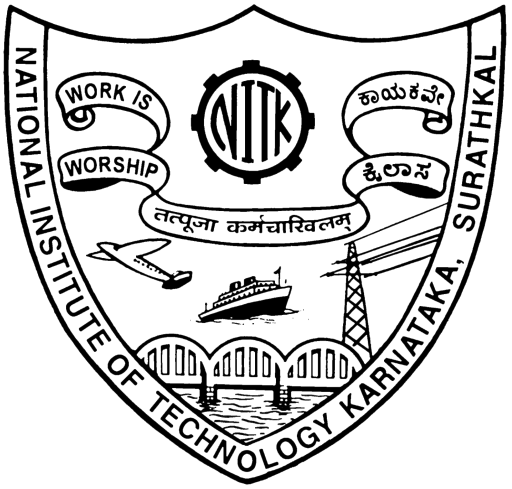
Intermediate Code Generator for the C Language



National Institute of Technology Karnataka Surathkal

Date: 28th March 2019

Submitted To:
Prof. P. Santhi Thilagam
CSE Dept, NITK

Group Members:
Namrata Ladda, 16CO121
Mehnaz Yunus, 16CO124
Sharanya Kamath, 16CO140

Abstract

This report contains the details of the tasks finished as a part of Phase Four of Compiler Design Lab. In this phase, we generated a three address code for C language .We tested the results by compiling the resulting program, running it, and checking that it produces the expected output.

FEATURES

The project objective is to construct a compiler that studies the C programming language. It will have the following features:

- Support int, float and char data types
- Detection of looping constructs such as while and nested while
- Detection conditional statements such as if-else and nested if-else.
- Identification of user-defined functions with one argument with return types int, char, void.
- Construction of symbol table.
- Support for single line as well as multiline comments and return appropriate error messages.
- Appropriate error messages for comments and strings that don't end until the end of the file.

RESULTS

- Generated three address code for the original grammar. Following cases are taken into consideration:
    - Conditional statements (if-else)
    - Looping construct (while loop)
    - Assignment expressions
    - Relational expressions

TOOLS USED

- Yacc
- Flex

Contents :

1. Introduction

1.1 Intermediate Code Generation

The front end of a compiler constructs an intermediate representation of the source program from which the back end generates the target program.
Two most important intermediate representations are:

- Trees, including parse trees and (abstract) syntax trees.
- Linear representations, especially “three-address code."

During parsing, syntax-tree nodes are created to represent significant programming constructs. As analysis proceeds, information is added to the nodes in the form of attributes associated with the nodes. The choice of attributes depends on the translation to be performed.

Three-address code, on the other hand, is a sequence of elementary program steps, such as the addition of two values. Unlike the tree, there is no hierarchical structure. We need this representation if we are to do any significant optimization of code. In that case, we break the long sequence of three-address statements that form a program into “basic blocks", which are sequences of statements that are always executed one-after-the-other, with no branching.

Three Address Code

Three-address code is a sequence of instructions of the form

$$x = y \; \mathbf{op} \; z$$

where x, y, and z are names, constants, or compiler-generated temporaries and op stands for an operator.

Three-address instructions are executed in numerical sequence unless forced to do otherwise by a conditional or unconditional jump. We choose the following instructions for control flow:

ifFalse x goto L	(if x is false, next execute the instruction labeled L)
ifTrue x goto L	(if x is true, next execute the instruction labeled L)
goto L	(next execute the instruction labeled L)

A label L can be attached to any instruction by prepending a prex L:. An instruction can have more than one label.
Finally, we need instructions that copy a value. The following three-address instruction copies the value of y into x:

$$x = y$$

1.2 Yacc Script

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one:
LALR(1) grammars with disambiguating rules.

The structure of our Yacc script is given below. Files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section
%%
Rules section
%%
C code section

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file. The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code. The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

1.3 C Program

The workflow is explained as under:

- Compile the script using Yacc tool
  `$ yacc semantic.y`
- Compile the flex *s*cript using Flex tool
- `$ lex lexer.l`
- After compiling the lex file, lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
- Compilation is done with the options –ll, –ly and -w

