Report on

Mini-C Compiler

*Submitted in partial fulfilment of the requirements for **Sem VI***

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Chethan M	**PES2201800331**
Anirudh R	**PES2201800068**
Kaustub S	**PES2201800061**

Under the guidance of

Ghambire Swati S
Assistant Professor
PES University, Bengaluru

January – May 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

INTRODUCTION

This project being a Mini Compiler for the C language, focuses on generating an intermediate code for the language of specific constructs. It works for basic statements, conditional statements, loops etc.

The main functionality of the project is to generate an optimized intermediate code for the given C source code. This is done using the following steps:

1. Generate symbol table after performing expression evaluation.
2. Generate abstract syntax tree for the code.
3. Generate 3 address code.
4. Perform code optimization.

The main tools used in the project include LEX which identifies pre-defined patterns and generates tokens for the patters matched and YACC which parses the input for semantic meaning an generates an abstract syntax tree and intermediate code for the source code.

ARCHITECTURE OF LANGUAGE

C constructs implemented:

1. Basic C commands.
2. Conditional statements like if, if-else etc.
3. Loop statements like while, for, do-while etc.

- Arithmetic expressions with +, -, *, /, ++, -- etc. are handled.
- Boolean expressions with >, <, >=, <=, ==, != are handled.
- Error handling reports the type of error.
- Error handling also reports the line number where the error occurred.

DESIGN STAGES AND IMPLEMENTATION

PHASE-1

1. Lexical Analysis

- The scanner scans for comments and writes the source file without comments onto an output file which is used in the further stages.
- Skipping over white spaces and recognizing all keywords, operators, variables and constants are handled in this phase.

Code for lexical analysis:

```
%{

#include <stdlib.h>
#include <stdio.h>
#include "symboltable.h"
#include "tokens.h"

entry_t** symbol_table;
entry_t** constant_table;
int cmnt_strt = 0;

%}

letter [a-zA-Z]
digit [0-9]
ws  [ \t\r\f\v]+
identifier (_|{letter})({letter}|{digit}|_){0,31}
hex [0-9a-f]

 /* Exclusive states */
%x CMNT
%x PREPROC

%%
  /* Keywords*/
"int"                   {printf("\t%-30s : %3d\n",yytext,INT);}
"long"                   {printf("\t%-30s : %3d\n",yytext,LONG);}
"long long"                 {printf("\t%-30s : %3d\n",yytext,LONG_LONG);}
"short"                  {printf("\t%-30s : %3d\n",yytext,SHORT);}
"signed"                   {printf("\t%-30s : %3d\n",yytext,SIGNED);}
```

```
"unsigned"                  {printf("\t%-30s : %3d\n",yytext,UNSIGNED);}
"for"                       {printf("\t%-30s : %3d\n",yytext,FOR);}
"break"                     {printf("\t%-30s : %3d\n",yytext,BREAK);}
"continue"                  {printf("\t%-30s : %3d\n",yytext,CONTINUE);}
"if"                        {printf("\t%-30s : %3d\n",yytext,IF);}
"else"                      {printf("\t%-30s : %3d\n",yytext,ELSE);}
"return"                    {printf("\t%-30s : %3d\n",yytext,RETURN);}

{identifier}                {printf("\t%-30s : %3d\n", yytext,IDENTIFIER);
                            insert( symbol_table,yytext,IDENTIFIER );}
{ws}                        ;
[+\-]?[0][x|X]{hex}+[lLuU]?     {printf("\t%-30s : %3d\n", yytext,HEX_CONSTANT);
                                                        insert(
constant_table,yytext,HEX_CONSTANT);}
[+\-]?{digit}+[lLuU]?           {printf("\t%-30s : %3d\n", yytext,DEC_CONSTANT);
                                                        insert(
constant_table,yytext,DEC_CONSTANT);}
"/*"                        {cmnt_strt = yylineno; BEGIN CMNT;}
<CMNT>.|{ws}                ;
<CMNT>\n                    {yylineno++;}
<CMNT>"*/"                  {BEGIN INITIAL;}
<CMNT>"/*"                  {printf("Line %3d: Nested comments are not
valid!\n",yylineno);}
<CMNT><<EOF>>               {printf("Line %3d: Unterminated comment\n", cmnt_strt);
yyterminate();}
^"#include"                 {BEGIN PREPROC;}
<PREPROC>"<"[^<>\n]+">"         {printf("\t%-30s : %3d\n",yytext,HEADER_FILE);}
<PREPROC>{ws}               ;
<PREPROC>\"[^"\n]+\"            {printf("\t%-30s : %3d\n",yytext,HEADER_FILE);}
<PREPROC>\n                 {yylineno++; BEGIN INITIAL;}
<PREPROC>.                  {printf("Line %3d: Illegal header file format \n",yylineno);}
"//".*                      ;

\"[^\"\n]*\"    {

  if(yytext[yyleng-2]=='\\') /* check if it was an escaped quote */
  {
    yyless(yyleng-1);       /* push the quote back if it was escaped */
    yymore();
  }
  else
  insert( constant_table,yytext,STRING);
 }

\"[^\"\n]*$                 {printf("Line %3d: Unterminated string %s\n",yylineno,yytext);}
```

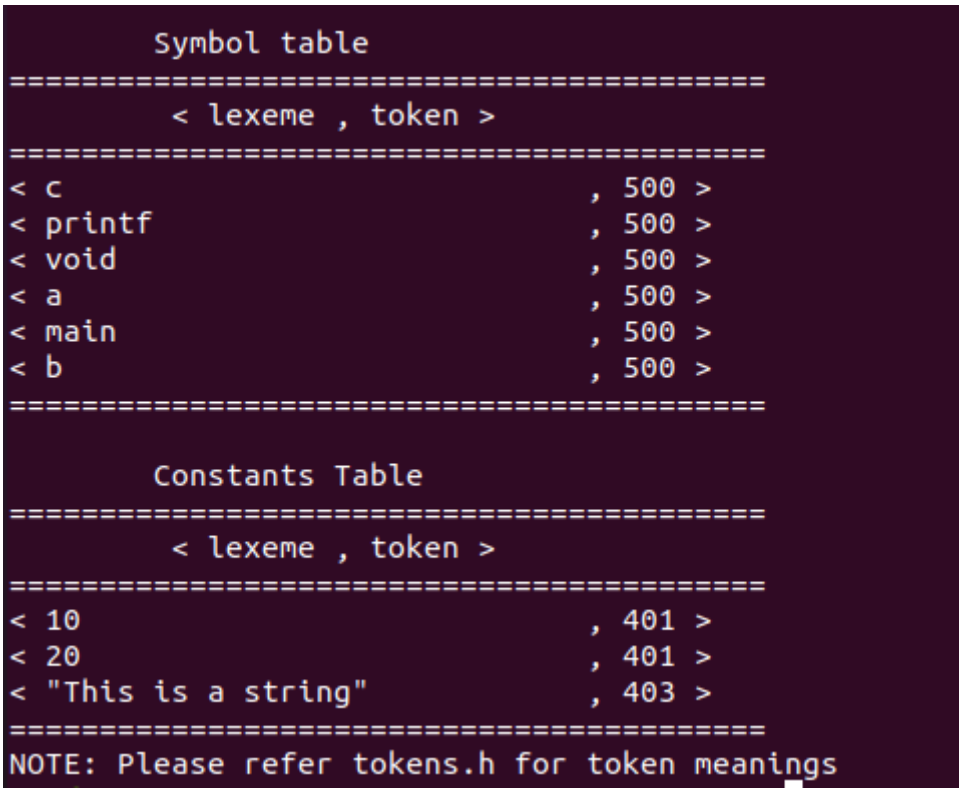
```
{digit}+({letter}|_)+        {printf("Line %3d: Illegal identifier name
%s\n",yylineno,yytext);}
\n                      {yylineno++;}
"--"                            {printf("\t%-30s : %3d\n",yytext,DECREMENT);}
"++"                            {printf("\t%-30s : %3d\n",yytext,INCREMENT);}
"->"                            {printf("\t%-30s : %3d\n",yytext,PTR_SELECT);}
"&&"                            {printf("\t%-30s : %3d\n",yytext,LOGICAL_AND);}
"||"                            {printf("\t%-30s : %3d\n",yytext,LOGICAL_OR);}
"<="                            {printf("\t%-30s : %3d\n",yytext,LS_THAN_EQ);}
">="                            {printf("\t%-30s : %3d\n",yytext,GR_THAN_EQ);}
"=="                            {printf("\t%-30s : %3d\n",yytext,EQ);}
"!="                        {printf("\t%-30s : %3d\n",yytext,NOT_EQ);}
";"                                 {printf("\t%-30s : %3d\n",yytext,DELIMITER);}
"{"                     {printf("\t%-30s : %3d\n",yytext,OPEN_BRACES);}
"}"                     {printf("\t%-30s : %3d\n",yytext,CLOSE_BRACES);}
","                             {printf("\t%-30s : %3d\n",yytext,COMMA);}
"="                             {printf("\t%-30s : %3d\n",yytext,ASSIGN);}
"("                             {printf("\t%-30s : %3d\n",yytext,OPEN_PAR);}
")"                             {printf("\t%-30s : %3d\n",yytext,CLOSE_PAR);}
"["                     {printf("\t%-30s : %3d\n",yytext,OPEN_SQ_BRKT);}
"]"                     {printf("\t%-30s : %3d\n",yytext,CLOSE_SQ_BRKT);}
"-"                             {printf("\t%-30s : %3d\n",yytext,MINUS);}
"+"                             {printf("\t%-30s : %3d\n",yytext,PLUS);}
"*"                             {printf("\t%-30s : %3d\n",yytext,STAR);}
"/"                         {printf("\t%-30s : %3d\n",yytext,FW_SLASH);}
"%"                             {printf("\t%-30s : %3d\n",yytext,MODULO);}
"<"                             {printf("\t%-30s : %3d\n",yytext,LS_THAN);}
">"                             {printf("\t%-30s : %3d\n",yytext,GR_THAN);}
.                       {printf("Line %3d: Illegal character %s\n",yylineno,yytext);}

%%

int main()
{
  yyin=fopen("test.c","r");
  symbol_table=create_table();
  constant_table=create_table();
  yylex();
  printf("\n\tSymbol table");
  display(symbol_table);
  printf("\n\tConstants Table");
  display(constant_table);
  printf("NOTE: Please refer tokens.h for token meanings\n");
}
```

