, allowing for complex declarations like int a, b;.

STEP5: The parser invokes yyerror to report any syntax errors encountered during parsing, aiding in debugging.

LEX CODE: File name .l

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

%%
"int"      { return INT; }
"float"    { return FLOAT; }
"double"   { return DOUBLE; }
[a-zA-Z][a-zA-Z0-9]* {
   printf("Identifier is %s\n", yytext);
   return ID;
}
[ \t]+     ; /* Ignore whitespace */
\n         { return '\n'; }
.          { return yytext[0]; }
%%

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

<u>YACC CODE:</u> File name.y

```
%{
#include <stdio.h>
#include <stdlib.h>
extern int yylex();
extern char* yytext;
void yyerror(const char *s);
%}

%token ID INT FLOAT DOUBLE

%%
program: declarations
;

declarations: declarations declaration
            | declaration
;

declaration: type IDs
;

type: INT
    | FLOAT
    | DOUBLE
;

IDs: ID
   | ID ',' IDs
;

%%

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

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

<u>OUTPUT:</u>

```
technocrats@technocrats-HP-Laptop-15-fd0xxx:~$ cd COMPILER
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ flex EXNO3B.l
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ yacc -d EXNO3B.y
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ gcc lex.yy.c y.tab.c -o
parser -lfl
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ ./parser
int x,y
Identifier is x
Identifier is y
```

C) PROGRAM TO RECOGNIZE THE GRAMMAR(ABD WHERE N>=10)

ALGORITHM:

STEP1: The lexer recognizes characters a and b, returning tokens A and B, while also passing through any other characters and newline characters.
STEP2: The lexer processes input until it encounters a newline, preparing tokens for the parser to analyze the structure of the input string.
STEP3: The parser is defined to expect a sequence of ten A tokens followed by an anb structure, ensuring that valid strings conform to the format aaaaaaaab.
STEP4: The anb rule allows for recursive construction of sequences, ensuring at least ten A tokens are present before possibly concluding with a B.
STEP5: The parser utilizes yyerror to print "Invalid string" if the input does not match the expected format, while successfully matched strings output "Valid string."

LEX CODE: anb.l

```
%{

#include "y.tab.h"
%}

%%
a        { return A; }
b        { return B; }
.        { return yytext[0]; }
\n       { return '\n'; }

%%

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

YACC CODE: anb.y

```
%{
/* YACC program for recognizing anb (n >= 10) */
#include <stdio.h>
%}

%token A B

%%

stmt: A A A A A A A A A A anb '\n' {
    printf("\nValid string\n");
}
;

anb: A anb | A B ;

%%

int main() {
```

```
    printf("\nEnter some valid string:\n");
    yyparse();
    return 0;
}

void yyerror(char *s) {
    printf("\nInvalid string\n");
}
```

OUTPUT:


```
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ ./a.out

Enter some valid string:
aaaaaaaaaaab

Valid string
^C
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ ./a.out

Enter some valid string:
aaaaaaaaaab

Invalid string
```

D) IMPLEMENTATION OF CALCULATOR USING LEX AND YACC

ALGORITHM:

STEP1: The lexer identifies numbers and operators (+, -, *, /, (,)) and returns corresponding tokens, while ignoring whitespace and handling unrecognized characters with an error message.
STEP2: Tokens for numbers and operators are defined, enabling the parser to recognize and process arithmetic expressions.
STEP3: The parser's grammar rules support basic arithmetic operations, allowing for addition, subtraction, multiplication, and division, with proper handling of parentheses.
STEP4: The parser checks for division by zero and invokes an error function to print appropriate error messages, preventing runtime errors.
STEP5: The main function runs an interactive calculator, continuously parsing user input until an error occurs or the user exits.