2. Design of Programs

2.1 Code

Phase 1: Lexical Analysis

```
%x comment
%{

    #include<stdio.h>
    #include<string.h>
    char bstack[100];
    int btop = -1;
    int nested_comment_stack = 0;
    int line = 0;

    struct hashtable{
        char name[100];
        char type[100];
        int len;
    }table[1000];

    int Hash(char *s){
        int mod = 1001;
        int l = strlen(s), val = 0, i;
        for(i = 0; i < l; i++){
            val = val * 10 + (s[i]-'A');
            val = val % mod;
            while(val < 0){
                val += mod;
            }
        }
        return val;
    }

    void insert_symbol(char *lexeme, char *token_name){

        int l1 = strlen(lexeme);
        int l2 = strlen(token_name);
        int v = Hash(lexeme);
        if(table[v].len == 0){
            strcpy(table[v].name, lexeme);
```

```
            strcpy(table[v].type, token_name);

            table[v].len = strlen(lexeme);
            return;
        }

        if(strcmp(table[v].name,lexeme) == 0)
        return;

        int i, pos = 0;

        for (i = 0; i < 1001; i++){
            if(table[i].len == 0){
                pos = i;
                break;
            }
        }

        strcpy(table[pos].name, lexeme);
        strcpy(table[pos].type, token_name);
        table[pos].len = strlen(lexeme);

    }

    void print(){
        int i;
        for(i = 0;i < 1001; i++){
            if(table[i].len == 0){
                continue;
            }
            printf("%15s \t %40s\n",table[i].name,table[i].type);
        }
    }

%}

LEQ <=
GEQ >=
EQ =
LES <
GRE >
PLUS \+
INCREMENT \+\+
DECREMENT \-\-
MINUS \-
```

```
MULT \*
DIV \/
REM %
AND &
OR \|
XOR \^
NOT \~

PREPROCESSOR
#(include<.*>|define.*|ifdef|endif|if|else|ifndef|undef|pragma)
STRING \".*\"|\'.*\'
WRONG_STRING \"[^"\n]*|\'[^'\n]*
SINGLELINE \/\/.*
MULTILINE "/*"([^*]|\*+[^*/])*\*+"/"
KEYWORD
auto|const|default|enum|extern|register|return|sizeof|static|struct|type
def|union|volatile|break|continue|goto|else|switch|if|case|default|for|d
o|while|char|double|float|int|long|short|signed|unsigned|void
IDENTIFIER [a-zA-Z_]([a-zA-Z0-9_])*
NUMBER_CONSTANT [1-9][0-9]*(\.[0-9]+)?|0(\.[0-9]+)?
OPERATOR {INCREMENT}|{DECREMENT}|{PLUS}|{MINUS}|{MULT}|{DIV}|{EQ}
COMPARISON {LEQ}|{GEQ}|{LES}|{GRE}
BITWISE {XOR}|{REM}|{AND}|{OR}|{NOT}
INVALID [^\n\t ]
WRONG_ID ([0-9\*\-\+\%\/]+[a-zA-Z][a-zA-Z0-9\*\-\+\%\/]*)


%%
\n line++;
[\t ] ;
; {printf("%s \t---- SEMICOLON DELIMITER\n", yytext);}
, {printf("%s \t---- COMMA DELIMITER\n", yytext);}
\{ {printf("%s \t---- PARENTHESIS\n", yytext);
      if(btop==-1){
            bstack[0]='{'; btop=1;}
      else {bstack[btop]='{';
      btop++;
    }
      }
\} {printf("%s \t---- PARENTHESIS\n", yytext);
      if(bstack[btop-1]!='{')
            printf("ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER:
%d\n",line);
      btop--;
      }
```

```
\( {printf("%s \t---- PARENTHESIS\n", yytext);
      if(btop==-1){
            bstack[0]='('; btop=1;}
      else {
        bstack[btop]='(';
      btop++;
    }
      }
\) {printf("%s \t---- PARENTHESIS\n", yytext);
      if(bstack[btop-1]!='(')
            printf("ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER:
%d\n",line);
      btop--;
      }
\[ {printf("%s \t---- PARENTHESIS\n", yytext);
      if(btop==-1){
            bstack[0]='['; btop=1;}
      else {
        bstack[btop]='[';
      btop++;
    }
      }
\] {printf("%s \t---- PARENTHESIS\n", yytext);
      if(bstack[btop-1]!='[')
            printf("ERROR: UNBALANCED PARENTHESIS AT LINE NUMBER:
%d\n",line);
      btop--;
      }
\\ {printf("%s \t- FSLASH\n", yytext);}
\. {printf("%s \t- DOT DELIMITER\n", yytext);}

"/*"                    {BEGIN(comment); nested_comment_stack=1;
yymore();}
<comment><<EOF>>        {printf("\nERROR: MULTILINE COMMENT: \"");
yyless(yyleng-2); ECHO; printf("\", NOT TERMINATED AT LINE NUMBER:
%d",line); yyterminate();}
<comment>"/*"           {nested_comment_stack++; yymore();}
<comment>.              {yymore();}
<comment>\n             {yymore();line++;}
<comment>"*/"           {nested_comment_stack--;
                        if(nested_comment_stack<0)
                        {
                          printf("\n \"%s\"\t---- ERROR: UNBALANCED
COMMENT AT LINE NUMBER: %d.", yytext, line);
                          yyterminate();
```

```
                        }
                        else if(nested_comment_stack==0)
                        {
                          BEGIN(INITIAL);
                        }
                        else
                          yymore();
                        }

"*/"                    {printf("%s \t---- ERROR: UNINITIALISED COMMENT
AT LINE NUMBER: %d\n", yytext,line); yyterminate();}

"//".*                  {printf("%s \t---- SINGLE LINE COMMENT\n",
yytext);}

{PREPROCESSOR} printf("%s \t---- PREPROCESSOR\n", yytext);
{STRING} {printf("%s \t---- STRING \n", yytext);
insert_symbol(yytext,"STRING CONSTANT");}
{MULTILINE} {printf("%s \t---- MULTI LINE COMMENT\n", yytext);}
{KEYWORD} {printf("%s \t---- KEYWORD\n", yytext); insert_symbol(yytext,
"KEYWORD");}
{IDENTIFIER} {printf("%s \t---- IDENTIFIER\n", yytext);
insert_symbol(yytext, "IDENTIFIER");}
{WRONG_ID} {printf("%s \t---- ERROR: ILL-FORMED IDENTIFIER\n", yytext);}
{NUMBER_CONSTANT} {printf("%s \t---- NUMBER CONSTANT\n", yytext);
insert_symbol(yytext, "NUMBER CONSTANT");}
{OPERATOR} {printf("%s \t---- ARITHMETIC OPERATOR\n", yytext);}
{BITWISE} {printf("%s \t---- BITWISE OPERATOR\n", yytext);}
{COMPARISON} {printf("%s \t---- COMPARISON OPERATOR\n", yytext);}
{WRONG_STRING} {printf("%s \t---- ERROR: UNTERMINATED STRING AT LINE
NUMBER: %d\n", yytext,line);}
{INVALID} {printf("%s \t---- ERROR: ILL-FORMED IDENTIFIER AT LINE
NUMBER: %d\n", yytext,line); }

%%

int yywrap(){
    return 1;
}

int main(){

    int i;
    for (i = 0; i < 1001; i++){
        table[i].len=0;
```

```
    }
    yyin = fopen("test-1.c","r");

    yylex();

printf("\n\n-----------------------------------------------------------------
---------------\n\t\t\t\tSYMBOL
TABLE\n-----------------------------------------------------------------------
----------\n");
      printf("\tLexeme \t\t\t\t\t\tToken\n");

printf("-----------------------------------------------------------------
------------\n");
    print();
}
```

Phase 2: Parser

lexer.l

```
%{
int yylineno;
#include <stdio.h>
%}

%%
\n    { yylineno++; }
"/*"  { multicomment(); }
"//"  { singlecomment(); }
"#include<"([A-Za-z_])*".h>" {}
"#define"([ ])""([A-Za-z_])""([A-Za-z_]|[0-9])*""([ ])""([0-9])+""
                        { }
"#define"([ ])""([A-Za-z_]([A-Za-z_]|[0-9])*)""([
])""(([0-9]+)\.([0-9]+))""          { }
"#define"([ ])""([A-Za-z_]([A-Za-z_]|[0-9])*)""([
])""([A-Za-z_]([A-Za-z_]|[0-9])*)""       { }

\"[^\n]*\"                    { yylval = yytext; return STRING_CONSTANT;
}
\'[A-Za-z_]\'                       { yylval = yytext; return
CHAR_CONSTANT; }
[0-9]+                              { yylval = yytext; return
INT_CONSTANT; }
([0-9]+)\.([0-9]+)                  { yylval = yytext; return
FLOAT_CONSTANT; }
```

```
([0-9]+)\.([0-9]+)([eE][-+]?[0-9]+)?      { yylval = yytext; return
FLOAT_CONSTANT; }

"sizeof"                        {  return SIZEOF; }
"++"            {  return INC_OP; }
"--"            {  return DEC_OP; }
"<<"            {  return LEFT_OP; }
">>"            {  return RIGHT_OP; }
"<="            {  return LE_OP; }
">="            {  return GE_OP; }
"=="            {  return EQ_OP; }
"!="            {  return NE_OP; }
"&&"            {  return AND_OP; }
"||"            {  return OR_OP; }
"*="            {  return MUL_ASSIGN; }
"/="            {  return DIV_ASSIGN; }
"%="            {  return MOD_ASSIGN; }
"+="            {  return ADD_ASSIGN; }
"-="            {  return SUB_ASSIGN; }
"<<="           {  return LEFT_ASSIGN; }
">>="           {  return RIGHT_ASSIGN; }
"&="            {  return AND_ASSIGN; }
"^="            {  return XOR_ASSIGN; }
"|="            {  return OR_ASSIGN; }
"char"                  {  yylval = yytext; return CHAR; }
"short"                 {  yylval = yytext; return SHORT; }
"short int"     {  yylval = yytext; return SHORT_INT; }
"int"           {  yylval = yytext; return INT; }
"float"                 {  yylval = yytext; return FLOAT; }
"long"                  {  yylval = yytext; return LONG; }
"long int"      {  yylval = yytext; return LONG_INT; }
"signed int"            {  yylval = yytext; return SIGNED_INT; }
"unsigned int"          {  yylval = yytext; return UNSIGNED_INT; }
"void"                  {  yylval = yytext; return VOID; }
"if"            {  return IF; }
"else"                  {  return ELSE; }
"while"                 {  return WHILE; }
"for"           {  return FOR; }
"break"                 {  return BREAK; }
"return"        {  return RETURN; }
";"                     {  return(';'); }
("{")           {  return('{'); }
("}")           {  return('}'); }
","                     {  return(','); }
":"                     {  return(':'); }
```

```
"="                     {  return('='); }
"("                     {  return('('); }
")"                     {  return(')'); }
("["|"<:")        {  return('['); }
("]"|":>")        {  return(']'); }
"."                     {  return('.'); }
"&"                     {  return('&'); }
"!"                     {  return('!'); }
"~"                     {  return('~'); }
"-"                     {  return('-'); }
"+"                     {  return('+'); }
"*"                     {  return('*'); }
"/"                     {  return('/'); }
"%"                     {  return('%'); }
"<"                     {  return('<'); }
">"                     {  return('>'); }
"^"                     {  return('^'); }
"|"                     {  return('|'); }
"?"                     {  return('?'); }
[A-Za-z_]([A-Za-z_]|[0-9])*              {  yylval = yytext; return
IDENTIFIER;  }
[ \t\v\n\f]       {  }
.                 { }
%%
yywrap()
{
      return(1);
}

multicomment()
{
      char c, c1;
      while ((c = input()) != '*' && c != 0);
      c1=input();
      if(c=='*' && c1=='/')
      {
            c=0;
      }
      if (c != 0)
            putchar(c1);
}

singlecomment()
{
      char c;
```

```
	while(c=input()!='\n');
	if(c=='\n')
		c=0;
	if(c!=0)
		putchar(c);
}
```