Sample Input:

```
#include<stdio.h>
#include <<stdlib.h>
#include "custom.h"
#include ""wrong.h"

void main(){
        int a=10;
        int b=20;
        int c=a+b;
        printf("This is a string");
        printf("This is a string that never terminates);
}
```

Output:

```
seed@PES2201800331@server:~/CDproject/lex$ lex lexer.l
seed@PES2201800331@server:~/CDproject/lex$ cc lex.yy.c -ll
seed@PES2201800331@server:~/CDproject/lex$ ./a.out
        <stdio.h>                       : 402
Line   4: Illegal header file format
        <stdlib.h>                      : 402
        "custom.h"                      : 402
Line   6: Illegal header file format
        "wrong.h"                       : 402
        void                            : 500
        main                            : 500
        (                               : 304
        )                               : 305
        {                               : 301
        int                             : 100
        a                               : 500
        =                               : 209
        10                              : 401
        ;                               : 300
        int                             : 100
        b                               : 500
        =                               : 209
        20                              : 401
        ;                               : 300
        int                             : 100
        c                               : 500
        =                               : 209
        a                               : 500
        +                               : 211
        b                               : 500
        ;                               : 300
        printf                          : 500
        (                               : 304
        )                               : 305
        ;                               : 300
        printf                          : 500
        (                               : 304
Line  13: Unterminated string "This is a string that never terminates);
        }                               : 302
```



2. Syntax analysis

- Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your programming language grammar.
- YACC tool is used for parsing. It reports shift-reduce and reduce-reduce conflicts on parsing an ambiguous grammar.

Code for syntax analyser (CFG):

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

        entry_t** symbol_table;
  entry_t** constant_table;

        double Evaluate (double lhs_value,int assign_type,double rhs_value);
        int current_dtype;
```

```
	int yyerror(char *msg);
%}

%union
{
	double dval;
	entry_t* entry;
	int ival;
}

%token <entry> IDENTIFIER

 /* Constants */
%token <dval> DEC_CONSTANT HEX_CONSTANT
%token STRING

 /* Logical and Relational operators */
%token LOGICAL_AND LOGICAL_OR LS_EQ GR_EQ EQ NOT_EQ

 /* Short hand assignment operators */
%token MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN SUB_ASSIGN
%token LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN XOR_ASSIGN OR_ASSIGN
%token INCREMENT DECREMENT

 /* Data types */
%token SHORT INT LONG LONG_LONG SIGNED UNSIGNED CONST

 /* Keywords */
%token IF FOR WHILE CONTINUE BREAK RETURN

%type <dval> expression
%type <dval> sub_expr
%type <dval> constant
%type <dval> unary_expr
%type <dval> arithmetic_expr
%type <dval> assignment_expr
%type <entry> lhs
%type <ival> assign_op

%start starter

%left ','
%right '='
%left LOGICAL_OR
%left LOGICAL_AND
```

```
%left EQ NOT_EQ
%left '<' '>' LS_EQ GR_EQ
%left '+' '-'
%left '*' '/' '%'
%right '!'


%nonassoc UMINUS
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE


%%

 /* Program is made up of multiple builder blocks. */
starter: starter builder
                              |builder;

 /* Each builder block is either a function or a declaration */
builder: function|
       declaration;

 /* This is how a function looks like */
function: type IDENTIFIER '(' argument_list ')' compound_stmt;

 /* Now we will define a grammar for how types can be specified */

type :data_type pointer
    |data_type;

pointer: '*' pointer
    |'*'
    ;

data_type :sign_specifier type_specifier
    |type_specifier
    ;

sign_specifier :SIGNED
    |UNSIGNED
    ;

type_specifier :INT                 {current_dtype = INT;}
    |SHORT INT                      {current_dtype = SHORT;}
    |SHORT                         {current_dtype = SHORT;}
```

```
   |LONG                    {current_dtype = LONG;}
        |LONG INT                 {current_dtype = LONG;}
   |LONG_LONG                {current_dtype = LONG_LONG;}
   |LONG_LONG INT              {current_dtype = LONG_LONG;}
   ;

 /* grammar rules for argument list */
 /* argument list can be empty */
argument_list :arguments
   |
   ;
 /* arguments are comma separated TYPE ID pairs */
arguments :arguments ',' arg
   |arg
   ;

 /* Each arg is a TYPE ID pair */
arg :type IDENTIFIER
  ;

 /* Generic statement. Can be compound or a single statement */
stmt:compound_stmt
   |single_stmt
   ;

 /* The function body is covered in braces and has multiple statements. */
compound_stmt :'{' statements '}'
   ;

statements:statements stmt
   |
   ;

 /* Grammar for what constitutes every individual statement */
single_stmt :if_block
   |for_block
   |while_block
   |declaration
   |function_call ';'
        |RETURN ';'
        |CONTINUE ';'
        |BREAK ';'
        |RETURN sub_expr ';'
   ;
```

```
for_block:FOR '(' expression_stmt  expression_stmt ')' stmt
   |FOR '(' expression_stmt expression_stmt expression ')' stmt
   ;

if_block:IF '(' expression ')' stmt %prec LOWER_THAN_ELSE
                                |IF '(' expression ')' stmt ELSE stmt
   ;

while_block: WHILE '(' expression   ')' stmt
                ;

declaration:type declaration_list ';'
                        |declaration_list ';'
                        | unary_expr ';'

declaration_list: declaration_list ',' sub_decl
                |sub_decl;

sub_decl: assignment_expr
   |IDENTIFIER              {$1 -> data_type = current_dtype;}
   |array_index
   /*|struct_block ';'*/
   ;

/* This is because we can have empty expession statements inside for loops */
expression_stmt:expression ';'
   |';'
   ;

expression:
   expression ',' sub_expr                                                    {$$ =
$1,$3;}
   |sub_expr                                {$$ = $1;}
                ;

sub_expr:
   sub_expr '>' sub_expr                                                      {$$ = ($1 > $3);}
   |sub_expr '<' sub_expr                                                     {$$ = ($1 < $3);}
   |sub_expr EQ sub_expr                                                      {$$ = ($1 == $3);}
   |sub_expr NOT_EQ sub_expr              {$$ = ($1 != $3);}
   |sub_expr LS_EQ sub_expr               {$$ = ($1 <= $3);}
   |sub_expr GR_EQ sub_expr               {$$ = ($1 >= $3);}
        |sub_expr LOGICAL_AND sub_expr          {$$ = ($1 && $3);}
        |sub_expr LOGICAL_OR sub_expr           {$$ = ($1 || $3);}
        |'!' sub_expr                   {$$ = (!$2);}
```

```
        |arithmetic_expr                                                  {$$ = $1;}
   |assignment_expr                  {$$ = $1;}
        |unary_expr                     {$$ = $1;}
   /* |IDENTIFIER                        {$$ = $1->value;}
   |constant                       {$$ = $1;} */
                //|array_index
   ;


assignment_expr :lhs assign_op arithmetic_expr     {$$ = $1->value = Evaluate($1-
>value,$2,$3);}
   |lhs assign_op array_index              {$$ = 0;}
   |lhs assign_op function_call            {$$ = 0;}
        |lhs assign_op unary_expr               {$$ = $1->value = Evaluate($1-
>value,$2,$3);}
        |unary_expr assign_op unary_expr          {$$ = 0;}
   ;

unary_expr:    lhs INCREMENT                  {$$ = $1->value = ($1->value)++;}
        |lhs DECREMENT                      {$$ = $1->value = ($1->value)--;}
        |DECREMENT lhs                      {$$ = $2->value = --($2->value);}
        |INCREMENT lhs                     {$$ = $2->value = ++($2->value);}

lhs:IDENTIFIER                     {$$ = $1; if(! $1->data_type) $1->data_type =
current_dtype;}
   //|array_index
   ;

assign_op:'='                     {$$ = '=';}
   |ADD_ASSIGN                        {$$ = ADD_ASSIGN;}
   |SUB_ASSIGN                       {$$ = SUB_ASSIGN;}
   |MUL_ASSIGN                        {$$ = MUL_ASSIGN;}
   |DIV_ASSIGN                       {$$ = DIV_ASSIGN;}
   |MOD_ASSIGN                        {$$ = MOD_ASSIGN;}
   ;

arithmetic_expr: arithmetic_expr '+' arithmetic_expr    {$$ = $1 + $3;}
   |arithmetic_expr '-' arithmetic_expr          {$$ = $1 - $3;}
   |arithmetic_expr '*' arithmetic_expr          {$$ = $1 * $3;}
   |arithmetic_expr '/' arithmetic_expr          {$$ = ($3 == 0) ? yyerror("Divide by 0!") :
($1 / $3);}
        |arithmetic_expr '%' arithmetic_expr          {$$ = (int)$1 % (int)$3;}
        |'(' arithmetic_expr ')'                {$$ = $2;}
   |'-' arithmetic_expr %prec UMINUS            {$$ = -$2;}
   |IDENTIFIER                        {$$ = $1 -> value;}
```

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

constant: DEC_CONSTANT                          {$$ = $1;}
   |HEX_CONSTANT                              {$$ = $1;}
   ;

array_index: IDENTIFIER '[' sub_expr ']'

function_call: IDENTIFIER '(' parameter_list ')'
          |IDENTIFIER '(' ')'
          ;

parameter_list:
           parameter_list ','  parameter
           |parameter
           ;

parameter: sub_expr
                                        |STRING

      ;
%%

#include "lex.yy.c"
#include <ctype.h>


double Evaluate (double lhs_value,int assign_type,double rhs_value)
{
        switch(assign_type)
        {
                case '=': return rhs_value;
                case ADD_ASSIGN: return (lhs_value + rhs_value);
                case SUB_ASSIGN: return (lhs_value - rhs_value);
                case MUL_ASSIGN: return (lhs_value * rhs_value);
                case DIV_ASSIGN: return (lhs_value / rhs_value);
                case MOD_ASSIGN: return ((int)lhs_value % (int)rhs_value);
        }
}

int main(int argc, char *argv[])
{
        symbol_table = create_table();
        constant_table = create_table();
```

```
	yyin = fopen(argv[1], "r");

	if(!yyparse())
	{
		printf("\nParsing complete\n");
	}
	else
	{
			printf("\nParsing failed\n");
	}


	printf("\n\tSymbol table");
	display(symbol_table);


	fclose(yyin);
	return 0;
}

int yyerror(char *msg)
{
	printf("Line no: %d Error message: %s Token: %s\n", yylineno, msg, yytext);
}
```

Sample input:

```
int main(){

  int a = 10;
  int b = 10;

  for(a = 2 ;a<3; a++)
  b = b + 1;

  printf (" This string is enclosed in double quotes ");



  return 0;
}
```

Output:

```
seed@PES2201800331@server:~/CDproject/par$ lex lexl.l
seed@PES2201800331@server:~/CDproject/par$ yacc -d parser.y -v
seed@PES2201800331@server:~/CDproject/par$ gcc -w -g y.tab.c -ll -o parser
seed@PES2201800331@server:~/CDproject/par$ ./parser test.c

Parsing complete

        Symbol table
================================================
 lexeme               value               data-type
================================================
 printf               2147483647          0
 a                    2                   281
 main                 2147483647          0
 b                    11                  281
================================================
seed@PES2201800331@server:~/CDproject/par$
```

3. Semantic Analysis

At this phase, the semantics of the programs that's obtained as an output from syntax analysis will be checked.