LEX CODE: File name .l

```
%{
#include "y.tab.h"
#include <stdlib.h>
void yyerror(const char *s);
%}

%%

[0-9]+          { yylval = atoi(yytext); return NUMBER; }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return MULTIPLY; }
"/"             { return DIVIDE; }
```

```
"("             { return LPAREN; }
")"             { return RPAREN; }
[ \t]+          { /* ignore whitespace */ }
\n              { return 0; } // Handle new lines
.               { yyerror("Unrecognized character"); }

%%

// Define yywrap function
int yywrap() {
   return 1;
}
```

YACC CODE: File name.y

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

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

int yylex(void); // Declare yylex
%}

%token NUMBER
%token PLUS MINUS MULTIPLY DIVIDE LPAREN RPAREN

%%

// Grammar rules
expr:
   expr PLUS expr { printf("Result: %g\n", (double)($1 + $3)); }
   | expr MINUS expr { printf("Result: %g\n", (double)($1 - $3)); }
   | expr MULTIPLY expr { printf("Result: %g\n", (double)($1 * $3)); }
   | expr DIVIDE expr {
      if ($3 == 0) {
         yyerror("divide by zero");
         $$ = 0; // Avoid division by zero
      } else {
         printf("Result: %g\n", (double)($1 / $3));
      }
   }
   | LPAREN expr RPAREN { $$ = $2; }
   | NUMBER { $$ = $1; }
   ;

%%
int main() {
   printf("Simple Calculator:\n");
   while (yyparse() == 0); // Loop until an error or EOF
   return 0;
}
```

OUTPUT:

```
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ ./calci
Simple Calculator (Ctrl+C to exit):
1+2
Result: 3
^C
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ ./calci
Simple Calculator (Ctrl+C to exit):
1/1
Result: 1
^C
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/COMPILER$ ./calci
Simple Calculator (Ctrl+C to exit):
6/3
Result: 2
```

RESULT:

Thus the yacc specification for a few syntactic categories program was successfully implemented.

EX NO :04	GENERATE THREE ADDRESS CODE FOR A SIMPLE PROGRAM USING PROGRAM USING LEX AND YACC
DATE :	

AIM:

To generate three address code for a simple program using LEX and YACC

ALGORITHM:

STEP 1 : The lexer processes input strings to identify tokens using regular expressions.
STEP 2 : The parser uses the defined grammar rules to build a parse tree, processing statements and expressions based on operator precedence and associativity.
STEP 3 : For each grammar rule, semantic actions are executed to print intermediate results and build expressions.
STEP 4 : Strings are dynamically allocated using strdup, requiring careful memory management to prevent leaks.
STEP 5 : The parser invokes yyerror for syntax errors, providing feedback to the user when the input does not conform to the expected grammar.

PROGRAM :

FLEX CODE: File name.l

```
%{
#include "y.tab.h"
#include <stdlib.h>
#include <string.h>
%}
%%
[a-z]+      { yylval.sval = strdup(yytext); return IDENT; }
[0-9]+      { yylval.sval = strdup(yytext); return NUM; }
"+"         { return PLUS; }
"*"         { return MUL; }
"="         { return ASSIGN; }
\n          { return EOL; }
%%
```

YACC CODE: File name.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern int yylex();
int t = 0;
void yyerror(const char *s) { fprintf(stderr, "Error: %s\n", s); }
%}

%union {
   char *sval;
   int ival;
```

```
}

%token <sval> IDENT NUM
%token PLUS MUL ASSIGN EOL

%type <sval> expr term factor

%%

stmt: IDENT ASSIGN expr EOL { printf("%s = %s\n", $1, $3); }
expr: term { $$ = $1; } | expr PLUS term { printf("t%d = %s + %s\n", t++, $1, $3); $$ = strdup("t0"); }
term: factor { $$ = $1; } | term MUL factor { printf("t%d = %s * %s\n", t++, $1, $3); $$ = strdup("t0"); }
factor: IDENT { $$ = $1; } | NUM { $$ = $1; }

%%

int main() { return yyparse(); }
```

OUTPUT:

RESULT:

Thus the three address code for a simple program using program using lex and yacc has been executed successfully.

EX NO :05 DATE :	IMPLEMENTATION OF TYPE CHECKING

AIM:

To write a c program for implementing of type checking for given expressions

ALGORITHM:

STEP 1: The lexer defines tokens for integers, floats, identifiers, and arithmetic operators, along with rules for recognizing valid expressions and statements.

STEP 2: The parser processes a program consisting of statements, which include variable type declarations, variable assignments, and expressions.

STEP 3: It checks for type consistency during assignments by tracking the expected type of variables.

STEP 4: The parsing rules allow for basic arithmetic expressions, ensuring they conform to type expectations.