parser.y

```
%nonassoc NO_ELSE
%nonassoc ELSE
%left '<' '>' '=' GE_OP LE_OP EQ_OP NE_OP
%left  '+'  '-'
%left  '*'  '/' '%'
%left  '|'
%left  '&'
%token IDENTIFIER STRING_CONSTANT CHAR_CONSTANT INT_CONSTANT
FLOAT_CONSTANT SIZEOF
%token INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN
%token CHAR SHORT SHORT_INT INT LONG LONG_INT SIGNED_INT UNSIGNED_INT
FLOAT DOUBLE CONST VOID
%token IF ELSE WHILE FOR CONTINUE BREAK RETURN
%start start_state
%nonassoc UNARY
%glr-parser

%{
#include<string.h>
char type[100];
char temp[100];
%}

%%

start_state
	: global_declaration
	| start_state global_declaration
	;

global_declaration
	: function_definition
	| variable_declaration
```

```
	;

function_definition
	: type_specifier direct_declarator compound_statement
	;

type_specifier
	: VOID			{ $$ = "void"; }  { strcpy(type, $1); }
	| CHAR			{ $$ = "char"; }  { strcpy(type, $1); }
	| SHORT			{ $$ = "short"; }	{ strcpy(type, $1);
}
	| SHORT_INT	{ $$ = "short int"; }   { strcpy(type, $1); }
	| INT		{ $$ = "int"; }   { strcpy(type, $1); }
	| FLOAT			{ $$ = "float";}  { strcpy(type, $1); }
	| LONG			{ $$ = "long"; }  { strcpy(type, $1); }
	| LONG_INT	{ $$ = "long int"; }    { strcpy(type, $1); }
	| SIGNED_INT		{ $$ = "signed int"; }  { strcpy(type, $1);
}
	| UNSIGNED_INT		{ $$ = "unsigned int"; }{ strcpy(type, $1);
}
	;

direct_declarator
	: IDENTIFIER				{ symbolInsert($1, type);
}
	| '(' direct_declarator ')'
	| direct_declarator '[' conditional_expression ']'
	| direct_declarator '[' ']'
	| direct_declarator '(' parameter_type_list ')'
	| direct_declarator '(' identifier_list ')'
	| direct_declarator '(' ')'
	;

parameter_type_list
	: parameter_list
	;

parameter_list
	: parameter_declaration
	| parameter_list ',' parameter_declaration
	;

parameter_declaration
	: type_specifier direct_declarator
	| type_specifier
```

```
      ;

identifier_list
      : IDENTIFIER
      | identifier_list ',' IDENTIFIER
      ;

compound_statement
      : '{' '}'
      | '{' statement_list '}'
      | '{' declaration_list '}'
      | '{' declaration_list statement_list '}'
      | '{' declaration_list statement_list declaration_list
statement_list '}'
      | '{' declaration_list statement_list declaration_list '}'
      | '{' statement_list declaration_list statement_list '}'
      ;

declaration_list
      : variable_declaration
      | declaration_list variable_declaration
      ;

statement_list
      : statement
      | statement_list statement
      ;

statement
      : compound_statement
      | expression_statement
      | selection_statement
      | iteration_statement
      | jump_statement
      ;

selection_statement
      : IF '(' expression ')' statement %prec NO_ELSE
      | IF '(' expression ')' statement ELSE statement
      ;

iteration_statement
      : WHILE '(' expression ')' statement
      | FOR '(' expression_statement expression_statement expression ')'
      | FOR '(' expression_statement expression_statement ')'
```

```
	;

jump_statement
	: CONTINUE ';'
	| BREAK ';'
	| RETURN ';'
	| RETURN expression ';'
	;

expression_statement
	: ';'
	| expression ';'
	;

expression
	: assignment_expression
	| expression ',' assignment_expression
	;

assignment_expression
	: all_expression
	| assignment_expression assignment_operator all_expression
	;

assignment_operator
	: '='
	| ADD_ASSIGN
	;

all_expression
	: fundamental_expression
	| all_expression '>' all_expression
	| all_expression '<' all_expression
	| all_expression '+' all_expression
	| all_expression '-' all_expression
	| all_expression '/' all_expression
	| all_expression '*' all_expression
	| all_expression '%' all_expression
	;

fundamental_exp
	: IDENTIFIER
	| STRING_CONSTANT { constantInsert($1, "string"); }
	| CHAR_CONSTANT         { constantInsert($1, "char"); }
	| FLOAT_CONSTANT  { constantInsert($1, "float"); }
```

```
      | INT_CONSTANT          { constantInsert($1, "int"); }
      | '(' expression ')'
      ;

arg_list
      : assignment_expression
      | arg_list ',' assignment_expression
      ;

variable_declaration
      : type_specifier init_declarator_list ';'
      | error
      ;

init_declarator_list
      : init_declarator
      | init_declarator_list ',' init_declarator
      ;

init_declarator
      : direct_declarator
      | direct_declarator '=' init
      ;

init
      : assignment_expression
      | '{' init_list '}'
      | '{' init_list ',' '}'
      ;

init_list
      : init
      | init_list ',' init
      ;



%%
#include"lex.yy.c"
#include <ctype.h>
#include <stdio.h>
#include <string.h>

struct symbol
{
```

```
      char token[100];  // Name of the token
      char dataType[100];           // Date type: int, short int, long
int, char etc
}symbolTable[100000], constantTable[100000];

int i=0; // Number of symbols in the symbol table
int c=0;

//Insert function for symbol table
void symbolInsert(char* tokenName, char* DataType)
{
  strcpy(symbolTable[i].token, tokenName);
  strcpy(symbolTable[i].dataType, DataType);
  i++;
}

void constantInsert(char* tokenName, char* DataType)
{
      for(int j=0; j<c; j++)
      {
            if(strcmp(constantTable[j].token, tokenName)==0)
                  return;
      }
  strcpy(constantTable[c].token, tokenName);
  strcpy(constantTable[c].dataType, DataType);
  c++;
}

void showSymbolTable()
{


printf("\n\n----------------------------------------------------------------
----------------\n\t\t\t\tSYMBOL
TABLE\n----------------------------------------------------------------------
-----------\n");
      printf("\tSNo\t\t\tLexeme\t\t\tToken\n");

printf("----------------------------------------------------------------------
------------\n");
  for(int j=0;j<i;j++)

printf("%10d\t%20s\t\t\t<%s>\t\t\n",j+1,symbolTable[j].token,symbolTable
[j].dataType);
}
```

```
void showConstantTable()
{

printf("\n\n-----------------------------------------------------------------
----------------\n\t\t\t\tCONSTANT
TABLE\n-----------------------------------------------------------------------
-----------\n");
      printf("\tSNo\t\t\tConstant\t\tDatatype\n");

printf("-----------------------------------------------------------------------
------------\n");
  for(int j=0;j<c;j++)

printf("%10d\t%20s\t\t\t<%s>\t\t\n",j+1,constantTable[j].token,constantT
able[j].dataType);
}

int err=0;
int main(int argc, char *argv[])
{
      yyin = fopen(argv[1], "r");
      yyparse();
      if(err==0)
            printf("\nParsing complete\n");
      else
            printf("\nParsing failed\n");
      fclose(yyin);

      showSymbolTable();
      showConstantTable();
      return 0;
}
extern char *yytext;
extern int yylineno;
yyerror(char *s)
{
      err=1;
      printf("\nLine %d : %s\n", (yylineno), s);
      showSymbolTable();
      showConstantTable();
      exit(0);
}
```

