NOIDA INSTITUE OF ENGINEERING AND TECHNOLOGY GREATER NOIDA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COMPLIER DESIGN LAB (ACSE0554)

Report File

Submitted By:

KARAN TIWARI

2201330109018

Submitted To:

MR. VIVEK KR. SHARMA

MR. IBRAR AHMAD

Assistant Professor, CSE

INDEX

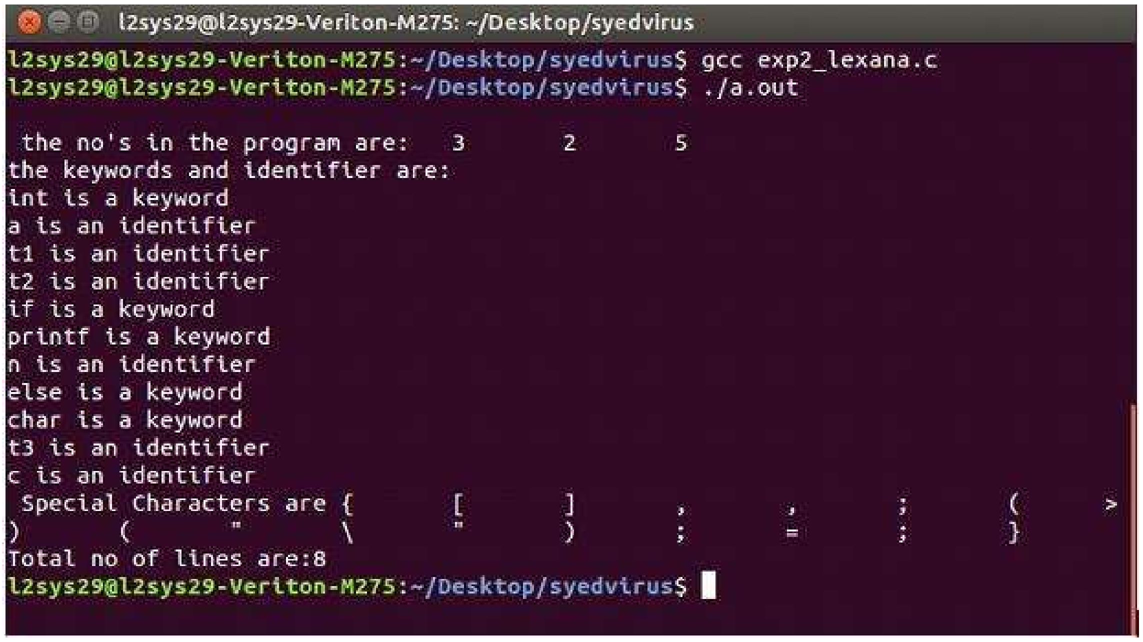
PRACTICAL- 1

Develop a lexical analyzer to recognize few patterns in C. (Ex. identifiers, constants, comments, operators etc.).

SOURCE CODE:-

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
#include<stdlib.h>
void keyword(char str[10])
{
if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strcmp("int",str)==
0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("printf
",str)==0||strcmp("switch",str)==0||strcmp("case",str)==0)
  printf("\n%s is a keyword",str);
else
  printf("\n%s is an identifier",str);
}
void main()
{
FILE *f1,*f2,*f3;
char c,str[10],st1[10];
int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF)
{
  if(isdigit(c))
  {
  tokenvalue=c-'0';
  c=getc(f1);
  while(isdigit(c))
  {
   tokenvalue*=10+c-'0';
   c=getc(f1);
  }
  num[i++]=tokenvalue;
  ungetc(c,f1);
  }
  else
  if(isalpha(c))
  {
   putc(c,f2);
   c=getc(f1);
   while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
   {
    putc(c,f2);
    c=getc(f1);
   }
   putc(' ',f2);
   ungetc(c,f1);
  }
  else
  if(c==' '||c=='\t')
  printf(" ");
  else
  if(c=='\n')
  lineno++;
  else
  putc(c,f3);
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\n the no's in the program are:");
for(j=0;j<i;j++)
  printf("\t%d",num[j]);
printf("\n");
f2=fopen("identifier","r");
k=0;
printf("the keywords and identifier are:");
while((c=getc(f2))!=EOF)
if(c!=' ')
str[k++]=c;
else
{
  str[k]='\0';
  keyword(str);
  k=0;
}
fclose(f2);
f3=fopen("specialchar","r");
printf("\n Special Characters are");
while((c=getc(f3))!=EOF)
printf("\t%c",c);
printf("\n");
fclose(f3);
printf("Total no of lines are:%d",lineno);
}
```

OUTPUT:-



```
l2sys29@l2sys29-Veriton-M275: ~/Desktop/syedvirus
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ gcc exp2_lexana.c
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ ./a.out