Code for semantic analysis:

```
%{
        #include <stdlib.h>
        #include <stdio.h>
        int yyerror(char *msg);

        #include "symboltable.h"
        #include "lex.yy.c"

        #define SYMBOL_TABLE symbol_table_list[current_scope].symbol_table

 extern entry_t** constant_table;

        int current_dtype;

        table_t symbol_table_list[NUM_TABLES];

        int is_declaration = 0;
        int is_loop = 0;
```

```
        int is_func = 0;
        int func_type;

        int param_list[10];
        int p_idx = 0;
        int p=0;
   int rhs = 0;

        void type_check(int,int,int);
%}

%union
{
        int data_type;
        entry_t* entry;
}

%token <entry> IDENTIFIER

 /* Constants */
%token <entry> DEC_CONSTANT HEX_CONSTANT CHAR_CONSTANT
FLOAT_CONSTANT
%token STRING

 /* Logical and Relational operators */
%token LOGICAL_AND LOGICAL_OR LS_EQ GR_EQ EQ NOT_EQ

 /* Short hand assignment operators */
%token MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN SUB_ASSIGN
%token INCREMENT DECREMENT

 /* Data types */
%token SHORT INT LONG LONG_LONG SIGNED UNSIGNED CONST VOID CHAR
FLOAT

 /* Keywords */
%token IF FOR WHILE CONTINUE BREAK RETURN

%type <entry> identifier
%type <entry> constant
%type <entry> array_index

%type <data_type> sub_expr
%type <data_type> unary_expr
%type <data_type> arithmetic_expr
```

```
%type <data_type> assignment_expr
%type <data_type> function_call
%type <data_type> array_access
%type <data_type> lhs

%left ','
%right '='
%left LOGICAL_OR
%left LOGICAL_AND
%left EQ NOT_EQ
%left '<' '>' LS_EQ GR_EQ
%left '+' '-'
%left '*' '/' '%'
%right '!'


%nonassoc UMINUS
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE


%%

 /* Program is made up of multiple builder blocks. */
starter: starter builder
                        | builder;

 /* Each builder block is either a function or a declaration */
builder: function
                        | declaration
                        ;

 /* This is how a function looks like */
function: type
                                identifier
                                {

                                                                        func_type =
current_dtype;

                                                                        is_declaration =
0;

                                                                        current_scope =
create_new_scope();
```

```
                                                    }

                                '(' argument_list ')'                                   {

                                                                    is_declaration =
0;


        fill_parameter_list($2,param_list,p_idx);

                                                                    p_idx = 0;

                                                                    is_func = 1;

                                                                    p=1;

                                                    }

                                compound_stmt
                                {

                                                                    is_func = 0;

                                                    }
        ;
 /* Now we will define a grammar for how types can be specified */

type : data_type pointer
    {is_declaration = 1; }
    | data_type
    {is_declaration = 1; }
                ;

pointer: '*' pointer
          | '*'
     ;

data_type : sign_specifier type_specifier
                        | type_specifier
                        ;

sign_specifier : SIGNED
                                        | UNSIGNED
                                        ;
```

```
type_specifier :INT                 {current_dtype = INT;}
   |SHORT INT                      {current_dtype = SHORT;}
   |SHORT                         {current_dtype = SHORT;}
   |LONG                          {current_dtype = LONG;}
                |LONG INT                       {current_dtype = LONG;}
   |LONG_LONG                     {current_dtype = LONG_LONG;}
   |LONG_LONG INT                 {current_dtype = LONG_LONG;}
                |CHAR
                                               {current_dtype = CHAR;}
                |FLOAT
                                               {current_dtype = FLOAT;}
                |VOID
                                               {current_dtype = VOID;}
   ;

 /* grammar rules for argument list */
 /* argument list can be empty */
argument_list : arguments
                                             |
                                             ;
 /* arguments are comma separated TYPE ID pairs */
arguments : arguments ',' arg
                          | arg
                          ;

 /* Each arg is a TYPE ID pair */
arg : type identifier
        {param_list[p_idx++] = $2->data_type;}
   ;

 /* Generic statement. Can be compound or a single statement */
stmt:compound_stmt
   |single_stmt
   ;

 /* The function body is covered in braces and has multiple statements. */
compound_stmt :
                                                        '{'
                {

                                                  if(!p)current_scope = create_new_scope();

                                                  else p = 0;
```

```
                                    }

                                                        statements

                                                        '}'
        {current_scope = exit_scope();}
    ;

statements:statements stmt
    |
    ;

 /* Grammar for what constitutes every individual statement */
single_stmt :if_block
    |for_block
    |while_block
    |declaration
    |function_call ';'
                |RETURN ';'                                                          {

                                        if(is_func)

                                        {

                                                if(func_type != VOID)

                                                        yyerror("return type (VOID) does not
match function type");

                                        }

                                    else yyerror("return statement not inside function
definition");

                                    }

                |CONTINUE ';'
{if(!is_loop) {yyerror("Illegal use of continue");}}
                |BREAK ';'              {if(!is_loop) {yyerror("Illegal use of break");}}

                |RETURN sub_expr ';'                        {

                                        if(is_func)
```

```
                                        {
                                                if(func_type != $2)
                                                        yyerror("return type does not match
function type");
                                        }
                                        else yyerror("return statement not in function
definition");
                                }
   ;

for_block:FOR '(' expression_stmt  expression_stmt ')' {is_loop = 1;} stmt {is_loop = 0;}
                |FOR '(' expression_stmt expression_stmt expression ')' {is_loop = 1;} stmt
{is_loop = 0;}
                ;

if_block:IF '(' expression ')' stmt
         %prec LOWER_THAN_ELSE
                                |IF '(' expression ')' stmt ELSE stmt
   ;

while_block: WHILE '(' expression   ')' {is_loop = 1;} stmt {is_loop = 0;}
                ;

declaration: type  declaration_list ';'
        {is_declaration = 0; }
                                        | declaration_list ';'
                                        | unary_expr ';'


declaration_list: declaration_list ',' sub_decl
                                                                |sub_decl
                                                                ;

sub_decl: assignment_expr
                |identifier
                |array_access
                                ;

/* This is because we can have empty expession statements inside for loops */
```

```
expression_stmt: expression ';'
                                        | ';'
                                        ;

expression: expression ',' sub_expr
                        | sub_expr
                                    ;

sub_expr:
   sub_expr '>' sub_expr
                                            {type_check($1,$3,2); $$ = $1;}
   |sub_expr '<' sub_expr
                                        {type_check($1,$3,2); $$ = $1;}
   |sub_expr EQ sub_expr
                                                {type_check($1,$3,2); $$ = $1;}
   |sub_expr NOT_EQ sub_expr
                                        {type_check($1,$3,2); $$ = $1;}
   |sub_expr LS_EQ sub_expr
                                        {type_check($1,$3,2); $$ = $1;}
   |sub_expr GR_EQ sub_expr
                                        {type_check($1,$3,2); $$ = $1;}
            |sub_expr LOGICAL_AND sub_expr
                                        {type_check($1,$3,2); $$ = $1;}
            |sub_expr LOGICAL_OR sub_expr
                                        {type_check($1,$3,2); $$ = $1;}
            |'!' sub_expr
                                                                    {$$ = $2;}
            |arithmetic_expr
                                                                {$$ = $1;}
   |assignment_expr
                                                            {$$ = $1;}
            |unary_expr

        {$$ = $1;}
   ;


assignment_expr :
            lhs assign_op  arithmetic_expr
                                        {type_check($1,$3,1); $$ = $3; rhs=0;}
   |lhs assign_op  array_access
                                    {type_check($1,$3,1); $$ = $3;rhs=0;}
   |lhs assign_op  function_call
                                {type_check($1,$3,1); $$ = $3;rhs=0;}
```

```
        |lhs assign_op  unary_expr                                {type_check($1,$3,1); $$ =
$3;rhs=0;}
                |unary_expr assign_op  unary_expr
                        {type_check($1,$3,1); $$ = $3;rhs=0;}
   ;

unary_expr:    identifier INCREMENT
                                        {$$ = $1->data_type;}
                                        | identifier DECREMENT
                                                        {$$ = $1->data_type;}
                                        | DECREMENT identifier
                                                        {$$ = $2->data_type;}
                                        | INCREMENT identifier
                                                        {$$ = $2->data_type;}

lhs: identifier
                                                                {$$ = $1-
>data_type;}
  | array_access
                                                                {$$ = $1;}
          ;

identifier:IDENTIFIER                   {
                                        if(is_declaration
                                        && !rhs)
                                        {
                                           $1 =
insert(SYMBOL_TABLE,yytext,INT_MAX,current_dtype);
                                           if($1 == NULL) yyerror("Redeclaration of
variable");
                                        }
                                        else
                                        {
                                           $1 = search_recursive(yytext);
                                           if($1 == NULL) yyerror("Variable not
declared");
                                        }
                                        $$ = $1;
                                }
                        ;

assign_op:'=' {rhs=1;}
   |ADD_ASSIGN {rhs=1;}
   |SUB_ASSIGN {rhs=1;}
   |MUL_ASSIGN {rhs=1;}
```

```
   |DIV_ASSIGN {rhs=1;}
   |MOD_ASSIGN {rhs=1;}
   ;

arithmetic_expr: arithmetic_expr '+' arithmetic_expr
         {type_check($1,$3,0);}
   |arithmetic_expr '-' arithmetic_expr
                        {type_check($1,$3,0);}
   |arithmetic_expr '*' arithmetic_expr
                        {type_check($1,$3,0);}
   |arithmetic_expr '/' arithmetic_expr
                  {type_check($1,$3,0);}
                  |arithmetic_expr '%' arithmetic_expr
                              {type_check($1,$3,0);}
                  |'(' arithmetic_expr ')'
                                                            {$$ = $2;}
   |'-' arithmetic_expr %prec UMINUS
                                       {$$ = $2;}
   |identifier

         {$$ = $1->data_type;}
   |constant

         {$$ = $1->data_type;}
   ;

constant: DEC_CONSTANT
                        {$1->is_constant=1; $$ = $1;}
   | HEX_CONSTANT
                                       {$1->is_constant=1; $$ = $1;}
                  | CHAR_CONSTANT
                                                   {$1->is_constant=1; $$ = $1;}
                  | FLOAT_CONSTANT
                                             {$1->is_constant=1; $$ = $1;}
   ;

array_access: identifier '[' array_index ']'
         {


                                                   if(is_declaration)


                                                   {
```

```
                                        if($3->value <= 0)
                                            yyerror("size of array is
not positive");
                                        else
                                                if($3-
>is_constant && !rhs)
                                            $1->array_dimension =
$3->value;
                                            else if(rhs){
                                            {
                                        if($3->value > $1-
>array_dimension)
                                            yyerror("Array index out
of bound");
                                        if($3->value < 0)
                                            yyerror("Array index
cannot be negative");
                                    }
                                    }
```

```
                                                    }

                                                    else if($3->is_constant)

                                                    {

                                                            if($3->value > $1-
>array_dimension)

                                                                    yyerror("Array index out
of bound");

                                                            if($3->value < 0)

                                                                    yyerror("Array index
cannot be negative");

                                                    }

                                                    $$ = $1->data_type;

                                            }

array_index: constant
                                                            {$$ = $1;}
                                    | identifier
                                                                            {$$ = $1;}
                                    ;

function_call: identifier '(' parameter_list ')'                    {
```

```
                                        $$ = $1->data_type;

                                        check_parameter_list($1,param_list,p_idx);

                                        p_idx = 0;

                                    }

        | identifier '(' ')'
                                {

                                        $$ = $1->data_type;

                                        check_parameter_list($1,param_list,p_idx);

                                        p_idx = 0;

                                    }
        ;

parameter_list:
            parameter_list ','  parameter
            |parameter
            ;

parameter: sub_expr                                                     {param_list[p_idx++] =
$1;}
                                    | STRING

        {param_list[p_idx++] = STRING;}
                                    ;
%%

void type_check(int left, int right, int flag)
{
```

```
	if(left != right)
	{
		switch(flag)
		{
			case 0: yyerror("Type mismatch in arithmetic expression"); break;
			case 1: yyerror("Type mismatch in assignment expression"); break;
			case 2: yyerror("Type mismatch in logical expression"); break;
		}
	}
}

int main(int argc, char *argv[])
{
	 int i;
	 for(i=0; i<NUM_TABLES;i++)
	 {
	  symbol_table_list[i].symbol_table = NULL;
	  symbol_table_list[i].parent = -1;
	 }

	constant_table = create_table();
  symbol_table_list[0].symbol_table = create_table();
	yyin = fopen(argv[1], "r");

	if(!yyparse())
	{
		printf("\nPARSING COMPLETE\n\n\n");
	}
	else
	{
			printf("\nPARSING FAILED!\n\n\n");
	}


	printf("SYMBOL TABLES\n\n");
	display_all();

	printf("CONSTANT TABLE");
	display_constant_table(constant_table);


	fclose(yyin);
	return 0;
}
```