Phase 3: Semantic Analysis

lexer.l

```
%{
int yylineno;
#include<stdio.h>
%}

%%
\n {yylineno++;}
"/*" {multiline();}
"//" {while(input() != '\n');}

#include[" "]*<[a-zA-Z]+\.h> {}

#include[" "]*\"[a-zA-Z]+\.h\" {}


\"[^\n]*\" { yylval = strdup(yytext); return STRING_CONST; }
[0-9]+ { yylval = strdup(yytext); return INT_CONST; }
[0-9]+\.[0-9]+ { yylval = strdup(yytext); return FLOAT_CONST;}

"{" { startBlock(); yylval = strdup(yytext); return ('{'); }
"}" { endBlock(); yylval = strdup(yytext); return ('}'); }
"(" { yylval = strdup(yytext); return ('('); }
")" { yylval = strdup(yytext); return (')'); }
"=" { yylval = strdup(yytext); return ('='); }
";" { yylval = strdup(yytext); return (';'); }
"+" { yylval = strdup(yytext); return ('+'); }
"-" { yylval = strdup(yytext); return ('-'); }
"*" { yylval = strdup(yytext); return ('*'); }
"/" { yylval = strdup(yytext); return ('/'); }
"%" { yylval = strdup(yytext); return ('%'); }
"<" { yylval = strdup(yytext); return ('<'); }
">" { yylval = strdup(yytext); return ('>'); }
"," { yylval = strdup(yytext); return (','); }
"+=" { yylval = strdup(yytext); return ADD_ASSIGN; }

"int" { yylval = strdup(yytext); return INT; }
"float" { yylval = strdup(yytext); return FLOAT; }
"void" { yylval = strdup(yytext); return VOID; }
"for" { yylval = strdup(yytext); return FOR; }
"return" { yylval = strdup(yytext); return RETURN; }
```

```
"printf" { yylval = strdup(yytext); return PRINTF; }
"while" { yylval = strdup(yytext); return WHILE; }

[A-Za-z_]([A-Za-z0-9_])* {yylval = strdup(yytext); return IDENTIFIER; }

[ \n\t] { }
. { }

%%

int yywrap() {
return 1;
}

void multiline() {
	char c, c1;
	while(((c=input()) != '*') && c != 0);
	c1 = input();
	if(!(c == '*' && c1 == '/'))
		putchar(c1);
}
```