 the no's in the program are:   3       2       5
the keywords and identifier are:
int is a keyword
a is an identifier
t1 is an identifier
t2 is an identifier
if is a keyword
printf is a keyword
n is an identifier
else is a keyword
char is a keyword
t3 is an identifier
c is an identifier
 Special Characters are {       [       ]       ,       ,       ;       (       >
)       (       "       \       "       )       ;       =       ;       }
Total no of lines are:8
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ 
```

PRACTICAL –2

Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines.

SOURCE CODE:-

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
void keyword(char str[10])
{
if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)
==0||
strcmp("static",str)==0||strcmp("switch",str)==0||strcmp("case",str)==0)
printf("\n%s is a keyword",str);
else
printf("\n%s is an identifier",str);
}
main()
{
FILE *f1,*f2,*f3;
char c,str[10],st1[10];
int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;
printf("\nEnter the c program");/*gets(st1);*/
f1=fopen("input","w");
while((c=getchar())!=EOF)
putc(c,f1);
fclose(f1);
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF){
if(isdigit(c))
{
tokenvalue=c-'0';
c=getc(f1);
while(isdigit(c)){
tokenvalue*=10+c-'0';
c=getc(f1);
}
num[i++]=tokenvalue;
ungetc(c,f1);
}
else if(isalpha(c))
{
putc(c,f2);
c=getc(f1);
while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
{
```

```
putc(c,f2);
c=getc(f1);
}
putc(' ',f2);
ungetc(c,f1);
}
else if(c==' '||c=='\t')
printf(" ");
else
if(c=='\n')
lineno++;
else
putc(c,f3);
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\nThe no's in the program are");
for(j=0;j<i;j++)
printf("%d",num[j]);
printf("\n");
f2=fopen("identifier","r");
k=0;
printf("The keywords and identifiersare:");
while((c=getc(f2))!=EOF){
if(c!=' ')
str[k++]=c;
else
{
str[k]='\0';
keyword(str);
k=0;
}
}
fclose(f2);
f3=fopen("specialchar","r");
printf("\nSpecial characters are");
while((c=getc(f3))!=EOF)
printf("%c",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d",lineno);
}
```

OUTPUT:-

Input:

```
Enter Program $ for termination:
{
int a[3],t1,t2;
t1=2; a[0]=1; a[1]=2; a[t1]=3;
t2=-(a[2]+t1*6)/(a[2]-t1);
if t2>5 then
print(t2);
else {
int t3;
t3=99;
t2=-25;
print(-t1+t2*t3); /* this is a comment on 2 lines */
} endif
}
(cntrl+z)
```

Output:

Variables : a[3] t1 t2 t3
Operator : - + * / >
Constants : 2 1 3 6 5 99 -25
Keywords : int if then else endif
Special Symbols : , ; ( ) { }
Comments : this is a comment on 2 lines

PRACTICAL – 3

Write a C program to test whether a given identifier is valid or not.

SOURCE CODE: -

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
char a[10];
int flag, i=1;
clrscr();
printf("\n Enter an identifier:");
gets(a);
if(isalpha(a[0]))
flag=1;
else
printf("\n Not a valid identifier");
while(a[i]!='\0')
{
if(!isdigit(a[i])&&!isalpha(a[i]))
{
flag=0;
break;
}
i++;
}
if(flag==1)
printf("\n Valid identifier");
getch();
}
```

OUTPUT :-

```
Enter an identifier: first
Valid identifier
Enter an identifier:1aqw
Not a valid identifier
```

PRACTICAL – 4

Implementation of recursive descent parser.

SOURCE CODE :-

recursive.c

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
char input[10];
int i,error;
void E();
void T();
void Eprime();
void Tprime();
void F();
        main()
        {
i=0;
error=0;
              printf("Enter an arithmetic expression   :  "); // Eg: a+a*a
              gets(input);
              E();
              if(strlen(input)==i&&error==0)
                     printf("\nAccepted..!!!\n");
              else printf("\nRejected..!!!\n");
                        }
void E()
{
    T();
    Eprime();
```

```
}
void Eprime()
{
     if(input[i]=='+')
     {
     i++;
     T();
     Eprime();
     }
     }
void T()
{
     F();
     Tprime();
}
void Tprime()
{
     if(input[i]=='*')
     {
                    i++;
                     F();
                     Tprime();
                     }
                     }
     void F()
     {
          if(isalnum(input[i]))i++;
          else if(input[i]=='(')
          {
```

```
i++;
E();
if(input[i]==')')
i++;

else error=1;
  }

else error=1;
}
```

OUTPUT :-

a+(a*a) a+a*a , (a), a , a+a+a*a+a.... etc are accepted

++a, a**a, +a, a, ((a . . . etc are rejected.

PRACTICAL – 5

Implementation of Lexical Analyzer using Lex.

SOURCE CODE :-

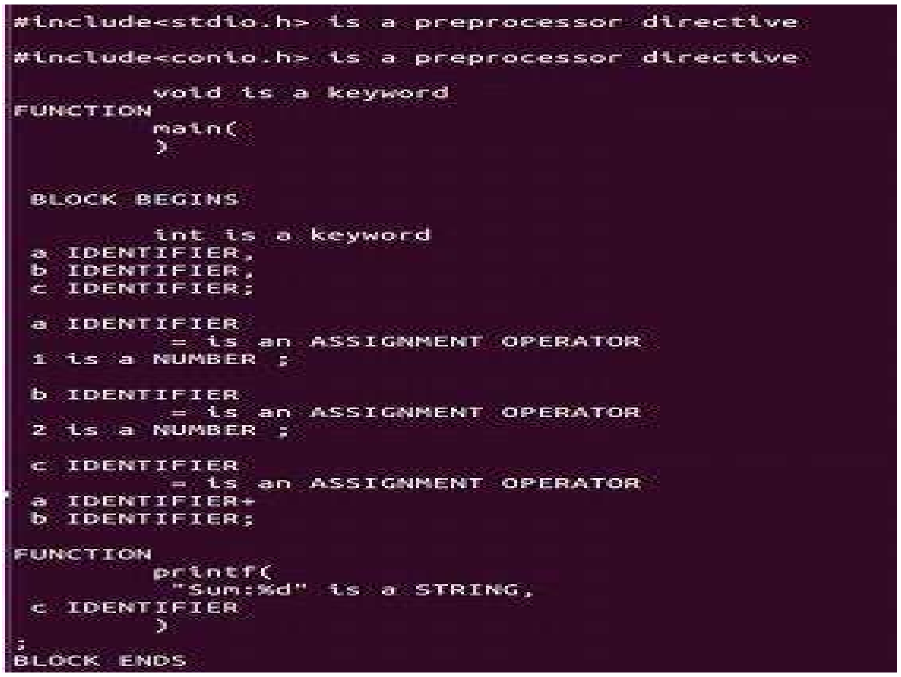
```
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;}{printf("\n\t  %s is a COMMENT",yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION  \n\t%s",yytext);}
\{  {if(!COMMENT)printf("\n  BLOCK BEGINS");}
\}  {if(!COMMENT)printf("BLOCK  ENDS ");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t  %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t  %s is an ASSIGNMENT
OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
```

