Intermediate Code Generator for C language

Project - 4

28th March 2019

Submitted to :
Dr. P. Santhi Thilagam
Faculty, Dept of Computer Science and Engineering
National Institute of Technology Karnataka, Surathkal

Group members :
Ayush Kumar - 16CO208
Gauri Baraskar - 16CO209
Sanjana Krishnam - 16CO139

Abstract

Objective

The main goal of this project is to design a mini compiler for a subset of the C language as part of the Compiler Design Lab(CO351) course. The compiler is to be built in four phases finishing at the Intermediate Code Generation Phase. The subset of the C language chosen is to include certain data types, constructs and functions as mentioned in the specifications below. The implementation will be carried out with LEX and YACC.

Phases of the project

- Implementation of Scanner/Lexical Analyser
- Implementation of Parser
- Implementation of Semantic Checker for C language
- Intermediate Code generation for C language

Mini-compiler Specifications

The compiler is going to support the following cases :

1. **Keywords** - int, break, continue, else, for, void, goto, char, do, if, return, while
2. **Identifiers** - identified by the regular expression (_ |{letter})({letter}|{digit}| _){0,31}
3. **Comments** - single line comments (specified with // or /* … */), multi-line comments (specified with /* … */)
4. **Strings** - can identify strings mentioned in double quotes
5. **Preprocessor directives** - can identify filenames (stdio.h) after #include
6. **Data types** - int,char (supports comma declaration)
    - Syntax:
        - int a=5;
        - Char newchar = ‘a’
7. **Arrays** - int A[n]
    - Syntax:
        - int A[3];
        - Array initialization through for loops
8. **Punctuators** - [ ] , <> , {} , , , : , = , ; , # , ” ” , ’ ’
9. **Operators** - arithmetic (+, - , * , /) ,increment(++) and decrement(--), assignment (=)
10. **Condition** - if else
    - Syntax:
        - if (condition == true){
            //code

}
else{
//code }

11. **Loops -**
    - Syntax
        - while(condition){
          //code
          }
        - for(initialization;condition;increment/decrement){
          //code
          }
        - while(condition){
          //code
          }
        - do{
          //code
          }while(condition);

    The loop control structures that are supported are break,continue and goto.

12. **Functions-**
    Void functions with no return type and a single parameter will be implemented
    - Syntax
        - void sample_function(int a){
          //code
          }

The mini compiler will be implemented in a straightforward fashion , using Lex and YACC tools starting off with the Scanner as the first module. If time permits we will add more features to cover a larger subset of the C language.

Contents

List of Figures and Tables

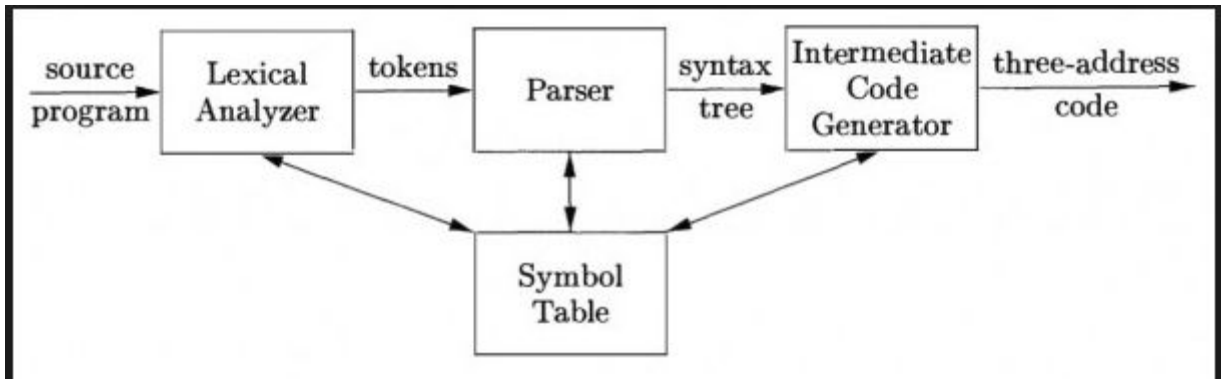
Introduction

Intermediate Code Generator

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation , from which the back end generates target code. Ideally details of the source language are confined to the front end , and the details of the target machine to the back end. With a suitably defined intermediate representation , a compiler for language i and machine j can then be built by combining the front end of language and i and back end of machine j. This tackles the need the create a native compiler for each language and machine.

During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as intermediate code or intermediate text. The complexity of this code lies between the source language code and the object code. The intermediate code can be represented in the form of postfix notation, syntax tree, directed acyclic graph (DAG), three-address code, quadruples, and triples.

Here in this project we have represented the code in the form of three-address code.The compiler phases can be summarised as follows -



The above figure shows the order of phases along with how all of them make use of the symbol table.

Yacc Script

Yacc stands for Yet Another Compiler-Compiler. 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.

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 is used to define any parameters for the C program, any header files to be included and global variable declarations. We also define all parameters related to the parser here, specifications like using Leftmost derivations or Rightmost derivations, precedence and left right associativity are declared here, data types and tokens which will be used by the lexical analyser are also declared in this stage. We also declare all the variables and types associated for non-terminals to assist with semantic checking.