semantic.y

```
%{
#include <string.h>
#include<stdio.h>
#include<stdlib.h>
#define YYSTYPE char*
extern int yylineno;
char type[200];
int scope = 0;
int n = 1;
char funcReturnType[100][100];
int currType = 0;
int var = 0; //no of variables declared in one statement

struct declaration_info {
	char id[100];
	char value[100];
}dec_info[100];

struct symbolTable
{
```

```
	char token[100];
	char type[100];
	int tn;
	float fvalue;
	int value;
	int scope;
	int isFunction;
	int fType[100];
	char paramType[100];
	int numParams;
}table[100];
void saveReturnType(int currType, char* returnType) {
	strcpy(funcReturnType[currType], returnType);
}

int isPresent(char* token){ //check if token is already present in
symbol table
	int i;
	for(i = 1; i < n; ++i)
		if(!strcmp(table[i].token, token))
			return i;
	return 0;
}

int getNumParams(char* token){
	int i;
	for(i = 1; i < n; ++i)
		if(strcmp(table[i].token, token) == 0)
			return table[i].numParams;
}

char* getParamType(char* token){
	int i;
	for(i = 1; i <n; ++i)
		if(strcmp(table[i].token, token) == 0)
			return table[i].paramType;
}

char* getDataType(char* token){
	int i;
	for(i = 1; i < n; ++i)
		if(strcmp(table[i].token, token) == 0)
			return table[i].type;
}
```

```
int getMinScope(char* token) {
	int i;
	int min = 999;
	for(i = 1; i < n; ++i){
		if(strcmp(table[i].token, token) == 0)
			if(table[i].scope < min)
				min = table[i].scope;
	}
	return min;
}

void storeValue(char* token, char* value, int scope){
	int i;
	for(i = 1; i < n; ++i){
		if(strcmp(table[i].token, token) == 0 && table[i].scope ==
scope) {
			if(strcmp(table[i].type, "float") == 0)
				table[i].fvalue = atof(value);
			if(strcmp(table[i].type, "int") == 0)
				table[i].value = atoi(value);
			if(strcmp(table[i].type, "char") == 0)
				table[i].value = value;
			break;
		}
	}
}

void insert(char* token, char* type, int scope, char* value) {
	if(!isPresent(token) || table[isPresent(token)].scope != scope){
		strcpy(table[n].token, token);
		strcpy(table[n].type, type);
		table[n].numParams = 0;
		table[n].isFunction = 0;
		table[n].scope = scope;
		if(strcmp(type, "float") == 0)
			table[n].fvalue = atof(value);
		if(strcmp(type, "int") == 0)
			table[n].value = atoi(value);
		n++;
	}
	return;
}

void insertFunction(char* token, char* type, char* paramType, int
numParams){
```

```
	if(!isPresent(token)){
		strcpy(table[n].token, token);
		strcpy(table[n].type, type);
		strcpy(table[n].paramType, paramType);
		table[n].numParams = numParams;
		table[n].isFunction = 1;
		table[n].scope = scope;
		n++;
	}
	return;
}

void showSymbolTable(){
	int i;

printf("\n\n----------------------------------------------------------------
--------------------------------\n\t\t\t\tSYMBOL
TABLE\n----------------------------------------------------------------------
---------------------------\n");
	printf("\n\nToken\tType\tParameterType\tNo of
Parameters\tisFunction\tValue\t\tScope");

printf("\n----------------------------------------------------------------
-----------------------------------\n");
	for(i = 1; i < n; ++i){
		printf("\n%s\t%s\t%s\t\t%d\t\t\t%d\t\t",
		table[i].token,
		table[i].type,
		(table[i].isFunction ? table[i].paramType : "-"),
		(table[i].isFunction ? table[i].numParams : 0),
		table[i].isFunction);

		!strcmp(table[i].type, "int") ?
table[i].value==9999?printf("-\t\t"):printf("%d\t\t",table[i].value) :
table[i].fvalue==9999.000000?printf("-\t\t"):printf("%f\t",table[i].fval
ue);
		printf("%d",table[i].scope);
	}
	printf("\n");
}

void end() {
	showSymbolTable();
	printf("\nParsing Failed\n");
	exit(0);
```

```
}

%}

%left '<' '>' '='
%left '+' '-'
%left '*' '/' '%'
%token IDENTIFIER
%token INT FOR RETURN PRINTF FLOAT VOID WHILE CHAR
%token STRING_CONST INT_CONST FLOAT_CONST
%token ADD_ASSIGN

%start start_state
%glr-parser

%%

start_state:
      function_definition_list
      ;

function_definition_list:
      function_definition_list function_definition
      | function_definition
      ;

function_definition:
      type_specifier IDENTIFIER '(' ')' compound_statement
       {
            if(strcmp($1, funcReturnType[currType-1]) != 0){
                  printf("\nError: Type mismatch at line no %d\n",
yylineno);
                  end();
            }
            else if(isPresent($2)) {
                  printf("\nError: Redefinition of function at line no
%d\n", yylineno);
                  end();
            }
            else
            insertFunction($2, $1, "-", 0);
      }
    | type_specifier IDENTIFIER '(' type_specifier_fn IDENTIFIER ')'
compound_statement
      {
```

```
            if(strcmp($1, funcReturnType[currType-1]) != 0){
                  printf("\nError Type mismatch at line no %d\n",
yylineno);
                  end();
            }
            else if(isPresent($2)) {
                  printf("\nError: Redefinition of function at line no
%d\n", yylineno);
                  end();
            }
            else
                  insertFunction($2, $1, $4, 1);
      }
      ;

type_specifier:
       INT  { $$ = "int"; strcpy(type, $1); }
      | FLOAT { $$ = "float"; }
      | VOID { $$ = "void"; }
      | CHAR { $$ = "char"; }
      ;
type_specifier_fn:
       INT  { $$ = "int"; }
      | FLOAT { $$ = "float"; }
      ;


compound_statement:
       '{' '}'
      | '{' statement_list '}'
      | '{' declaration_list '}'
      | '{' declaration_list statement_list '}'
      | '{' declaration_list statement_list declaration_list'}'
      | '{' declaration_list statement_list declaration_list
statement_list'}'
      ;

statement_list:
       statement
      | statement_list statement
      ;

statement:
       compound_statement
      | expression_statement
```

```
      | iteration_statement
      | jump_statement
      | print_statement
    | function_call_statement
      ;

expression_statement:
       expression ';'
      | ';'
      ;

expression:
       assignment_expression
      | expression ',' assignment_expression
      ;

assignment_expression:
       all_expression
      | assignment_expression assignment_operator all_expression
      {
            storeValue($1, $3, scope);
      }
      ;

assignment_operator:
       '='
      | ADD_ASSIGN
      ;

all_expression:
       fundamental_expression
      | all_expression '>' all_expression
      | all_expression '<' all_expression
      | all_expression '+' all_expression
      {
            char ans[100];
            int a = atoi($1) + atoi($3);
            sprintf(ans, "%d", a);
            strcpy($$, ans);
      }
      | all_expression '-' all_expression
      {
            char ans[100];
            int a = atoi($1) - atoi($3);
            sprintf(ans, "%d", a);
```

```
            strcpy($$, ans);
      }
      | all_expression '/' all_expression
      {
            char ans[100];
            int a = atoi($1) / atoi($3);
            sprintf(ans, "%d", a);
            strcpy($$, ans);
      }
      | all_expression '*' all_expression
      {
            char ans[100];
            int a = atoi($1) * atoi($3);
            sprintf(ans, "%d", a);
            strcpy($$, ans);
      }
      | all_expression '%' all_expression
      {
            char ans[100];
            int a = atoi($1) % atoi($3);
            sprintf(ans, "%d", a);
            strcpy($$, ans);
      }
      ;

fundamental_expression:
      IDENTIFIER
      {
            if(!isPresent($1)) {
                  printf("\nError: Undeclared variable at line %d\n",
yylineno);
                  end();
            }
            else{
                  int minScope = getMinScope($1);
                  if(scope < minScope){
                        printf("\nError Variable out of scope at line
%d\n", yylineno);
                        end();
                  }
            }

      }
      | STRING_CONST
      | INT_CONST
```

```
      | FLOAT_CONST
      | '(' expression ')'
      ;

iteration_statement:
       FOR '(' expression statement expression statement expression ')'
      | FOR '(' expression statement expression statement ')'
      | WHILE '(' expression ')'
      ;

jump_statement:
       RETURN INT_CONST ';'
       {
            strcpy(funcReturnType[currType], "int");
            currType++;
       }
      | RETURN FLOAT_CONST ';'
      { saveReturnType(currType, "float"); currType++; }
      | RETURN ';'
      {
            saveReturnType(currType, "void");
            currType++;
      }
      | RETURN IDENTIFIER ';'
      {
            char* types = getDataType($2);
            printf("Type%s", types);
            saveReturnType(currType, types);
            currType++;
      }
      ;

print_statement:
       PRINTF '(' STRING_CONST ',' init_declarator_list ')' ';'
      ;

function_call_statement:
       IDENTIFIER '(' ')' ';'
      {
            if(!isPresent($1)){
                  printf("\nError Undeclared function at line no %d\n",
yylineno);
                  end();
            }
            else if(getNumParams($1) != 0) {
```

```
                  printf("\nError at line no %d\n", yylineno);
                  end();
            }
      }
    | IDENTIFIER '(' IDENTIFIER ')' ';'
      {
            if(!isPresent($1)){
                  printf("Error Undeclared function at line no %d\n",
yylineno);
                  end();
            }
            else if(getNumParams($1) != 1){
                  printf("Error No of parameters does not match
signature at line no %d\n", yylineno);
                  end();
            }
            else if(strcmp(getParamType($1), getDataType($3)) != 0){
                  printf("Error Parameter type does not match signature
at line no %d\n", yylineno);
                  end();
            }

      }
    | IDENTIFIER '(' INT_CONST ')'';'
      {
            if(!isPresent($1)){
                  printf("\nError Undeclared function at line no %d\n",
yylineno);
                  end();
            }
            else if(getNumParams($1) != 1){
                  printf("\nError No of parameters does not match
signature at line no %d\n", yylineno);
                  end();
            }
            else if(strcmp(getParamType($1), "int") != 0){
                  printf("\nError Parameter type does not match
signature at line no %d\n", yylineno);
                  end();
            }

      }
      | IDENTIFIER '(' FLOAT_CONST ')' ';'
      {
            if(!isPresent($1)){
```

```
                  printf("Error Undeclared function at line no %d\n",
yylineno);
                  end();
            }
            else if(getNumParams($1) != 1){
                  printf("Error No of parameters does not match
signature at line no %d\n", yylineno);
                  end();
            }
            else if(strcmp(getParamType($1), "float") != 0){
                  printf("Error Parameter type does not match signature
at line no %d\n", yylineno);
                  end();
            }

      }
      ;

declaration_list:
       declaration
      | declaration_list declaration
      ;

declaration:
      type_specifier init_declarator_list ';'
      {
            for(int i = 0; i < var; ++i){
                  if(!isPresent(dec_info[i].id) ||
table[isPresent(dec_info[i].id)].scope != scope)
                        insert(dec_info[i].id, $1, scope,
dec_info[i].value);
                  else{
                        printf("\nError: Redeclaration of variable at
line no %d\n", yylineno);
                        end();
                  }
            }
            var = 0;
      }
      ;

init_declarator_list:
       init_declarator
      | init_declarator_list ',' init_declarator
      ;
```

```
init_declarator:
	variable
	{
	strcpy(dec_info[var].id, $1);
	strcpy(dec_info[var].value, "9999");
	var++;
	}
	| variable '=' assignment_expression
	{
		strcpy(dec_info[var].id, $1);
		strcpy(dec_info[var].value, $3);
		var++;
	}
	;

variable:
	 IDENTIFIER
	;
%%

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


int err = 0;
int main(int argc, char* argv[]) {
	yyin = fopen(argv[1], "r");
	yyparse();

	if(err == 0)
		printf("\nParsing Complete\n");
	else
		printf("\nParsing Failed\n");

	fclose(yyin);

	showSymbolTable();
}

extern char* yytext;
extern int yylineno;
yyerror() {
```

```
	err = 1;
	printf("\nYYError at line no %d\n", yylineno);
	exit(0);
}

void startBlock() {
	scope++;
	return;
}

void endBlock() {
	scope--;
	return;
}
```