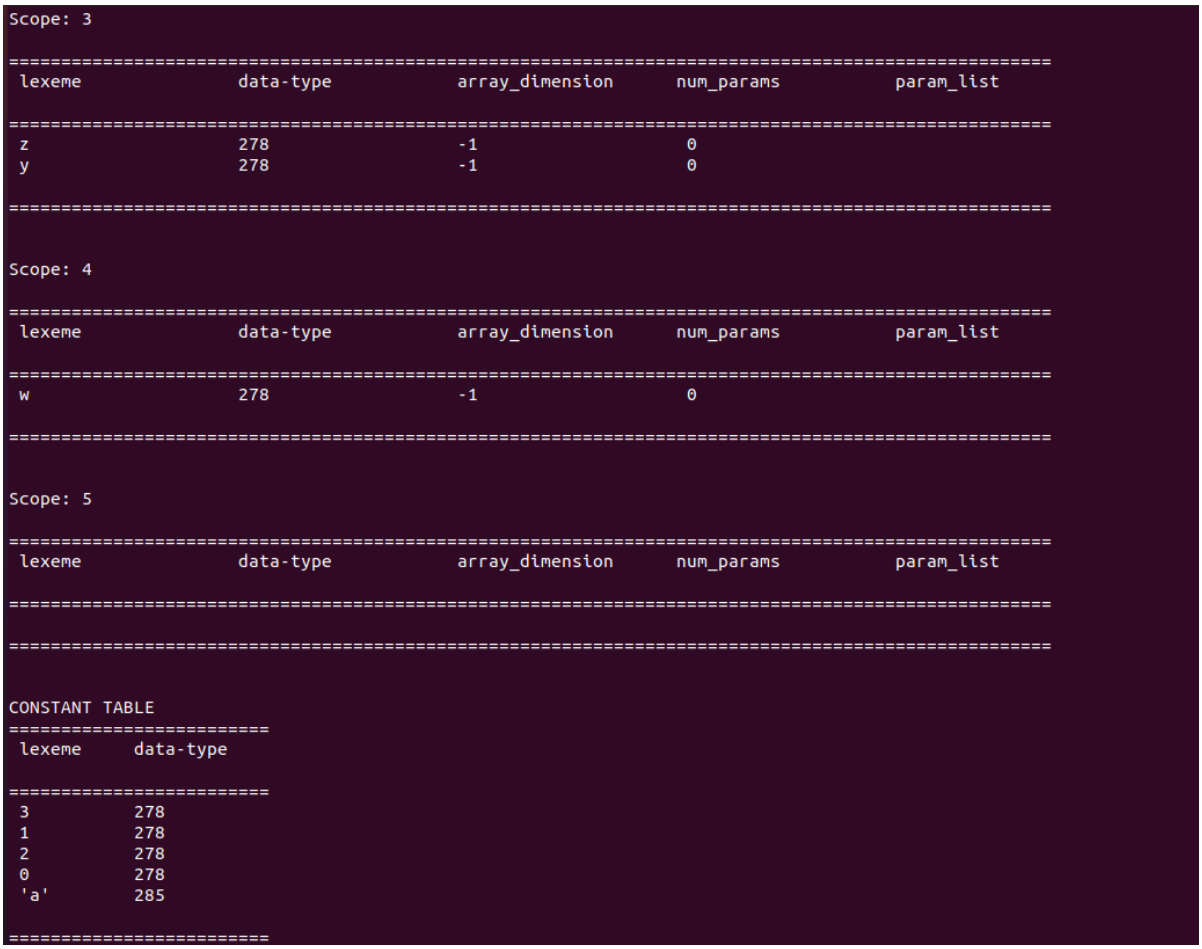
```
int yyerror(char *msg)
{
        printf("Line no: %d Error message: %s Token: %s\n", yylineno, msg, yytext);
        exit(0);
}
```

Sample input:

```
int fun(int a, int b, int c)
{
 return 2 ;
}

int main()
{
 int x = 1;

 int y;
 int z;

 if(x<0)
 {
  int y;

  int z=1;

  {
   int w =1;
  }

 }
 char c='a';
 for(x=2;x<y;x++)
 {
  break;
 }
 return 3;



}
```

Output:

```
seed@PES2201800331@server:~/CDproject/sem$ lex lexer.l
seed@PES2201800331@server:~/CDproject/sem$ yacc -d parser.y -v
seed@PES2201800331@server:~/CDproject/sem$ gcc -w -g y.tab.c -ll -o out
seed@PES2201800331@server:~/CDproject/sem$ ./out test.c

PARSING COMPLETE


SYMBOL TABLES

Scope: 0

================================================================================================
 lexeme               data-type            array_dimension      num_params           param_list

================================================================================================
 fun                  278                  -1                    3                    278 278 278
 main                 278                  -1                    0

================================================================================================


Scope: 1

================================================================================================
 lexeme               data-type            array_dimension      num_params           param_list

================================================================================================
 c                    278                  -1                    0
 a                    278                  -1                    0
 b                    278                  -1                    0

================================================================================================


Scope: 2

================================================================================================
 lexeme               data-type            array_dimension      num_params           param_list

================================================================================================
 c                    285                  -1                    0
 x                    278                  -1                    0
 z                    278                  -1                    0
 y                    278                  -1                    0
```



PHASE-2

1. Intermediate code generation

Intermediate code generator receives input from its predecessor phase semantic analyser, in the form of an annotated syntax tree. This syntax tree then can be converted into a linear representation. Intermediate code tends to be machine independent code.