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

INPUT:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

OUTPUT :-

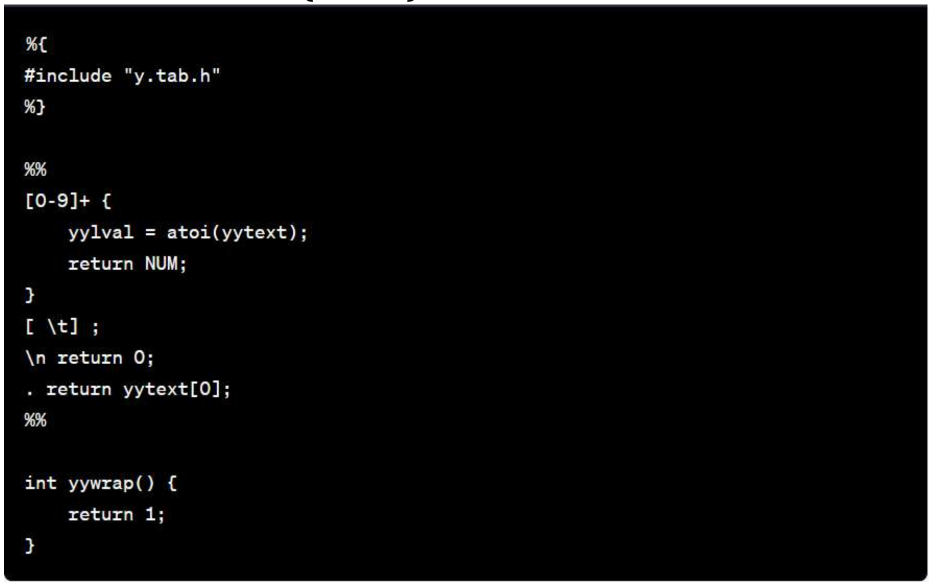


```
#include<stdio.h> is a preprocessor directive

#include<conio.h> is a preprocessor directive

        void is a keyword
FUNCTION
        main(
        )

 BLOCK BEGINS

        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

 a IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 1 is a NUMBER ;

 b IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 2 is a NUMBER ;

 c IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 a IDENTIFIER+
 b IDENTIFIER;

FUNCTION
        printf(
         "Sum:%d" is a STRING,
 c IDENTIFIER
        )
;
BLOCK ENDS
```

PRACTICAL-6

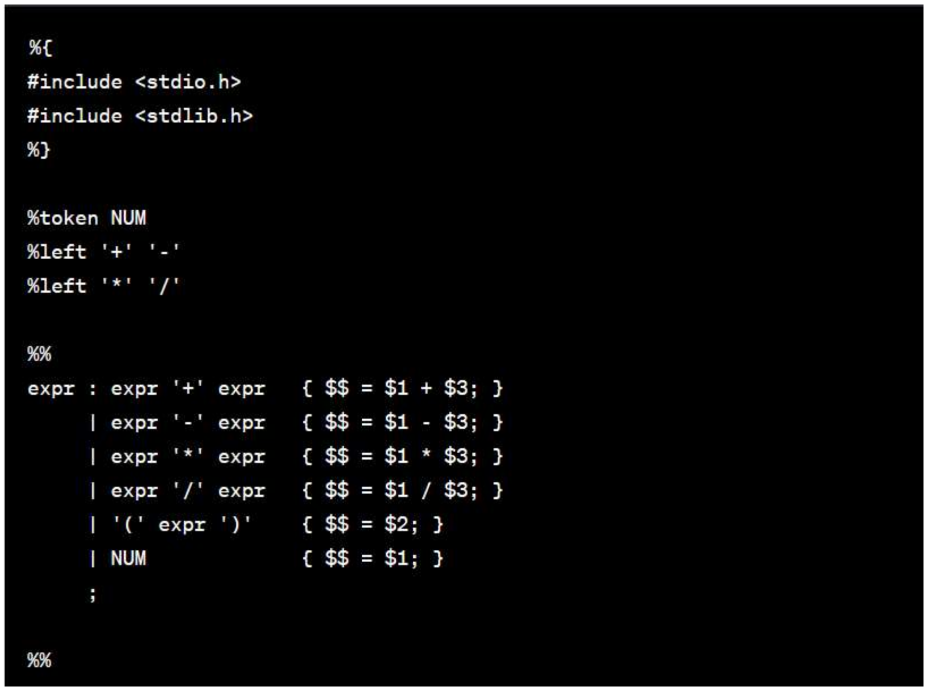
Creating a parser for an expression grammar using Lex and Yacc (or Bison) involves several steps. Here's a simplified example of how you can implement a parser for a basic arithmetic expression grammar in C using Lex and Yacc. In this example, we'll parse expressions like "2 + 3 * (4 - 1)".