Intermediate Code Generation

lexer.l

```
%{
int yylineno;
#include<stdio.h>
%}

%%
\n    {yylineno++;}
"/*"  {multiline();}
"//"  {while(input() != '\n');}

#include[" "]*<[a-zA-Z]+\.h> {}

#include[" "]*\"[a-zA-Z]+\.h\" {}


\"[^\n]*\"  { yylval = strdup(yytext); return STRING_CONST; }
[0-9]+            { yylval = strdup(yytext); return INT_CONST; }
[0-9]+\.[0-9]+    { yylval = strdup(yytext); return FLOAT_CONST;}

"{"         { startBlock(); yylval = strdup(yytext); return ('{'); }
"}"         { endBlock(); yylval = strdup(yytext); return ('}'); }
"("         { yylval = strdup(yytext); return ('('); }
")"         { yylval = strdup(yytext); return (')'); }
"="         { yylval = strdup(yytext); return ('='); }
";"         { yylval = strdup(yytext); return (';'); }
"+"         { yylval = strdup(yytext); return ('+'); }
"-"         { yylval = strdup(yytext); return ('-'); }
```

```
"*"         { yylval = strdup(yytext); return ('*'); }
"/"         { yylval = strdup(yytext); return ('/'); }
"%"         { yylval = strdup(yytext); return ('%'); }
"<"         { yylval = strdup(yytext); return ('<'); }
">"         { yylval = strdup(yytext); return ('>'); }
","         { yylval = strdup(yytext); return (','); }
"+="        { yylval = strdup(yytext); return ADD_ASSIGN; }

"int"             { yylval = strdup(yytext); return INT; }
"float"     { yylval = strdup(yytext); return FLOAT; }
"void"            { yylval = strdup(yytext); return VOID; }
"for"             { yylval = strdup(yytext); return FOR; }
"return"    { yylval = strdup(yytext); return RETURN; }
"printf"    { yylval = strdup(yytext); return PRINTF; }
"while"     { yylval = strdup(yytext); return WHILE; }
"char"            { yylval = strdup(yytext); return CHAR; }
"if"        { yylval = strdup(yytext); return IF; }
"else"            { yylval = strdup(yytext); return ELSE; }

[A-Za-z_]([A-Za-z0-9_])* {yylval = strdup(yytext); return IDENTIFIER; }

[ \n\t]     { }
.           { }

%%

int yywrap() {
return 1;
}

void multiline() {
      char c, c1;
      while(((c=input()) != '*') && c != 0);
      c1 = input();
      if(!(c == '*' && c1 == '/'))
            putchar(c1);
}
```