Code for ICG:

```
%{
	#include <bits/stdc++.h>
	#include "symboltable.h"
```

```
	#include "lex.yy.c"

	using namespace std;

	int yyerror(char *msg);

	#define SYMBOL_TABLE symbol_table_list[current_scope].symbol_table

	extern entry_t** constant_table;

	int current_dtype;

	table_t symbol_table_list[NUM_TABLES];

	int is_declaration = 0;
	int is_loop = 0;
	int is_func = 0;
	int func_type;

	int param_list[10];
	int p_idx = 0;
	int p=0;
	int rhs = 0;

	void type_check(int,int,int);
	vector<int> merge(vector<int>& v1, vector<int>& v2);
	void backpatch(vector<int>&, int);
	void gencode(string);
	void gencode_math(content_t* & lhs, content_t* arg1, content_t* arg2, const string&
op);
	void gencode_rel(content_t* & lhs, content_t* arg1, content_t* arg2, const string&
op);
	void printlist(vector<int>);

	int nextinstr = 0;
	int temp_var_number = 0;

	vector<string> ICG;

%}

%union
{
	int data_type;
	entry_t* entry;
```

```
        content_t* content;
        string* op;
        vector<int>* nextlist;
        int instr;
}

%token <entry> IDENTIFIER

 /* Constants */
%token <entry> DEC_CONSTANT HEX_CONSTANT CHAR_CONSTANT
FLOAT_CONSTANT STRING

 /* Logical and Relational operators */
%token LOGICAL_AND LOGICAL_OR LS_EQ GR_EQ EQ NOT_EQ

 /* Short hand assignment operators */
%token MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN SUB_ASSIGN
%token INCREMENT DECREMENT

 /* Data types */
%token SHORT INT LONG LONG_LONG SIGNED UNSIGNED CONST VOID CHAR
FLOAT CHAR_STAR

 /* Keywords */
%token IF FOR WHILE CONTINUE BREAK RETURN

%type <entry> identifier
%type <entry> constant
%type <entry> array_index

%type <op> assign;
%type <data_type> function_call

%type <content> lhs
%type <content> sub_expr
%type <content> expression
%type <content> expression_stmt
%type <content> unary_expr
%type <content> arithmetic_expr
%type <content> assignment_expr
%type <content> array_access

%type <content> if_block
%type <content> for_block
%type <content> while_block
```

```
%type <content> compound_stmt

%type <content> statements
%type <content> single_stmt
%type <content> stmt


%type <instr> M
%type <content> N

%left ','
%right '='
%left LOGICAL_OR
%left LOGICAL_AND
%left EQ NOT_EQ
%left '<' '>' LS_EQ GR_EQ
%left '+' '-'
%left '*' '/' '%'
%right '!'


%nonassoc UMINUS
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE


%%

 /* Program is made up of multiple builder blocks. */
starter: starter builder
                        | builder;

 /* Each builder block is either a function or a declaration */
builder: function
                        | declaration
                        ;

 /* This is how a function looks like */
function: type identifier
                        {
                                func_type = current_dtype;
                                is_declaration = 0;
                                current_scope = create_new_scope();
                                gencode($2->lexeme + string(":"));
                        }
```

```
                '(' argument_list ')'
                        {
                                is_declaration = 0;
                                fill_parameter_list($2,param_list,p_idx);
                                p_idx = 0;
                                is_func = 1;
                                p=1;
                        }

                compound_stmt        {   is_func = 0;          }

               ;

 /* Now we will define a grammar for how types can be specified */

type : data_type pointer    {is_declaration = 1; }
    | data_type              {is_declaration = 1; }
                ;

pointer: '*' pointer
        | '*'
      ;

data_type : sign_specifier type_specifier
        | type_specifier
        ;

sign_specifier : SIGNED
                | UNSIGNED
                ;

type_specifier :INT              {current_dtype = INT;}
   |SHORT INT                   {current_dtype = SHORT;}
   |SHORT                      {current_dtype = SHORT;}
   |LONG                       {current_dtype = LONG;}
        |LONG INT                    {current_dtype = LONG;}
   |LONG_LONG                  {current_dtype = LONG_LONG;}
   |LONG_LONG INT                {current_dtype = LONG_LONG;}
        |CHAR                                                   {current_dtype =
CHAR;}
        |FLOAT                                                  {current_dtype =
FLOAT;}
        |VOID                                                {current_dtype = VOID;}
```

```
        |CHAR_STAR                                                {current_dtype =
STRING;}
   ;

 /* grammar rules for argument list */
 /* argument list can be empty */
argument_list : arguments
        |
        ;
 /* arguments are comma separated TYPE ID pairs */
arguments : arguments ',' arg
        | arg
        ;

 /* Each arg is a TYPE ID pair */
arg : type identifier     {
                                                        param_list[p_idx++] = $2->data_type;
                                                        gencode(string("arg ") + $2->lexeme);
                                                }
   ;

 /* Generic statement. Can be compound or a single statement */
stmt:compound_stmt          {$$ = new content_t(); $$=$1;}
   |single_stmt          {$$ = new content_t(); $$=$1;}
   ;

 /* The function body is covered in braces and has multiple statements. */
compound_stmt :
                                '{'

                                {
                                        if(!p)current_scope = create_new_scope();
                                        else p = 0;
                                }

                                statements

                                '}'

                                {
                                        current_scope = exit_scope();
                                        $$ = new content_t();
                                        $$ = $3;
                                }
   ;
```

```
statements:statements M stmt {
                                                        backpatch($1-
>nextlist,$2);
                                                        $$ = new content_t();
                                                        $$->nextlist = $3-
>nextlist;
                                                        $$->breaklist =
merge($1->breaklist,$3->breaklist);
                                                        $$->continuelist =
merge($1->continuelist,$3->continuelist);
                                                }

   |                                            {       $$ = new content_t(); }
   ;

 /* Grammar for what constitutes every individual statement */
single_stmt :if_block  {
                                                $$ = new content_t();
                                                $$ = $1;
                                                backpatch($$->nextlist, nextinstr);
                                        }

                |for_block    {
                                                $$ = new content_t();
                                                $$ = $1;
                                                backpatch($$->nextlist, nextinstr);
                                        }

                |while_block {
                                                $$ = new content_t();
                                                $$ = $1;
                                                backpatch($$->nextlist, nextinstr);
                                         }
                |declaration            {$$ = new content_t();}
                |function_call ';'      {$$ = new content_t();}
                        |RETURN ';'     {
                                                        if(is_func)
                                                        {
                                                                if(func_type != VOID)
                                                                        yyerror("return
type (VOID) does not match function type");
                                                        }
                                                        else yyerror("return statement
not inside function definition");
```

```
                                                }

                        |CONTINUE ';'          {
                                                        if(!is_loop)
                                                                yyerror("Illegal use of
continue");
                                                        $$ = new content_t();
                                                        $$->continuelist = {nextinstr};
                                                        gencode("goto _");
                                                }

                        |BREAK ';'      {
                                                        if(!is_loop) {yyerror("Illegal use
of break");}
                                                        $$ = new content_t();
                                                        $$->breaklist = {nextinstr};
                                                        gencode("goto _");
                                                }

                        |RETURN sub_expr ';'
                                                {
                                                        if(is_func)
                                                        {
                                                                if(func_type != $2-
>data_type)
                                                                        yyerror("return
type does not match function type");
                                                        }
                                                        else yyerror("return statement
not in function definition");
                                                }
                ;

for_block: FOR '(' expression_stmt M expression_stmt M expression ')' {is_loop = 1;} N M
stmt {is_loop = 0;}
                {
                                backpatch($5->truelist,$11);
                                backpatch($12->nextlist,$6);
                                backpatch($12->continuelist, $6);
                                backpatch($10->nextlist, $4);
                                $$ = new content_t();
                                $$->nextlist = merge($5->falselist,$12->breaklist);
                                gencode(string("goto ") + to_string($6));
                        }
```

```
                ;

if_block:IF '(' expression ')' M stmt    %prec LOWER_THAN_ELSE
                        {
                                backpatch($3->truelist,$5);
                                $$ = new content_t();
                                $$->nextlist = merge($3->falselist,$6->nextlist);
                                $$->breaklist = $6->breaklist;
                                $$->continuelist = $6->continuelist;
                        }

                |IF '(' expression ')' M stmt  ELSE N M stmt
                        {
                                backpatch($3->truelist,$5);
                                backpatch($3->falselist,$9);

                                $$ = new content_t();
                                vector<int> temp = merge($6->nextlist,$8->nextlist);
                                $$->nextlist = merge(temp,$10->nextlist);
                                $$->breaklist = merge($10->breaklist,$6->breaklist);
                                $$->continuelist = merge($10->continuelist,$6->continuelist);
                        }
  ;

while_block: WHILE M '(' expression          ')' M {is_loop = 1;} stmt {is_loop = 0;}
                        {
                                backpatch($8->nextlist,$2);
                                backpatch($4->truelist,$6);
                                backpatch($8->continuelist, $2);
                                $$ = new content_t();
                                $$->nextlist = merge($4->falselist,$8->breaklist);
                                gencode(string("goto ") + to_string($2));
                        }
                ;

declaration: type  declaration_list ';'                 {is_declaration = 0;}
                        | declaration_list ';'
                        | unary_expr ';'


declaration_list: declaration_list ',' sub_decl
                                        |sub_decl
                                        ;

sub_decl: assignment_expr
```

```
                |identifier
                |array_access
                        ;

/* This is because we can have empty expession statements inside for loops */
expression_stmt: expression ';'
                                        {
                                                $$ = new content_t();
                                                $$->truelist = $1->truelist;
                                                $$->falselist = $1->falselist;
                                        }

                                | ';'   {       $$ = new content_t();  }
                        ;

expression: expression ',' sub_expr
                                {
                                        $$ = new content_t();
                                        $$->truelist = $3->truelist;
                                        $$->falselist = $3->falselist;
                                }
                | sub_expr
                                {
                                        $$ = new content_t();
                                        $$->truelist = $1->truelist;
                                        $$->falselist = $1->falselist;
                                }
                        ;

sub_expr:

                sub_expr '>' sub_expr
                        {
                                type_check($1->data_type,$3->data_type,2);
                                $$ = new content_t();
                                gencode_rel($$, $1, $3, string(" > "));
                        }
                | sub_expr '<' sub_expr
                        {
                                type_check($1->data_type,$3->data_type,2);
                                $$ = new content_t();
                                gencode_rel($$, $1, $3, string(" < "));
                        }

                | sub_expr EQ sub_expr
```

```
	{
		type_check($1->data_type,$3->data_type,2);
		$$ = new content_t();
		gencode_rel($$, $1, $3, string(" == "));
	}

| sub_expr NOT_EQ sub_expr
	{
		type_check($1->data_type,$3->data_type,2);
		$$ = new content_t();
		gencode_rel($$, $1, $3, string(" != "));
	}

| sub_expr GR_EQ sub_expr
	{
		type_check($1->data_type,$3->data_type,2);
		$$ = new content_t();
		gencode_rel($$, $1, $3, string(" >= "));
	}

| sub_expr LS_EQ sub_expr
	{
		type_check($1->data_type,$3->data_type,2);
		$$ = new content_t();
		gencode_rel($$, $1, $3, string(" <= "));
	}

|sub_expr LOGICAL_AND M sub_expr
	{
		type_check($1->data_type,$4->data_type,2);
		$$ = new content_t();
		$$->data_type = $1->data_type;
		backpatch($1->truelist,$3);
		$$->truelist = $4->truelist;
		$$->falselist = merge($1->falselist,$4->falselist);
	}

|sub_expr LOGICAL_OR M sub_expr
	{
		type_check($1->data_type,$4->data_type,2);
		$$ = new content_t();
		$$->data_type = $1->data_type;
		backpatch($1->falselist,$3);
		$$->truelist = merge($1->truelist,$4->truelist);
		$$->falselist = $4->falselist;
```

```
                }

        |'!' sub_expr
                {
                        $$ = new content_t();
                        $$->data_type = $2->data_type;
                        $$->truelist = $2->falselist;
                        $$->falselist = $2->truelist;
                }

        |arithmetic_expr
                {
                        $$ = new content_t();
                        $$->data_type = $1->data_type;
                        $$->addr = $1->addr;
                }
    |assignment_expr
                {
                        $$ = new content_t();
                        $$->data_type = $1->data_type;
                }
        |unary_expr
                {
                        $$ = new content_t();
                        $$->data_type = $1->data_type;
                }
  ;

assignment_expr :
        lhs assign arithmetic_expr
                {
                        type_check($1->entry->data_type,$3->data_type,1);
                        $$ = new content_t();
                        $$->data_type = $3->data_type;
                        $$->code = $1->entry->lexeme + *$2 + $3->addr;
                        gencode($$->code);
                        rhs = 0;
                }

   |lhs assign array_access
                {
                        type_check($1->entry->data_type,$3->data_type,1);
                        $$ = new content_t();
                        $$->data_type = $3->data_type;
                        $$->code = $1->entry->lexeme + *$2 + $3->code;
```

```
                        gencode($$->code);
                        rhs = 0;
                }

    |lhs assign function_call
                {
                        type_check($1->entry->data_type,$3,1);
                        $$ = new content_t();
                        $$->data_type = $3;
                }

        |lhs assign unary_expr
                {
                        type_check($1->entry->data_type,$3->data_type,1);
                        $$ = new content_t();
                        $$->data_type = $3->data_type;
                        $$->code = $1->entry->lexeme + *$2 + $3->code;
                        gencode($$->code);
                        rhs = 0;
                }

        |unary_expr assign unary_expr
                {
                        type_check($1->data_type,$3->data_type,1);
                        $$ = new content_t();
                        $$->data_type = $3->data_type;
                        $$->code = $1->code + *$2 + $3->code;
                        gencode($$->code);
                        rhs = 0;
                }
    ;

unary_expr:
        identifier INCREMENT
                {
                        $$ = new content_t();
                        $$->data_type = $1->data_type;
                        $$->code = string($1->lexeme) + string("++");
                        gencode($$->code);
                }

        | identifier DECREMENT
                {
                        $$ = new content_t();
                        $$->data_type = $1->data_type;
```

```
                        $$->code = string($1->lexeme) + string("--");
                        gencode($$->code);
                }

        | DECREMENT identifier
                {
                        $$ = new content_t();
                        $$->data_type = $2->data_type;
                        $$->code = string("--") + string($2->lexeme);
                        gencode($$->code);
                }

        | INCREMENT identifier
                {
                        $$ = new content_t();
                        $$->data_type = $2->data_type;
                        $$->code = string("++") + string($2->lexeme);
                        gencode($$->code);
                }

lhs: identifier         {$$ = new content_t(); $$->entry = $1;}
  | array_access        {$$ = new content_t(); $$->code = $1->code;}
        ;

identifier:IDENTIFIER
           {
              if(is_declaration && !rhs)
              {
               $1 = insert(SYMBOL_TABLE,yytext,INT_MAX,current_dtype);
               if($1 == NULL)
                                        yyerror("Redeclaration of variable");
              }
              else
              {
               $1 = search_recursive(yytext);
               if($1 == NULL)
                                        yyerror("Variable not declared");
              }

                                    $$ = $1;
           }
               ;

assign:'='                      {rhs=1; $$ = new string(" = ");}
   |ADD_ASSIGN      {rhs=1; $$ = new string(" += ");}
```

```
    |SUB_ASSIGN      {rhs=1; $$ = new string(" -= ");}
    |MUL_ASSIGN      {rhs=1; $$ = new string(" *= ");}
    |DIV_ASSIGN      {rhs=1;          $$ = new string(" /= ");}
    |MOD_ASSIGN      {rhs=1; $$ = new string(" %= ");}
    ;

arithmetic_expr: arithmetic_expr '+' arithmetic_expr
                                        {
                                                type_check($1->data_type,$3->data_type,0);
                                                $$ = new content_t();
                                                $$->data_type = $1->data_type;
                                                gencode_math($$, $1, $3, string(" + "));
                                        }

                        | arithmetic_expr '-' arithmetic_expr
                                        {
                                                type_check($1->data_type,$3->data_type,0);
                                                $$ = new content_t();
                                                $$->data_type = $1->data_type;
                                                gencode_math($$, $1, $3, string(" - "));
                                        }

                        | arithmetic_expr '*' arithmetic_expr
                                        {
                                                type_check($1->data_type,$3->data_type,0);
                                                $$ = new content_t();
                                                $$->data_type = $1->data_type;
                                                gencode_math($$, $1, $3, string(" * "));
                                        }

                        | arithmetic_expr '/' arithmetic_expr
                                        {
                                                type_check($1->data_type,$3->data_type,0);
                                                $$ = new content_t();
                                                $$->data_type = $1->data_type;
                                                gencode_math($$, $1, $3, string(" / "));
                                        }

                    | arithmetic_expr '%' arithmetic_expr
                                        {
                                                type_check($1->data_type,$3->data_type,0);
                                                $$ = new content_t();
                                                $$->data_type = $1->data_type;
                                                gencode_math($$, $1, $3, string(" % "));
                                        }
```

```
                |'(' arithmetic_expr ')'
                                {
                                        $$ = new content_t();
                                        $$->data_type = $2->data_type;
                                        $$->addr = $2->addr;
                                        $$->code = $2->code;
                                }

        |'-' arithmetic_expr %prec UMINUS
                                {
                                        $$ = new content_t();
                                        $$->data_type = $2->data_type;
                                        $$->addr = "t" + to_string(temp_var_number);
                                        std::string expr = $$->addr + " = " + "minus " +
$2->addr;
                                        $$->code = $2->code + expr;
                                        temp_var_number++;
                                }

    |identifier
                                {
                                        $$ = new content_t();
                                        $$->data_type = $1->data_type;
                                        $$->addr = $1->lexeme;
                                }

        |constant
                                {
                                        $$ = new content_t();
                                        $$->data_type = $1->data_type;
                                        $$->addr = to_string($1->value);
                                }
         ;

constant: DEC_CONSTANT          {$1->is_constant=1; $$ = $1;}
        | HEX_CONSTANT          {$1->is_constant=1; $$ = $1;}
        | CHAR_CONSTANT                 {$1->is_constant=1; $$ = $1;}
        | FLOAT_CONSTANT            {$1->is_constant=1; $$ = $1;}
  ;

array_access: identifier '[' array_index ']'
                        {
                                if(is_declaration)
                                {
```

```
                                        if($3->value <= 0)
                                                yyerror("size of array is not positive");
                                        else if($3->is_constant)
                                                $1->array_dimension = $3->value;
                                }
                                else if($3->is_constant)
                                {
                                        if($3->value > $1->array_dimension)
                                                yyerror("Array index out of bound");
                                        if($3->value < 0)
                                                yyerror("Array index cannot be
negative");
                                }

                                $$ = new content_t();
                                $$->data_type = $1->data_type;

                                if($3->is_constant)
                                        $$->code = string($1->lexeme) + string("[") +
to_string($3->value) + string("]");
                                else
                                        $$->code = string($1->lexeme) + string("[") +
string($3->lexeme) + string("]");
                                $$->entry = $1;
                        }

array_index: constant           {$$ = $1;}
                | identifier            {$$ = $1;}
                                        ;

function_call: identifier '(' parameter_list ')'
                        {
                                $$ = $1->data_type;
                                check_parameter_list($1,param_list,p_idx);
                                p_idx = 0;
                                gencode(string("call ") + $1->lexeme);
                        }

        | identifier '(' ')'
                        {
                                $$ = $1->data_type;
                                check_parameter_list($1,param_list,p_idx);
                                p_idx = 0;
                                gencode(string("call ") + $1->lexeme);
                        }
```

```
        ;

parameter_list:
            parameter_list ','  parameter
            |parameter
            ;

parameter: sub_expr
                                    {
                                            param_list[p_idx++] = $1->data_type;
                                            gencode(string("param ") + $1->addr);
                                    }
                    | STRING
                                    {
                                            param_list[p_idx++] = STRING;
                                            gencode(string("param ") + $1->lexeme);
                                    }
                    ;

M:                          {$$ = nextinstr;}
 ;

N:                          {
                                    $$ = new content_t;
                                    $$->nextlist = {nextinstr};
                                    gencode("goto _");
                            }
          ;

%%

void gencode(string x)
{
          std::string instruction;

          instruction = to_string(nextinstr) + string(": ") + x;
          ICG.push_back(instruction);
          nextinstr++;
}


void gencode_rel(content_t* & lhs, content_t* arg1, content_t* arg2, const string& op)
{
          lhs->data_type = arg1->data_type;
```

```
	lhs->truelist = {nextinstr};
	lhs->falselist = {nextinstr + 1};

	std::string code;

	code = string("if ") + arg1->addr + op + arg2->addr + string(" goto _");
	gencode(code);

	code = string("goto _");
	gencode(code);
}

void gencode_math(content_t* & lhs, content_t* arg1, content_t* arg2, const string& op)
{
	lhs->addr = "t" + to_string(temp_var_number);
	std::string expr = lhs->addr + string(" = ") + arg1->addr + op + arg2->addr;
	lhs->code = arg1->code + arg2->code + expr;

	temp_var_number++;

	gencode(expr);
}

void backpatch(vector<int>& v1, int number)
{
	for(int i = 0; i<v1.size(); i++)
	{
		string instruction = ICG[v1[i]];

		if(instruction.find("_") < instruction.size())
		{
			instruction.replace(instruction.find("_"),1,to_string(number));
			ICG[v1[i]] = instruction;
		}
	}
}

vector<int> merge(vector<int>& v1, vector<int>& v2)
{
	vector<int> concat;
	concat.reserve(v1.size() + v2.size());
	concat.insert(concat.end(), v1.begin(), v1.end());
	concat.insert(concat.end(), v2.begin(), v2.end());

	return concat;
```

```
}

void type_check(int left, int right, int flag)
{
	if(left != right)
	{
		switch(flag)
		{
			case 0: yyerror("Type mismatch in arithmetic expression"); break;
			case 1: yyerror("Type mismatch in assignment expression"); break;
			case 2: yyerror("Type mismatch in logical expression"); break;
		}
	}
}

void displayICG()
{
	ofstream outfile("ICG.code");

	for(int i=0; i<ICG.size();i++)
	outfile << ICG[i] <<endl;

	outfile << nextinstr << ": exit";

	outfile.close();
}

void printlist(vector<int> v){
	for(auto it:v)
		cout<<it<<" ";
	cout<<"Next: "<<nextinstr<<endl;
}

int main(int argc, char *argv[])
{
	 int i;
	 for(i=0; i<NUM_TABLES;i++)
	 {
	  symbol_table_list[i].symbol_table = NULL;
	  symbol_table_list[i].parent = -1;
	 }

	constant_table = create_table();
  symbol_table_list[0].symbol_table = create_table();
	yyin = fopen(argv[1], "r");
```