- **Write the Lex file (lexer.l):**



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

%%
[0-9]+ {
    yylval = atoi(yytext);
    return NUM;
}
[ \t] ;
\n return 0;
. return yytext[0];
%%

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

- **Write the Yacc file (parser.y):**



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

%token NUM
%left '+' '-'
%left '*' '/'

%%
expr : expr '+' expr   { $$ = $1 + $3; }
     | expr '-' expr   { $$ = $1 - $3; }
     | expr '*' expr   { $$ = $1 * $3; }
     | expr '/' expr   { $$ = $1 / $3; }
     | '(' expr ')'    { $$ = $2; }
     | NUM             { $$ = $1; }
     ;


%%
```

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

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

- **Generate the lexer and parser source files using Lex and Yacc:**

```
lex lexer.l
yacc -d parser.y
```

- **This will create "lex.yy.c," "y.tab.c," and "y.tab.h" files.**
- **Compile the code:**

```
gcc lex.yy.c y.tab.c -o parser
```

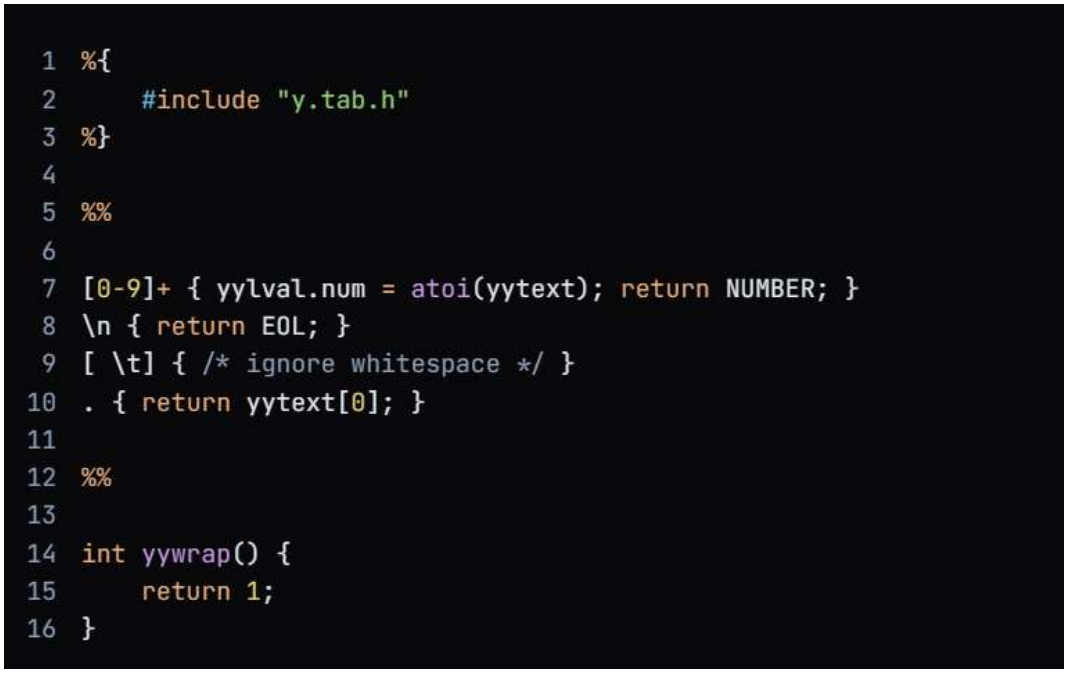
- **Now you can run the parser with input expressions:**