icg.y

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

```
#define YYSTYPE char*
extern int yylineno;
char type[200];
int scope = 0;
int n = 1;
char funcReturnType[100][100];
int currType = 0;
int var = 0; //no of variables declared in one statement

char tokenstack[100][10],i_[2]="0",temp[2]="t", null[2]=" ";
int top = 0;
int label[20],label_index=0,ltop=0;

struct declaration_info {
      char id[100];
      char value[100];
}dec_info[100];

struct symbolTable
{
      char token[100];
      char type[100];
      int tn;
      float fvalue;
      int value;
      int scope;
      int isFunction;
      int fType[100];
      char paramType[100];
      int numParams;
}table[100];

void saveReturnType(int currType, char* returnType) {
      strcpy(funcReturnType[currType], returnType);
}

int isPresent(char* token){ //check if token is already present in
symbol table
      int i;
      for(i = 1; i < n; ++i)
            if(!strcmp(table[i].token, token))
                  return i;
      return 0;
}
```

```
int getNumParams(char* token){
	int i;
	for(i = 1; i < n; ++i)
		if(strcmp(table[i].token, token) == 0)
			return table[i].numParams;
}

char* getParamType(char* token){
	int i;
	for(i = 1; i <n; ++i)
		if(strcmp(table[i].token, token) == 0)
			return table[i].paramType;
}

char* getDataType(char* token){
	int i;
	for(i = 1; i < n; ++i)
		if(strcmp(table[i].token, token) == 0)
			return table[i].type;
}

int getMinScope(char* token) {
	int i;
	int min = 999;
	for(i = 1; i < n; ++i){
		if(strcmp(table[i].token, token) == 0)
			if(table[i].scope < min)
				min = table[i].scope;
	}
	return min;
}

void storeValue(char* token, char* value, int scope){
	int i;
	for(i = 1; i < n; ++i){
		if(strcmp(table[i].token, token) == 0 && table[i].scope ==
scope) {
			if(strcmp(table[i].type, "float") == 0)
				table[i].fvalue = atof(value);
			if(strcmp(table[i].type, "int") == 0)
				table[i].value = atoi(value);
			if(strcmp(table[i].type, "char") == 0)
				table[i].value = value;
			break;
		}
```

```
	}
}

void insert(char* token, char* type, int scope, char* value) {
	if(!isPresent(token) || table[isPresent(token)].scope != scope){
		strcpy(table[n].token, token);
		strcpy(table[n].type, type);
		table[n].numParams = 0;
		table[n].isFunction = 0;
		table[n].scope = scope;
		if(strcmp(type, "float") == 0)
			table[n].fvalue = atof(value);
		if(strcmp(type, "int") == 0)
			table[n].value = atoi(value);
		n++;
	}
	return;
}

void insertFunction(char* token, char* type, char* paramType, int
numParams){
	if(!isPresent(token)){
		strcpy(table[n].token, token);
		strcpy(table[n].type, type);
		strcpy(table[n].paramType, paramType);
		table[n].numParams = numParams;
		table[n].isFunction = 1;
		table[n].scope = scope;
		n++;
	}
	return;
}

void showSymbolTable(){
	int i;

printf("\n\n----------------------------------------------------------------
--------------------------------\n\t\t\t\tSYMBOL
TABLE\n----------------------------------------------------------------------
---------------------------\n");
	printf("\n\nToken\tType\tParameterType\tNo of
Parameters\tisFunction\tValue\t\tScope");

printf("\n----------------------------------------------------------------
----------------------------------\n");
```

```
	for(i = 1; i < n; ++i){
		printf("\n%s\t%s\t%s\t\t%d\t\t\t%d\t\t",
		table[i].token,
		table[i].type,
		(table[i].isFunction ? table[i].paramType : "-"),
		(table[i].isFunction ? table[i].numParams : 0),
		table[i].isFunction);

		!strcmp(table[i].type, "int") ?
table[i].value==9999?printf("-\t\t"):printf("%d\t\t",table[i].value) :
table[i].fvalue==9999.000000?printf("-\t\t"):printf("%f\t",table[i].fval
ue);
		printf("%d",table[i].scope);
	}
	printf("\n");
}

void end() {
	showSymbolTable();
	printf("\nParsing Failed\n");
	exit(0);
}

void gencode()
{
	strcpy(temp,"t");
	strcat(temp,i_);
	printf("----------%s = %s %s
%s\n",temp,tokenstack[top-2],tokenstack[top-1],tokenstack[top]);
	top-=2;
	strcpy(tokenstack[top],temp);
	i_[0]++;
}

void gencodeAssignment()
{
	printf("----------%s = %s\n",tokenstack[top-2],tokenstack[top]);
	top-=2;
}

void push(char *a)
{
	strcpy(tokenstack[++top],a);
}
```

```
void if_gencode1()
{
      label_index++;
      strcpy(temp,"t");
      strcat(temp,i_);
      printf("----------%s = not %s\n",temp,tokenstack[top]);
      printf("----------if %s goto L%d\n",temp,label_index);
      i_[0]++;
      label[++ltop]=label_index;
}

void if_gencode2()
{
      label_index++;
      printf("----------goto L%d\n",label_index);
      printf("----------L%d: \n",label[ltop--]);
      label[++ltop]=label_index;
}

void if_gencode3()
{
      printf("----------L%d:\n",label[ltop--]);
}

void while_gencode1()
{
      label_index++;
      label[++ltop]=label_index;
      printf("----------L%d:\n",label_index);
}
void while_gencode2()
{
      label_index++;
      strcpy(temp,"t");
      strcat(temp,i_);
      printf("----------%s = not %s\n",temp,tokenstack[top--]);
      printf("----------if %s goto L%d\n",temp,label_index);
      i_[0]++;
      label[++ltop]=label_index;
}
void while_gencode3()
{
      int y=label[ltop--];
      printf("----------goto L%d\n",label[ltop--]);
      printf("----------L%d:\n",y);
```

```
}

%}

%left '<' '>' '='
%left '+' '-'
%left '*' '/' '%'
%token IDENTIFIER
%token INT FOR RETURN PRINTF FLOAT VOID WHILE CHAR IF ELSE
%token STRING_CONST INT_CONST FLOAT_CONST
%token ADD_ASSIGN

%start start_state
%glr-parser

%%

start_state:
      function_definition_list
      ;

function_definition_list:
      function_definition_list function_definition
      | function_definition
      ;

function_definition:
      type_specifier IDENTIFIER '(' ')' { printf("\nfunction begin
%s:\n", $2); } compound_statement
       {
            if(strcmp($1, funcReturnType[currType-1]) != 0){
                  printf("\nError: Return Type mismatch at line no
%d\n", yylineno);
                  end();
            }
            else if(isPresent($2)) {
                  printf("\nError: Redefinition of function at line no
%d\n", yylineno);
                  end();
            }
            else
            insertFunction($2, $1, "-", 0);
            printf("function end\n\n");
      }
    | type_specifier IDENTIFIER '(' type_specifier_fn IDENTIFIER ')' {
```

```
printf("\nfunction begin %s:\n", $2); } compound_statement
      {
            if(strcmp($1, funcReturnType[currType-1]) != 0){
                  printf("\nError Return Type mismatch at line no %d\n",
yylineno);
                  end();
            }
            else if(isPresent($2)) {
                  printf("\nError: Redefinition of function at line no
%d\n", yylineno);
                  end();
            }
            else
                  insertFunction($2, $1, $4, 1);
            printf("function end\n\n");
      }
      ;

type_specifier:
       INT  { $$ = "int"; strcpy(type, $1); }
      | FLOAT { $$ = "float"; }
      | VOID { $$ = "void"; }
      | CHAR { $$ = "char"; }
      ;

type_specifier_fn:
       INT  { $$ = "int"; }
      | FLOAT { $$ = "float"; }
      ;


compound_statement:
       '{' '}'
      | '{' statement_list '}'
      | '{' declaration_list '}'
      | '{' declaration_list statement_list '}'
      | '{' declaration_list statement_list declaration_list'}'
      | '{' declaration_list statement_list declaration_list
statement_list'}'
      ;

statement_list:
       statement
      | statement_list statement
      ;
```

```
statement:
       compound_statement
      | expression_statement
      | iteration_statement
      | jump_statement
      | print_statement
    | function_call_statement
      | if_statement
      ;

expression_statement:
       expression ';'
      | ';'
      ;

expression:
       assignment_expression
      | expression ',' assignment_expression
      ;

assignment_expression:
       all_expression
      | assignment_expression {push($1);} assignment_operator
all_expression {gencodeAssignment();}
      {
            storeValue($1, $3, scope);
      }
      ;

assignment_operator:
       '=' {strcpy(tokenstack[++top],"=");}
      | ADD_ASSIGN {strcpy(tokenstack[++top],"+=");}
      ;

all_expression:
      all_expression '>'{strcpy(tokenstack[++top],">");}
rel_expression{gencode();}
   | all_expression '<'{strcpy(tokenstack[++top],"<");}
rel_expression{gencode();}
   | rel_expression
   ;

rel_expression:
      rel_expression '+'{strcpy(tokenstack[++top],"+");}
```

```
additive_exp{gencode();}
   | rel_expression '-'{strcpy(tokenstack[++top],"-");}
additive_exp{gencode();}
   | additive_exp
   ;

additive_exp:
      additive_exp '*'{strcpy(tokenstack[++top],"*");}
fundamental_expression{gencode();}
   | additive_exp '/'{strcpy(tokenstack[++top],"/");}
fundamental_expression{gencode();}
   | fundamental_expression
   ;

fundamental_expression:
      IDENTIFIER
      {
            if(!isPresent($1)) {
                  printf("\nError: Undeclared variable at line %d\n",
yylineno);
                  end();
            }
            else{
                  int minScope = getMinScope($1);
                  if(scope < minScope){
                        printf("\nError Variable out of scope at line
%d\n", yylineno);
                        end();
                  }
                  else
                        push($1);
            }

      }
      | STRING_CONST {push($1);}
      | INT_CONST {push($1);}
      | FLOAT_CONST {push($1);}
      //| '(' expression ')' {$$=$2;}
      ;

iteration_statement:
       FOR '(' expression statement expression statement expression ')'
      | FOR '(' expression statement expression statement ')'
      | WHILE {while_gencode1();} '(' expression')' {while_gencode2();}
compound_statement {while_gencode3();}
```

```
      ;

if_statement
      :      IF '(' expression ')' {if_gencode1();} compound_statement
{if_gencode2();} else_statement
      ;

else_statement
      : ELSE compound_statement {if_gencode3();}
      ;

jump_statement:
       RETURN INT_CONST ';'
       {
            strcpy(funcReturnType[currType], "int");
            currType++;
       }
      | RETURN FLOAT_CONST ';'
      { saveReturnType(currType, "float"); currType++; }
      | RETURN ';'
      {
            saveReturnType(currType, "void");
            currType++;
      }
      | RETURN IDENTIFIER ';'
      {
            char* types = getDataType($2);
            printf("Type%s", types);
            saveReturnType(currType, types);
            currType++;
      }
      ;

print_statement:
       PRINTF '(' STRING_CONST ',' init_declarator_list ')' ';'
      ;

function_call_statement:
       IDENTIFIER '(' ')' ';'
      {
            if(!isPresent($1)){
                  printf("\nError Undeclared function at line no %d\n",
yylineno);
                  end();
            }
```

```
            else if(getNumParams($1) != 0) {
                  printf("\nError at line no %d\n", yylineno);
                  end();
            }
      }
    | IDENTIFIER '(' IDENTIFIER ')' ';'
      {
            if(!isPresent($1)){
                  printf("Error Undeclared function at line no %d\n",
yylineno);
                  end();
            }
            else if(getNumParams($1) != 1){
                  printf("Error No of parameters does not match
signature at line no %d\n", yylineno);
                  end();
            }
            else if(strcmp(getParamType($1), getDataType($3)) != 0){
                  printf("Error Parameter type does not match signature
at line no %d\n", yylineno);
                  end();
            }

      }
    | IDENTIFIER '(' INT_CONST ')'';'
      {
            if(!isPresent($1)){
                  printf("\nError Undeclared function at line no %d\n",
yylineno);
                  end();
            }
            else if(getNumParams($1) != 1){
                  printf("\nError No of parameters does not match
signature at line no %d\n", yylineno);
                  end();
            }
            else if(strcmp(getParamType($1), "int") != 0){
                  printf("\nError Parameter type does not match
signature at line no %d\n", yylineno);
                  end();
            }

      }
      | IDENTIFIER '(' FLOAT_CONST ')' ';'
      {
```

```
            if(!isPresent($1)){
                  printf("Error Undeclared function at line no %d\n",
yylineno);
                  end();
            }
            else if(getNumParams($1) != 1){
                  printf("Error No of parameters does not match
signature at line no %d\n", yylineno);
                  end();
            }
            else if(strcmp(getParamType($1), "float") != 0){
                  printf("Error Parameter type does not match signature
at line no %d\n", yylineno);
                  end();
            }

      }
      ;

declaration_list:
       declaration
      | declaration_list declaration
      ;

declaration:
      type_specifier init_declarator_list ';'
      {
            for(int i = 0; i < var; ++i){
                  if(!isPresent(dec_info[i].id) ||
table[isPresent(dec_info[i].id)].scope != scope)
                        insert(dec_info[i].id, $1, scope,
dec_info[i].value);
                  else{
                        printf("\nError: Redeclaration of variable at
line no %d\n", yylineno);
                        end();
                  }
            }
            var = 0;
      }
      ;

init_declarator_list:
       init_declarator
      | init_declarator_list ',' init_declarator
```

```
	;

init_declarator:
	variable
	{
	strcpy(dec_info[var].id, $1);
	strcpy(dec_info[var].value, "9999");
	var++;
	}
	| variable {push($1);} '=' {strcpy(tokenstack[++top],"=");}
assignment_expression {gencodeAssignment();}
	{
		strcpy(dec_info[var].id, $1);
		strcpy(dec_info[var].value, $3);
		var++;
	}
	;

variable:
	 IDENTIFIER
	;
%%

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


int err = 0;
int main(int argc, char* argv[]) {
	yyin = fopen(argv[1], "r");
	yyparse();

	if(err == 0)
		printf("\nParsing Complete\n");
	else
		printf("\nParsing Failed\n");

	fclose(yyin);

	showSymbolTable();
}

extern char* yytext;
```

```
extern int yylineno;
yyerror() {
	err = 1;
	printf("\nYYError at line no %d\n", yylineno);
	exit(0);
}

void startBlock() {
	scope++;
	return;
}

void endBlock() {
	scope--;
	return;
}
```