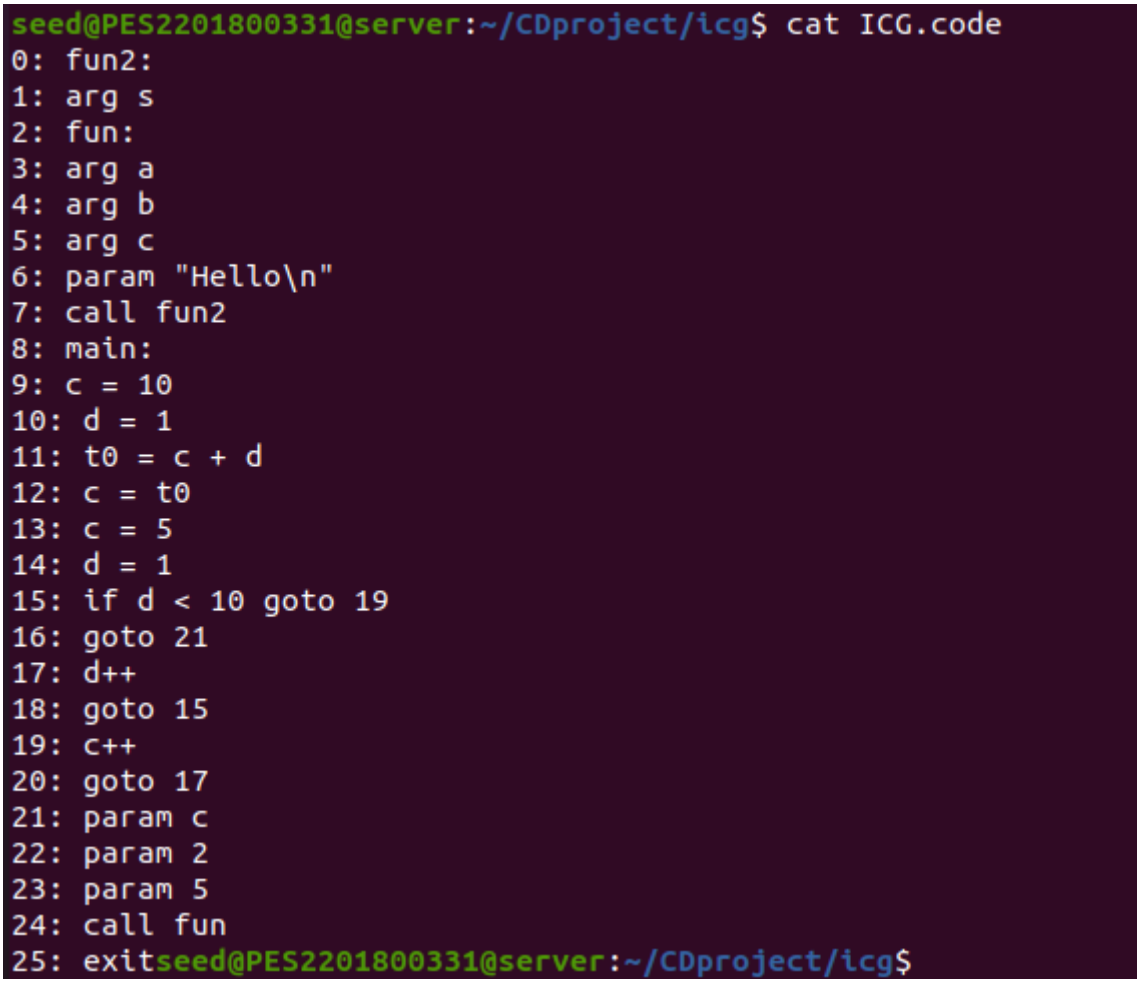
```
	if(!yyparse())
	{
		printf("\nPARSING COMPLETE\n\n\n");
	}
	else
	{
			printf("\nPARSING FAILED!\n\n\n");
	}

	displayICG();

	printf("SYMBOL TABLES\n\n");
	display_all();

	printf("CONSTANT TABLE");
	display_constant_table(constant_table);

}

int yyerror(const char *msg)
{
	printf("Line no: %d Error message: %s Token: %s\n", yylineno, msg, yytext);
	exit(0);
}
```