STEP 5: Error handling is incorporated to report unexpected characters and type mismatches during parsing.

PROGRAM:

FLEX CODE: File name.l

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

%%

int         { return INT; }
float       { return FLOAT; }
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[0-9]+"."[0-9]+ { yylval = atof(yytext); return FNUMBER; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }  // Variable names (identifiers)
"="         { return ASSIGN; }                 // Assignment operator
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MUL; }
"/"         { return DIV; }
";"         { return SEMI; }
[ \t\n]     { /* Ignore whitespaces */ }
.           { printf("Unexpected character: %s\n", yytext); }

%%

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

YACC CODE: File name.y

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

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

enum { TYPE_INT, TYPE_FLOAT }; // For type checking

int current_type = TYPE_INT;
%}

%token INT FLOAT NUMBER FNUMBER IDENTIFIER
%token PLUS MINUS MUL DIV ASSIGN SEMI

%%

program:
   program stmt SEMI
   | /* empty */
   ;

stmt:
   type var ASSIGN expr
   {
      if ($4 == current_type) {
         printf("Valid assignment.\n");
      } else {
         printf("Type error: Mismatched types in assignment.\n");
      }
   }
   ;

type:
   INT
   {
      current_type = TYPE_INT;
   }
   | FLOAT
   {
      current_type = TYPE_FLOAT;
   }
   ;

var:
   IDENTIFIER
   {
      // You can add variable tracking here
   }
   ;
```

```
expr:
    NUMBER
    {
        $$ = TYPE_INT;
    }
    | FNUMBER
    {
        $$ = TYPE_FLOAT;
    }
    | expr PLUS expr
    | expr MINUS expr
    | expr MUL expr
    | expr DIV expr
    ;

%%

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

int main(void) {
    return yyparse();
}
```

OUTPUT:

```
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/EXN05$ gcc lex.yy.c parser.tab.c -
o parser -lfl
technocrats@technocrats-HP-Laptop-15-fd0xxx:~/EXN05$ ./parser
int x = 6;
Valid assignment.
float b=3;
Type error: Mismatched types in assignment.
```

RESULT:

Thus the type checking was executed successfully .

EX NO :06	**IMPLEMENTATION OF CODE OPTIMIZATION TECHNIQUE**
DATE :	

AIM:

To write a program for implementation of code optimization technique.

ALGORITHM:

STEP1: Get the number of operations n and then read each operation's left-hand side character L and right-hand side string r into the op array.
STEP2: Check if each L is used in any other operation's r. If not, mark it as alive and store it in the pr array.
STEP3: Identify and remove operations with duplicate right-hand side strings by nullifying their left-hand side characters.
STEP4: Output the optimized operations from the pr array where the left-hand side character is not null.
STEP5: Complete execution and exit the program.

PROGRAM:

```
#include &lt;stdio.h&gt;
#include &lt;string.h&gt;
#define MAX 10
struct op {
   char L;
   char r[20];
} op[MAX], pr[MAX];
int main() {
   int n, z = 0;
   printf(&quot;Enter the Number of values: &quot;);
   scanf(&quot;%d&quot;, &amp;n);
   getchar();

   for (int i = 0; i &lt; n; i++) {

      printf(&quot;Left: &quot;);
      scanf(&quot;%c&quot;, &amp;op[i].L);
      printf(&quot;Right: &quot;);
      scanf(&quot;%s&quot;, op[i].r);
      getchar();
   }

   for (int i = 0; i &lt; n; i++) {
      int is_dead = 0;
      for (int j = 0; j &lt; n; j++) {
         if (strchr(op[j].r, op[i].L)) {
            is_dead = 1;
            break;
         }
      }
```

```
        if (!is_dead) {
            pr[z++] = op[i];
        }
    }

    for (int i = 0; i &lt; z; i++) {
        for (int j = i + 1; j &lt; z; j++) {
            if (strcmp(pr[i].r, pr[j].r) == 0) {
                pr[j].L = &#39;\0&#39;;
            }
        }
    }
    printf(&quot;OPTIMIZED CODE\n&quot;);
    for (int i = 0; i &lt; z; i++) {
        if (pr[i].L != &#39;\0&#39;) {
            printf(&quot;%c = %s \n&quot;, pr[i].L, pr[i].r);
        }
    }
    return 0;
}
```

OUTPUT:

```
dragoman@dragoman-HP-Laptop-14s-dy5xxx:~/AKKA$ ./EXNO_6
Enter the Number of values: 4
Left: A
Right: X
Left: B
Right: Y
Left: C
Right: X
Left: D
Right: Y
OPTIMIZED CODE
A = X
B = Y
```

RESULT:

Thus the code optimization technique has been successfully executed.

EX NO :07	**IMPLEMENTATION OF BACKEND OF THE COMPILER**
DATE :	

AIM:

To develop the Backend of the compiler the target assembly instructions can be simple move,add,sub,jump also simple addressing modes are used.

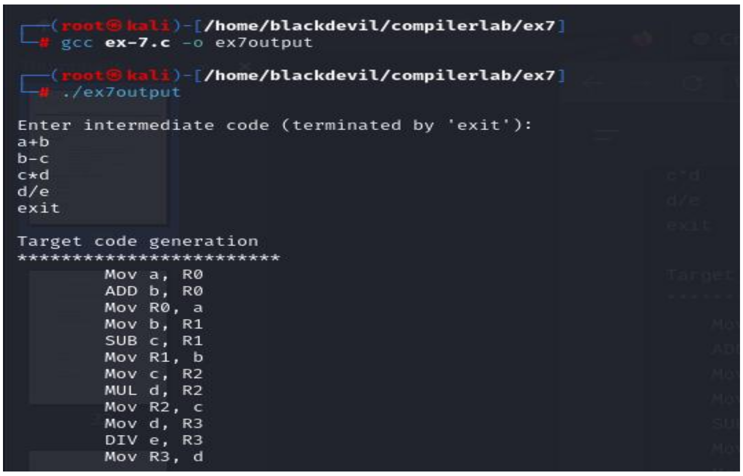
ALGORITHM:

STEP1: The program prompts the user to enter intermediate code lines until "exit" is entered, storing them in a 2D array.
STEP2: It iterates through the stored intermediate code, extracting operations and operands for each line.
STEP3: A switch statement identifies the operation based on the second character of the string, mapping it to corresponding assembly-like instructions (ADD, SUB, etc.).
STEP4: For each line of intermediate code, it generates and prints the appropriate target code, moving operands to registers and performing the identified operation.
STEP5: The output format ensures each operation is correctly structured, reflecting the source and destination registers.

PROGRAM:

```
#include <stdio.h>
#include <string.h>
void main() {
        char icode[10][30], str[20], opr[10];
        int i = 0;
        printf("\nEnter intermediate code (terminated by 'exit'):\n");
       while (scanf("%s", icode[i]), strcmp(icode[i],"exit") !=0)i++;
        printf("\nTarget code generation\n************************");
        for (i = 0; i < 10 && strcmp(icode[i], "exit") != 0; i++){
                strcpy(str, icode[i]);
                switch (str[1]) {
                        case '+': strcpy(opr, "ADD"); break;
                        case '-': strcpy(opr, "SUB"); break;
                        case '*': strcpy(opr, "MUL"); break;
                        case '/': strcpy(opr, "DIV"); break;
                        default: strcpy(opr, "UNKNOWN"); break;
        }
              printf("\n\tMov %c, R%d", str[0], i);
              printf("\n\t%s %c, R%d", opr, str[2], i);
              printf("\n\tMov R%d, %c", i, str[0]);
        }
}
```

OUTPUT:



```
┌──(root㉿kali)-[/home/blackdevil/compilerlab/ex7]
└─# gcc ex-7.c -o ex7output

┌──(root㉿kali)-[/home/blackdevil/compilerlab/ex7]
└─# ./ex7output

Enter intermediate code (terminated by 'exit'):
a+b
b-c
c*d
d/e
exit

Target code generation
************************
        Mov a, R0
        ADD b, R0
        Mov R0, a
        Mov b, R1
        SUB c, R1
        Mov R1, b
        Mov c, R2
        MUL d, R2
        Mov R2, c
        Mov d, R3
        DIV e, R3
        Mov R3, d
```

RESULT:

Thus the Backend of the compiler program was successfully executed.