```
./parser
2 + 3 * (4 - 1)
```

PRACTICAL-7

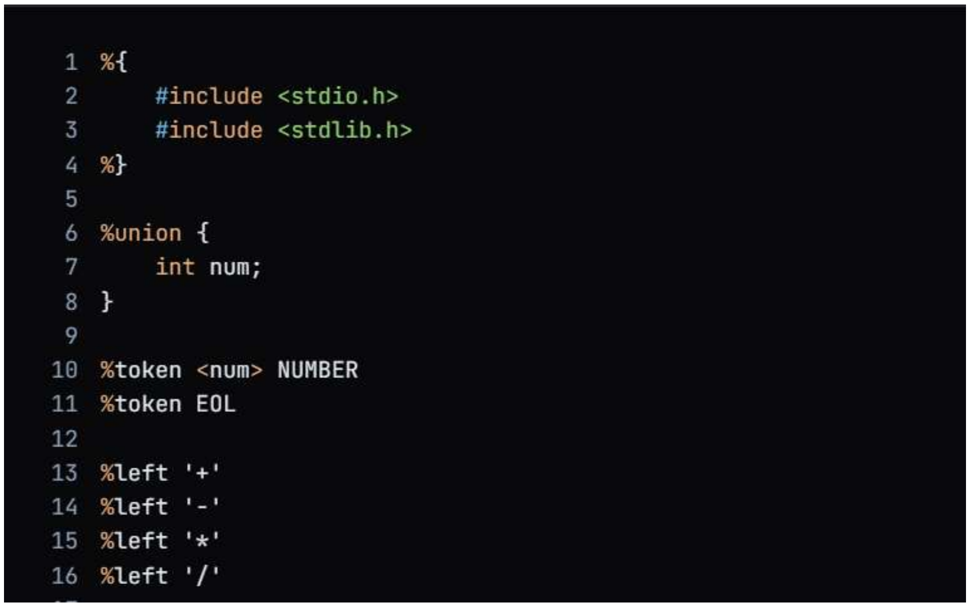
- **Generate three address codes for a simple program using LEX and YACC**

- **The lexer (lex) file (calculator.l) contains:**

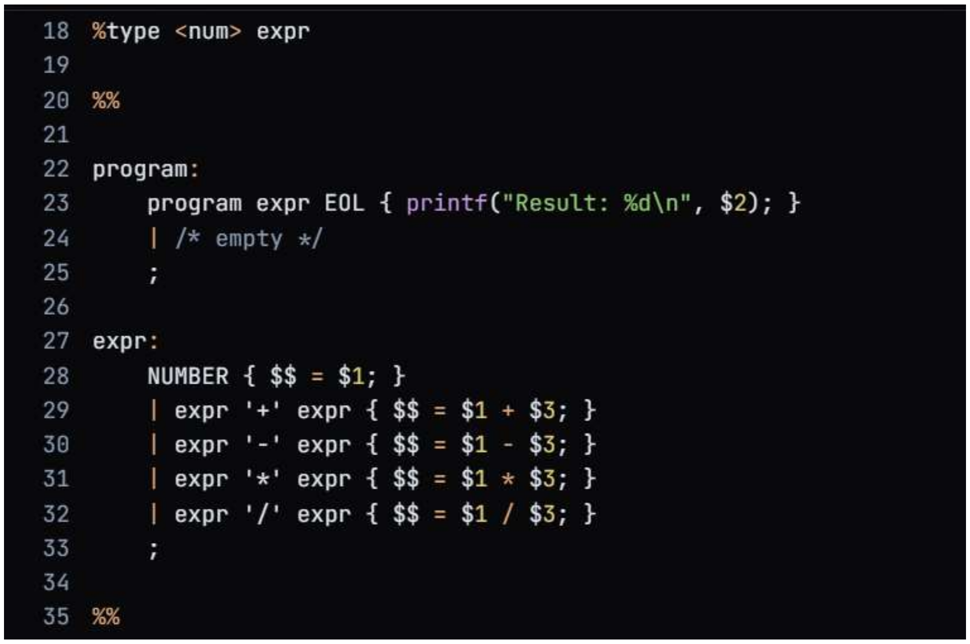


```
%{
    #include "y.tab.h"
%}

%%

[0-9]+ { yylval.num = atoi(yytext); return NUMBER; }
\n { return EOL; }
[ \t] { /* ignore whitespace */ }
. { return yytext[0]; }

%%

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

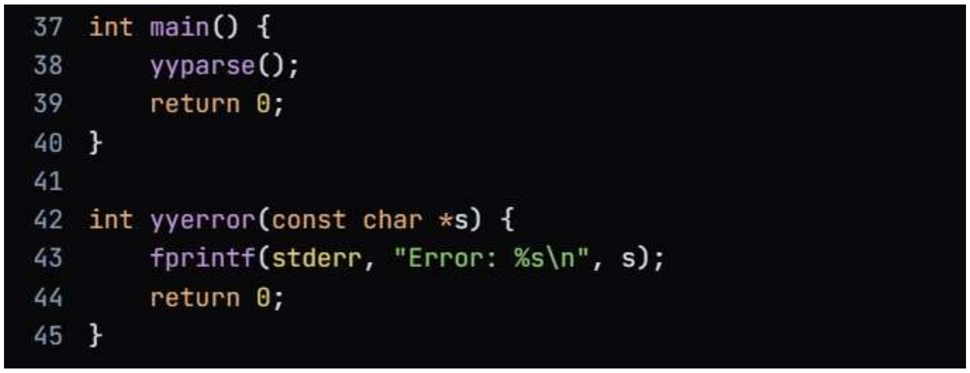
- **The parser (yacc) file (calculator.y) contains:**



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

%union {
    int num;
}

%token <num> NUMBER
%token EOL

%left '+'
%left '-'
%left '*'
%left '/'
```

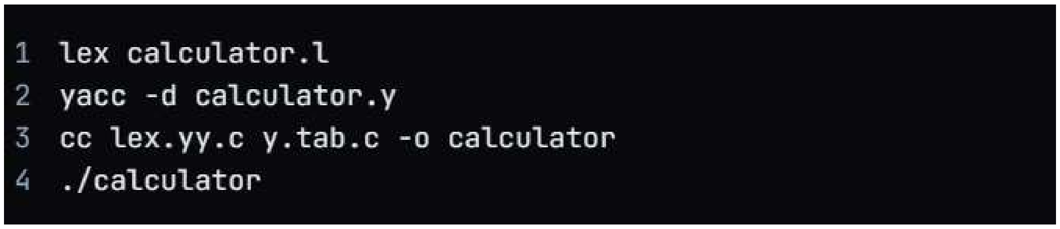


```
%type <num> expr

%%

program:
    program expr EOL { printf("Result: %d\n", $2); }
    | /* empty */
    ;

expr:
    NUMBER { $$ = $1; }
    | expr '+' expr { $$ = $1 + $3; }
    | expr '-' expr { $$ = $1 - $3; }
    | expr '*' expr { $$ = $1 * $3; }
    | expr '/' expr { $$ = $1 / $3; }
    ;