2.2 Explanation

We have generated three address code for the original grammar. Following cases are taken into consideration:

- Conditional statements (if-else)
- Looping construct (while loop)
- Assignment expressions
- Relational expressions

3. Test Cases

Without Errors:

3.1 test1.c

```
#include <stdio.h>

int main()
{
    int x=1;
    int y=2;
    int z=3;
    int ans=x+y+z-x*y/z;
    return 0;
}
```

```
$ ./a.out test1.c

function begin main:
x = 1
y = 2
z = 3
t0 = x + y
t1 = t0 + z
t2 = x * y
t3 = t2 / z
t4 = t1 - t3
ans = t4
function end


Parsing Complete


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------


Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------------

x       int     -               0                       0               0               1
y       int     -               0                       0               0               1
z       int     -               0                       0               0               1
ans     int     -               0                       0               0               1
main    int     -               0                       1               0               0
$
```

3.2 test2.c

```c
#include <stdio.h>

int main()
{
	int x=1;
	int y=2;
	if(x>3)
	{
		y=y-2;
	}
		else
		{
			y=0;
		}
	return 0;
}
```

```
$ ./a.out test2.c

function begin main:
x = 1
y = 2
t0 = x > 3
t1 = not t0
if t1 goto L1
t2 = y - 2
y = t2
goto L2
L1:
y = 0
L2:
function end


Parsing Complete


------------------------------------------------------------------------------------------
                              SYMBOL TABLE
------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------

x       int     -               0                       0               0               1
y       int     -               0                       0               0               1
main    int     -               0                       1               0               0
$
```

3.3 test3.c

```c
#include <stdio.h>

int main()
{
	int x=5;
	int y=6;
	while(x>7)
	{
		y=y+1;
	}
	return 0;
}
```

```
$ ./a.out test3.c

function begin main:
x = 5
y = 6
L1:
t0 = x > 7
t1 = not t0
if t1 goto L2
t2 = y + 1
y = t2
goto L1
L2:
function end


Parsing Complete


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------------

x       int     -               0                       0               0               1
y       int     -               0                       0               0               1
main    int     -               0                       1               0               0
$
```

3.4 test4.c

```c
#include <stdio.h>
int main()
{
	int a=5;
	int b=6;
	if(a<7)
	{
		if(a>9)
		{
			b=b*8;
			b=9;
		}
		else
		{
			a=10;
		}
	}
	else
	{
		b=2;
	}
	return 0;
}
```

```
$ ./a.out test4.c

function begin main:
a = 5
b = 6
t0 = a < 7
t1 = not t0
if t1 goto L1
t2 = a > 9
t3 = not t2
if t3 goto L2
t4 = b * 8
b = t4
b = 9
goto L3
L2:
a = 10
L3:
goto L4
L1:
b = 2
L4:
function end


Parsing Complete


------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------

a       int     -               0                       0               0               1
b       int     -               0                       0               0               1
main    int     -               0                       1               0               0
$
```

3.5 test5.c

```c
#include <stdio.h>
int main()
{
	int a=5;
	int b=6;
	while(a>7)
	{
		b=6;
		while(b>5)
		{
			a=9;
		}
	}
	return 0;
}
```

```
$ ./a.out test5.c

function begin main:
a = 5
b = 6
L1:
t0 = a > 7
t1 = not t0
if t1 goto L2
b = 6
L3:
t2 = b > 5
t3 = not t2
if t3 goto L4
a = 9
goto L3
L4:
goto L1
L2:
function end


Parsing Complete


--------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
--------------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
--------------------------------------------------------------------------------------------------

a       int     -               0                       0               0               1
b       int     -               0                       0               0               1
main    int     -               0                       1               0               0
$
```

3.6 test6.c

```c
#include <stdio.h>

void sum(int a)
{
    int c=0;

    while(a>7)
    {
      b=6;
      while(b>=5)
      {
        a=9;
      }
    }
    return;
}

int main ( )
```

```
{
  int a=5;
      int b=6;
  while(a>7)
  {
    b=6;
    while(b>=5)
    {
      a=9;
    }
  }
      sum(a,b);
  return 0;
}
```

```
$ ./a.out test6.c

function begin sum:
c = 0
L1:

Error: Undeclared variable at line 7


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------

c       int     -               0                       0               0               1

Parsing Failed
$
```

3.7 test7.c

```
#include <stdio.h>

int main()
{
      int a=5;
      int b=6;
      while(a>7)
      {
        if(a>9)
            {
                  b=b*8;
                  b=9;
```

```
            }
            else
            {
                  a=10;
            }
      }
      return 0;
}
```

```
$ ./a.out test7.c

function begin main:
a = 5
b = 6
L1:
t0 = a > 7
t1 = not t0
if t1 goto L2
t2 = a > 9
t3 = not t2
if t3 goto L3
t4 = b * 8
b = t4
b = 9
goto L4
L3:
a = 10
L4:
goto L1
L2:
function end


Parsing Complete


------------------------------------------------------------------------------------------------
                                SYMBOL TABLE
------------------------------------------------------------------------------------------------

Token   Type    ParameterType   No of Parameters        isFunction      Value           Scope
------------------------------------------------------------------------------------------------

a       int     -               0                       0               0               1
b       int     -               0                       0               0               1
main    int     -               0                       1               0               0
$
```

4. Implementation

Implementation of Three Address Code –

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple –

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

5. Results

- Generated three address code for the original grammar.
- Tested the compiler by generating assembly code which was be assembled, linked and executed to produce output.

7. References

- http://dinosaur.compilertools.net/yacc/
- https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf
- Compilers Principles Techniques and Tools book