Sample input:

```
void fun2(char* s)
{

}

void fun(int a, int b, int c)
{
  fun2("Hello\n");
}

int main()
{
   int c=10;
   int d=1;
   c=c+d;
   c = 5;
   for(d=1;d<10;d++)
```

```
    c++;
    fun(c, 2 , 5);
}
```

Output:



2. Code optimization

The code optimizer maintains a key-value mapping that resembles table structure to keep track of variables and their values. This structure is used to perform various code optimization techniques.

a. Constant Folding:

```
#Constant Folding and Propagation
import re
import operator

def get_operator_fn(op):
   return {
      '+' : operator.add,
      '-' : operator.sub,
      '*' : operator.mul,
      '/' : operator.truediv,
      '%' : operator.mod,
      '^' : operator.xor,
      }[op]

def eval_binary_expr(op1, oper, op2):
   op1,op2 = int(op1), int(op2)
   return get_operator_fn(oper)(op1, op2)

f = open("sample2.txt","r")
content = f.readlines()
constant_table=dict() #dictionary with key as variable and value as its constant
for i in range(len(content)):
        #content[i]=content[i].replace(" ","")
        if '=' in content[i] and not '==' in content[i]: #Fix for case L1: t1=10
                Assignexpr = content[i].strip().split('=')
                variable=Assignexpr[0]

                if ':' in Assignexpr[0]:
                        lhs=Assignexpr[0].replace(" ","").split(":")
                        variable=lhs[1]
                        #print constant_table
                        constant_table={}

                var_list=re.split('\+|-|\*|/|%', Assignexpr[1])
                if len(var_list)==1: #pure assignment
                        if var_list[0].isdigit():
                                constant_table[variable]=var_list[0] #Case 2
                        else:
                                if var_list[0] in constant_table.keys():
                                        Assignexpr[1]=constant_table[var_list[0]] #Case 1
                        print(str(Assignexpr[0])+'='+str(Assignexpr[1]))
                #RHS contains multiple operands - 4 types
```

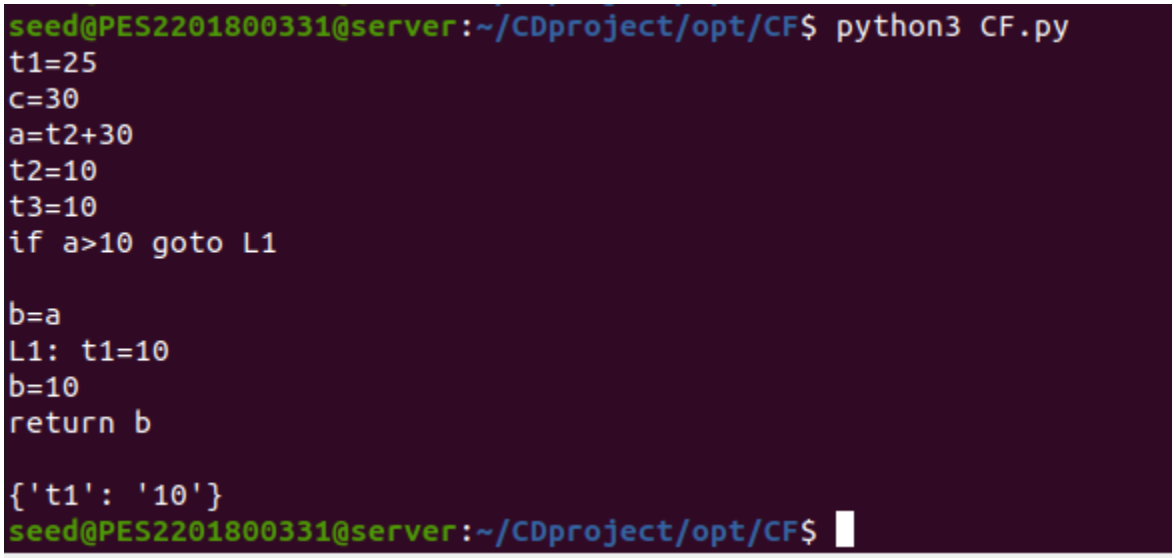
```
                # Type 1 - op1 is digit op2 is digit
                # Type 2 - op1 is digit op2 is variable
                # Type 3 - op1 is variable op2 is digit
                # Type 4 - op1 is variable op2 is variable

                if len(var_list)==2: #Case 3
                        constant_value="NOCHANGE"
                        op1 = var_list[0]
                        op2 = var_list[1]
                        if '+' in content[i]:
                                op='+'
                        if '-' in content[i]:
                                op='-'
                        if '*' in content[i]:
                                op='*'
                        if '/' in content[i]:
                                op='/'
                        if op1.isdigit() and op2.isdigit():
                                constant_value=eval_binary_expr(op1, op, op2)
                                constant_table[Assignexpr[0]]=constant_value
                        if op1.isdigit() and op2.isdigit()!=1:
                                if op2 in constant_table.keys():
                                        constant_value=eval_binary_expr(op1, op,
constant_table[op2])
                                        constant_table[Assignexpr[0]]=constant_value
                        if op1.isdigit()!=1 and op2.isdigit():
                                if op1 in constant_table.keys():
                                        constant_value=eval_binary_expr(constant_table[op1],
op, op2)
                                        constant_table[Assignexpr[0]]=constant_value
                        if op1.isdigit()!=1 and op2.isdigit()!=1:
                                if op1 in constant_table.keys():
                                        if op2 in constant_table.keys():

        constant_value=eval_binary_expr(constant_table[op1], op, constant_table[op2])
                                                constant_table[Assignexpr[0]]=constant_value
                                        else: #only op1 in constant table

        Assignexpr[1]=str(constant_table[op1])+str(op)+str(op2)
                                elif op2 in constant_table.keys():

        Assignexpr[1]=str(op1)+str(op)+str(constant_table[op2])
                        if constant_value!="NOCHANGE":
                                Assignexpr[1]=constant_value
                        print(str(Assignexpr[0])+'='+str(Assignexpr[1]))
```

```
        elif ':' in content[i]:
                constant_table={}
                print(content[i])
        else:
                print(content[i])
print(constant_table)
```