%%
```



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

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

- **To compile and run the program, execute the following commands:**



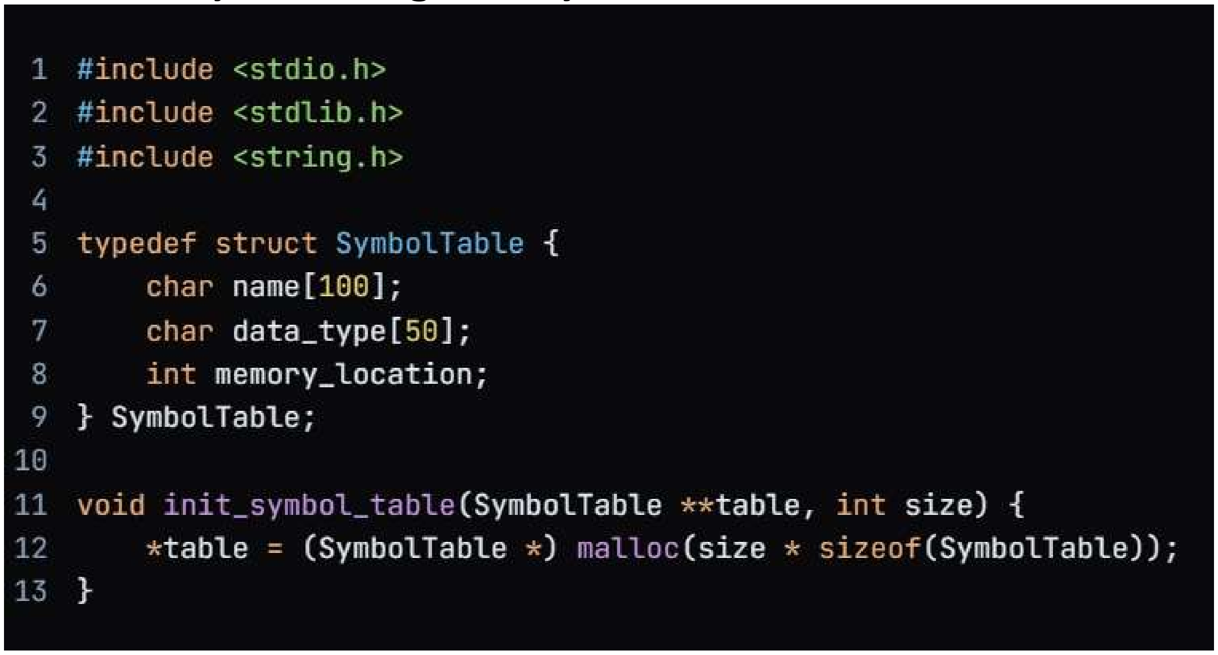
```
lex calculator.l
yacc -d calculator.y
cc lex.yy.c y.tab.c -o calculator
./calculator
```

PRACTICAL-8

Generate and populate appropriate symbol table.

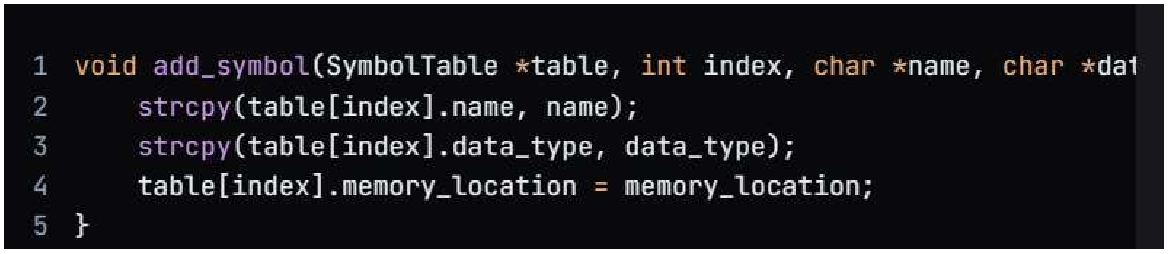
1. Use LEX (lexical analyzer) to parse the input file and identify symbols (identifiers) and their corresponding data types.
2. Define a data structure for the symbol table. For simplicity, you can use a hash table (dictionary) where the keys are the symbol names and the values are a structure that contains the symbol's data type and memory location (address).
3. When a new symbol is found, add an entry to the symbol table. In the case of an identifier that is a variable, also allocate memory for the variable.
4. To handle scope and shadowing, maintain a stack of scopes. Each scope can be represented as a hash table or dictionary. When entering a new scope, push a new scope onto the stack. When exiting a scope, pop the top scope from the stack.
5. If a symbol is already defined in the current scope, raise an error (shadowing). Otherwise, if the symbol is already defined in an outer scope, add an entry to the current scope with the same memory location.

- **Start by initializing a new symbol table:**



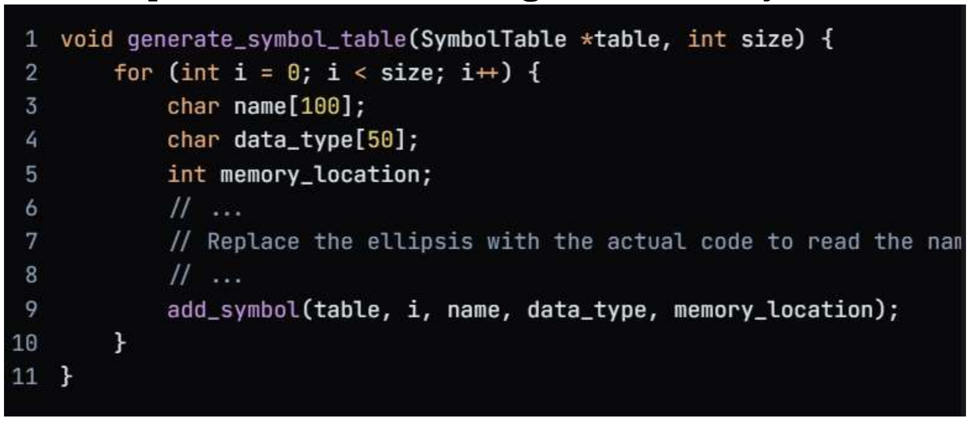
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct SymbolTable {
    char name[100];
    char data_type[50];
    int memory_location;
} SymbolTable;

void init_symbol_table(SymbolTable **table, int size) {
    *table = (SymbolTable *) malloc(size * sizeof(SymbolTable));
}
```

- **Define a function to add symbols to the symbol table:**



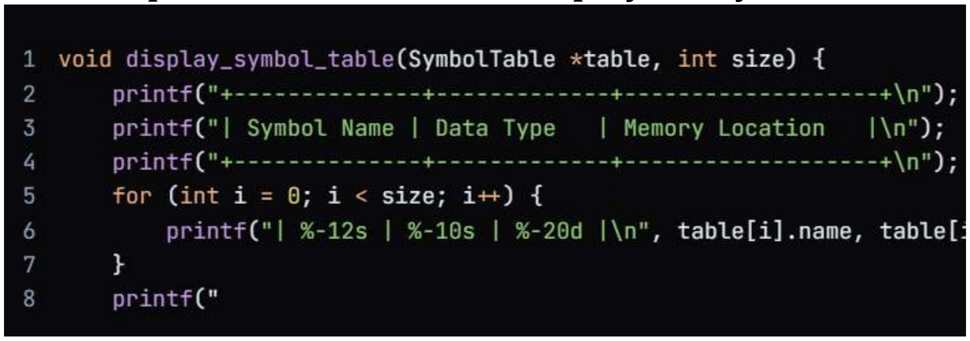
```c
void add_symbol(SymbolTable *table, int index, char *name, char *dat
    strcpy(table[index].name, name);
    strcpy(table[index].data_type, data_type);
    table[index].memory_location = memory_location;
}
```

- **Implement a function to generate the symbol table:**



```c
void generate_symbol_table(SymbolTable *table, int size) {
    for (int i = 0; i < size; i++) {
        char name[100];
        char data_type[50];
        int memory_location;
        // ...
        // Replace the ellipsis with the actual code to read the nam
        // ...
        add_symbol(table, i, name, data_type, memory_location);
    }
}
```

- **Implement a function to display the symbol table:**



```c
void display_symbol_table(SymbolTable *table, int size) {
    printf("+--------------+------------+-------------------+\n");
    printf("| Symbol Name | Data Type   | Memory Location   |\n");
    printf("+--------------+------------+-------------------+\n");
    for (int i = 0; i < size; i++) {
        printf("| %-12s | %-10s | %-20d |\n", table[i].name, table[
    }
    printf("
```

PRACTICAL-9

- **implementation of simple code optimization techniques (Constant folding, Strength reduction and Algebraic tranformation)**

1. Constant Folding

In the provided code snippet, constant folding can be performed on the arithmetic expressions involving constants. This process reduces the number of instructions executed by the CPU, resulting in optimized code.

Here is an example of constant folding applied to the arithmetic expression:

```
int main() {
    int a = 10, b = 20, c;
  c = a + b; // The expression a + b can be
folded into the constant value 30.
  printf("c = %d\n", c);
 return 0;
}
```

After applying constant folding, the optimized code becomes:

```
int main() {
    int a = 10, b = 20, c;
    c = 30; // The expression a + b has been
folded into the constant value 30.
    printf("c = %d\n", c);
   return 0;
}
```

2. Strength Reduction

In the provided code snippet, strength reduction can be applied to the arithmetic expressions involving constants. This process simplifies the arithmetic expression, resulting in optimized code.

Here is an example of strength reduction applied to the arithmetic expression:

```
int main() {
    int a = 10, b = 20, c;
    c = (a + b) * 2; // The expression (a + b) * 2 can be strength reduced into a * 2 + b * 2.
    printf("c = %d\n", c);
    return 0;
}
```

After applying strength reduction, the optimized code becomes:

```
int main() {
int a = 10, b = 20, c;
    c = a * 2 + b * 2; // The expression (a + b) * 2 has been strength reduced into a * 2 + b * 2.
    printf("c = %d\n", c);
    return 0;
}
```

3. Algebraic Transformation

In the provided code snippet, algebraic transformation can be applied to the arithmetic expressions involving constants. This process simplifies the arithmetic expression, resulting

```
#include <stdio.h>

 // Function prototypes

 double add(double a, double b);

 double subtract(double a, double b);
double multiply(double a, double b);
double divide(double a, double b);
int main() {
double a = 10.0, b = 20.0;
printf("a + b = %.2f\n", add(a, b));
printf("a - b = %.2f\n", subtract(a, b));
printf("a * b = %.2f\n", multiply(a, b));
printf("a / b = %.2f\n", divide(a, b));
return 0;
}
// Function implementations
double add(double a, double b) {
 return a + b;
}
double subtract(double a, double b) {
 return a - b;
}
double multiply(double a, double b) {
return a * b;
}
```

```c
double divide(double a, double b) {
 if (b == 0) {
 printf("Error: Division by zero is not allowed.\n");
 return 0;
 }
return a / b;
}
```