Sample input:

```
t1=5*5
c=t1+5
a=t2+c
t2=5+5
t3=t2
if a>10 goto L1
b=a
L1: t1=10
b=t1
return b
```

Output:



b. Common Sub-expression Evaluation:

```
#Common subexpression elimination
import re
```

```
f = open("sample1.txt","r")
content = f.readlines()
subexpr_table=dict() #dictionary with key as variable and value as its subexpr


#LOGIC
#Maintain 3 order tuple(op,op1,op2) in table. Store the temporary in temporaries list.
#if any of op1 or op2 is defined again later, remove the tuple from table.

for i in range(len(content)):
	#content[i]=content[i].replace(" ","")
	if '=' in content[i] and not '==' in content[i]: #Fix for case L1: t1=10
		Assignexpr = content[i].strip().split('=')
		variable=Assignexpr[0]		#variable holds the LHS value of assignment
		if ':' in Assignexpr[0]:
			subexpr_table=dict() #comment this line for across block CSE
			lhs=Assignexpr[0].replace(" ","").split(":")
			variable=lhs[1]

		var_list=re.split('\+|-|\*|/|%|>|<|>=|<=', Assignexpr[1]) #include carat in
RHS()
		if len(var_list)==1:
			found=0
			temp=""
			#print("variable",variable)
			for key,value in subexpr_table.items():
				for j in value:
					#print("here",variable,j)
					if variable==j: #one of the operands got redefined, so
pop that expression
						found=1
						temp=key
						break
			if(found==1):
				subexpr_table.pop(temp)
			print(content[i])

		if len(var_list)==2: #Expression
			tup=[]
			op1 = var_list[0]
			op2 = var_list[1]
			op=""
			flag=0
			line=content[i]
			if '+' in line:
```

```
                        op='+'
                    if '-' in line:
                        op='-'
                    if '*' in line:
                        op='*'
                    if '/' in line:
                        op='/'
                    if '<' in line:
                        op='<'
                    if '>' in line:
                        op='>'
                    if '<=' in line:
                        op='<='
                    if '>=' in line:
                        op='>='
                    if '%' in line:
                        op='%'
                    tup=[op,op1,op2]

                    for key,value in subexpr_table.items():
                        if tup==value: #common subexpression found
                            Assignexpr[1]=key #Assignment[1] is RHS of
assignment, so replace it with intermediate value holding the value of CS
                            flag=1
                    if(flag==0):
                        subexpr_table[variable]=tup #unique RHS,insert into subexpr
table

                    print(str(Assignexpr[0])+'='+str(Assignexpr[1]))

        elif '==' in content[i]:
            spl=content[i].strip().split('==')
            if '=' in spl[0]:
                tup=[]
                Assignexpr=spl[0].split('=') #Assignexpr[0] is lhs, assignexpr[1] is
first operand
                variable=Assignexpr[0]
                if ':' in Assignexpr[0]:
                    lhs=Assignexpr[0].replace(" ","").split(":")
                    variable=lhs[1]

                op1=Assignexpr[1]
                op2=spl[1]
                op="=="
                flag=0
```

```
                        tup=[op,op1,op2]
                        rhs=str(Assignexpr[1])+'=='+str(spl[1])

                        for key,value in subexpr_table.items():
                                if tup==value: #CS found
                                        rhs=key
                                        flag=1
                        if(flag==0):
                                subexpr_table[variable]=tup #unique expr, insert into subexpr
table
                        print(str(Assignexpr[0])+'='+rhs)
        else:
                print(content[i])
```

Sample Input:

```
i=0
t1=a+b
t2=i<5
a=10
t3=a+b
t4=a+b
L1: if t1 goto L2
t4=i+1
i=t4
goto L1
L2: i=10
```

Output:

```
seed@PES2201800331@server:~/CDproject/opt/CSE$ python3 CSE.py
i=0

t1=a+b
t2=i<5
a=10

t3=a+b
t4=t3
L1: if t1 goto L2

t4=i+1
i=t4

goto L1

L2: i=10
```

c. Constant Propagation:

```
#Constant Folding and Propagation
import re
import operator

def get_operator_fn(op):
   return {
      '+' : operator.add,
      '-' : operator.sub,
      '*' : operator.mul,
      '/' : operator.truediv,
      '%' : operator.mod,
      '^' : operator.xor,
      }[op]

def eval_binary_expr(op1, oper, op2):
   op1,op2 = int(op1), int(op2)
   return get_operator_fn(oper)(op1, op2)

f = open("sample2.txt","r")
content = f.readlines()
constant_table=dict() #dictionary with key as variable and value as its constant
for i in range(len(content)):
        #content[i]=content[i].replace(" ","")
        if '=' in content[i] and not '==' in content[i]: #Fix for case L1: t1=10
                Assignexpr = content[i].strip().split('=')
                variable=Assignexpr[0]

                if ':' in Assignexpr[0]:
                        lhs=Assignexpr[0].replace(" ","").split(":")
                        variable=lhs[1]
                        #print constant_table
                        constant_table={}

                var_list=re.split('\+|-|\*|/|%', Assignexpr[1])
                if len(var_list)==1: #pure assignment
                        if var_list[0].isdigit():
                                constant_table[variable]=var_list[0] #Case 2
                        else:
                                if var_list[0] in constant_table.keys():
```

```
                                        Assignexpr[1]=constant_table[var_list[0]] #Case 1
                                print(str(Assignexpr[0])+'='+str(Assignexpr[1]))
                        #RHS contains multiple operands - 4 types
                        # Type 1 - op1 is digit op2 is digit
                        # Type 2 - op1 is digit op2 is variable
                        # Type 3 - op1 is variable op2 is digit
                        # Type 4 - op1 is variable op2 is variable

                        if len(var_list)==2: #Case 3
                                constant_value="NOCHANGE"
                                op1 = var_list[0]
                                op2 = var_list[1]
                                if '+' in content[i]:
                                        op='+'
                                if '-' in content[i]:
                                        op='-'
                                if '*' in content[i]:
                                        op='*'
                                if '/' in content[i]:
                                        op='/'
                                if op1.isdigit() and op2.isdigit():
                                        constant_value=eval_binary_expr(op1, op, op2)
                                        constant_table[Assignexpr[0]]=constant_value
                                if op1.isdigit() and op2.isdigit()!=1:
                                        if op2 in constant_table.keys():
                                                constant_value=eval_binary_expr(op1, op,
constant_table[op2])
                                                constant_table[Assignexpr[0]]=constant_value
                                if op1.isdigit()!=1 and op2.isdigit():
                                        if op1 in constant_table.keys():
                                                constant_value=eval_binary_expr(constant_table[op1],
op, op2)
                                                constant_table[Assignexpr[0]]=constant_value
                                if op1.isdigit()!=1 and op2.isdigit()!=1:
                                        if op1 in constant_table.keys():
                                                if op2 in constant_table.keys():

        constant_value=eval_binary_expr(constant_table[op1], op, constant_table[op2])
                                                        constant_table[Assignexpr[0]]=constant_value
                                                else: #only op1 in constant table

        Assignexpr[1]=str(constant_table[op1])+str(op)+str(op2)
                                                elif op2 in constant_table.keys():

        Assignexpr[1]=str(op1)+str(op)+str(constant_table[op2])
```

```
                        if constant_value!="NOCHANGE":
                                Assignexpr[1]=constant_value
                        print(str(Assignexpr[0])+'='+str(Assignexpr[1]))
        elif ':' in content[i]:
                constant_table={}
                print(content[i])
        else:
                print(content[i])
print(constant_table)
```

Sample Input:

```
t1=5
c=t1+5
a=t2+c
t2=10
t3=t2
if a>10 goto L1
b=a
L1: t1=10
b=t1
return b
```

Output:

d. Dead Code Elimination

```
//Code Optimization Technique
#include<stdio.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;
/*
printf("Enter the Number of Values:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("left: ");
scanf(" %c",&op[i].l);
printf("right: ");
scanf(" %s",&op[i].r);
}
*/
n=3;
op[0].l='a';
strcpy(op[0].r,"10");
//op[0].r="10";
op[1].l='b';
strcpy(op[1].r,"c+d");
//op[1].r="c+d";
op[2].l='b';
strcpy(op[2].r,"c+d");
//op[2].r="c+d";
//op[3].l='e';
//strcpy(op[3].r,"f+g");
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 = ",pr[k].l);
printf("%s\n",pr[k].r);
}
}
```

Output:

```
seed@PES2201800331@server:~/CDproject/opt/DC$ ./a.out
Intermediate Code
a = 10
b = c+d
b = c+d

After Dead Code Elimination
b = c+d
seed@PES2201800331@server:~/CDproject/opt/DC$
```