PRACTICAL-10

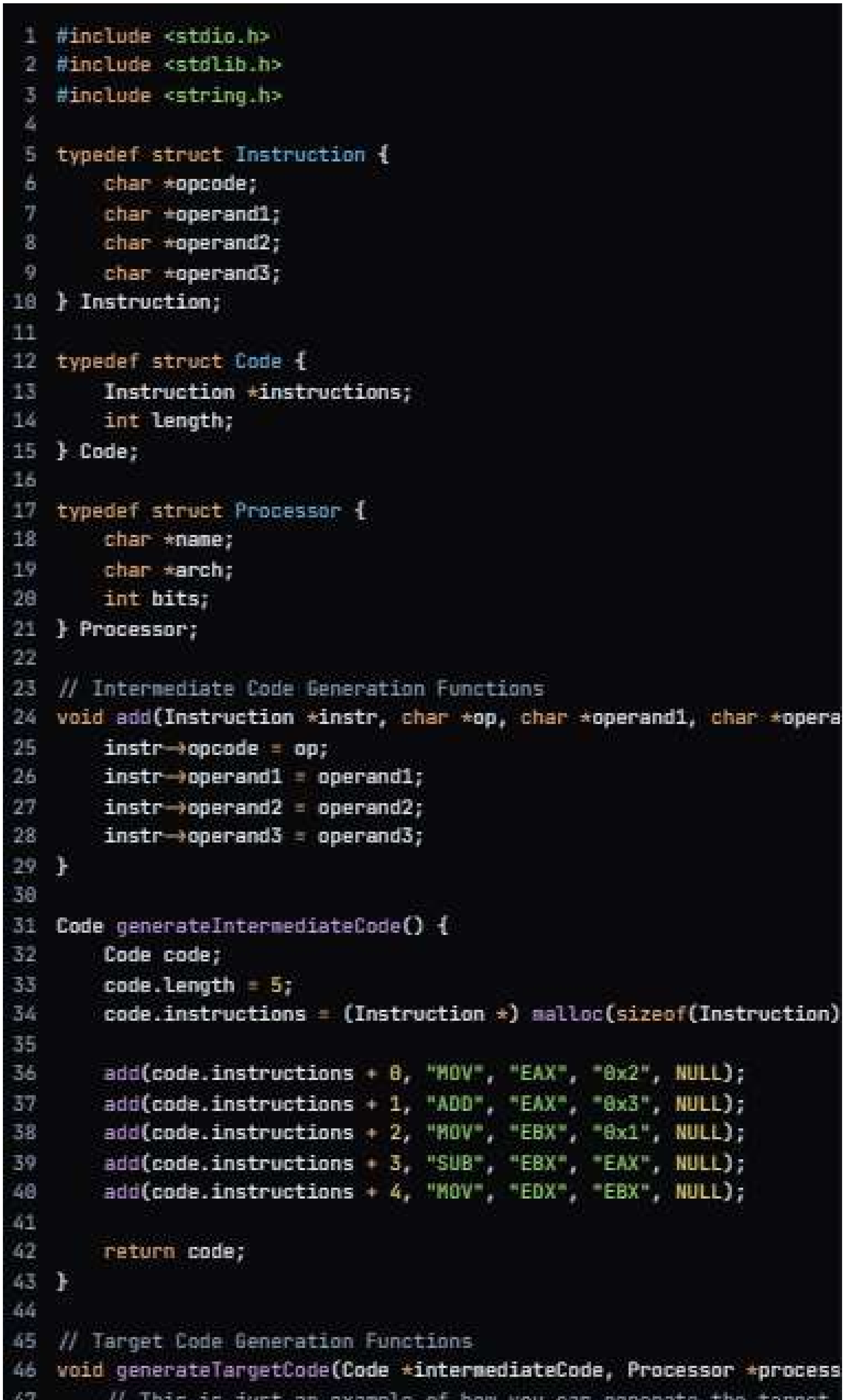
- **Generate an appropriate target code from the given intermidiate code assuming suitable processor details**

Generating target code from intermediate code in C requires you to map intermediate code operations to the specific assembly language instructions for your target architecture. Here's an example where we assume a simplified architecture with registers, and I'll map intermediate code operations to C code that resembles assembly instructions.

Let's consider the following intermediate code instructions:

- **`LOAD R1, 10`** (Load the value 10 into register R1)
- **`LOAD R2, 20`** (Load the value 20 into register R2)
- **`ADD R3, R1, R2`** (Add the values in R1 and R2 and store the result in R3)
- **`STORE R3, result`** (Store the value in R3 in the memory location labeled as "result")

Here's how you can generate C code that resembles target assembly instructions for this intermediate code:



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

typedef struct Instruction {
    char *opcode;
    char *operand1;
    char *operand2;
    char *operand3;
} Instruction;

typedef struct Code {
    Instruction *instructions;
    int length;
} Code;

typedef struct Processor {
    char *name;
    char *arch;
    int bits;
} Processor;

// Intermediate Code Generation Functions
void add(Instruction *instr, char *op, char *operand1, char *opera
    instr->opcode = op;
    instr->operand1 = operand1;
    instr->operand2 = operand2;
    instr->operand3 = operand3;
}

Code generateIntermediateCode() {
    Code code;
    code.length = 5;
    code.instructions = (Instruction *) malloc(sizeof(Instruction)

    add(code.instructions + 0, "MOV", "EAX", "0x2", NULL);
    add(code.instructions + 1, "ADD", "EAX", "0x3", NULL);
    add(code.instructions + 2, "MOV", "EBX", "0x1", NULL);
    add(code.instructions + 3, "SUB", "EBX", "EAX", NULL);
    add(code.instructions + 4, "MOV", "EDX", "EBX", NULL);

    return code;
}

// Target Code Generation Functions
void generateTargetCode(Code *intermediateCode, Processor *process
     // This is just an example of how you can generate the target
```