The Rules section contains the entire grammar which is used for deciding if the input text is legally correct according to the specifications of the language. It also consists of code blocks with semantic rules. Yacc uses this rules for reducing the token stream received from the scanner, all rules are linked to each other from the start state, which is declared in the rules section.

In the C code section the parser is called, and the symbol table and constant tables are initialised in this section. After parsing and semantic checking the tables are filled with nesting level, function flag, array flag, array dimension, etc. The lex.yy.c file created by the lex script is also included from here which the parser calls. In this section we also define the error function used by the parser/semantic checker to report syntactical and semantic errors along with line numbers.

C Program

The parser(yacc script) takes C source files as input for parsing and semantic checking, the files are supplied using command line arguments, the path is given as argument.

For testing the parser the following commands have to be executed

```
lex lexer_parser.l
yacc -d parser.y
gcc y.tab.c -ll -ly
./a.out testcases/test-file-name.c
```

Under test cases , we have included test files in the form of C programs without errors so that the intermediate code can be generated.

Design of Programs

Lex Code

```
  /* This file describes the rules for the lexer */
%{
    #include<stdio.h>
    #include<stdlib.h>
    #include<limits.h>
    int ErrFlag = 0;

    extern int is_declaration;
    extern int is_param;

    #include "y.tab.h"
%}


 /* Declarations */

letter [a-zA-Z]
digit [0-9]
whitespace [ \t\r\f\v]+
identifier (_|{letter})({letter}|{digit}|_)*
hex [0-9a-fA-F]


 /* States */

%x PREPROCESSOR
%x MACROPREPROCESSOR
%x COMMENT
%x SLCOMMENT

%%


 /* Keywords */

"int"                               {return INT;}
"short"                             {return SHORT;}
"long"                              {return LONG;}
"char"                              {return CHAR;}
"void"                              {return VOID;}
"if"                                {return IF;}
"else"                              {return ELSE;}
```

```
"for"                                {return FOR;}
"do"                                 {return DO;}
"while"                              {return WHILE;}
"goto"                               {return GOTO;}
"break"                              {return BREAK;}
"continue"                           {return CONTINUE;}
"return"                             {return RETURN;}
"float"                              {return FLOAT;}


 /* Constants */

[0][x|X]{hex}+                        {yylval.tbEntry  =  InsertEntry(ConstantTable,  yytext  ,
(int)strtol(yytext, NULL, 16),"HEX",yylineno,0);return HEX_CONSTANT;}
{digit}+                            {yylval.tbEntry = InsertEntry(ConstantTable, yytext , (int)
atoi(yytext),"INT",yylineno,0);return INT_CONSTANT;}
({digit}*)["."]({digit}+)           {yylval.tbEntry  =  InsertEntry(ConstantTable,  yytext  ,
atof(yytext),"FLOAT",yylineno,0);return DEC_CONSTANT;}
({digit}+)["."]({digit}*)           {yylval.tbEntry  =  InsertEntry(ConstantTable,  yytext  ,
atof(yytext),"FLOAT",yylineno,0);return DEC_CONSTANT;}

{identifier} {
  if(strlen(yytext) <= 32)
    {
      yylval.str = yytext;
      if(is_declaration)
      {
        int nesting;
        if(is_param)
        {
          is_param = 0;
          nesting = curr_nest_level + 1;
        }
        else
        {
          nesting = curr_nest_level;
        }
        yylval.tbEntry = InsertEntry(SymbolTable,yytext,0,curr_data_type,yylineno,nesting);
      }
      return IDENTIFIER;
    }
  else
  {
          printf("Error  %d:  Identifier  too  long,must  be  between  1  to  32  characters\n",
yylineno);
      ErrFlag = 1;
  }
}
```

```
{digit}+({letter}|_)+                    {printf("Error %d: Illegal identifier format\n",
yylineno); ErrFlag = 1;}
{whitespace}                        ;

/* Preprocessor Directives */

^"#include"                         {BEGIN PREPROCESSOR;}
<PREPROCESSOR>{whitespace}          ;
<PREPROCESSOR>"<"[^<>\n]*">"        {BEGIN INITIAL;}
<PREPROCESSOR>\"[^<>\n]*\"          {BEGIN INITIAL;}
<PREPROCESSOR>"\n"                  { yylineno++; BEGIN INITIAL; ErrFlag=1;}
<PREPROCESSOR>.                     {yyerror("Improper Header");}


 /* Macropreprocessor Directives */

^"#define"                                      {BEGIN MACROPREPROCESSOR;}
<MACROPREPROCESSOR>{whitespace}                 ;
<MACROPREPROCESSOR>({letter})({letter}|{digit})*  {BEGIN INITIAL;}
<MACROPREPROCESSOR>\n                           {yylineno++; BEGIN INITIAL;}
<MACROPREPROCESSOR>.                            {BEGIN INITIAL;ErrFlag=1;}


 /* Comments */

"/*"                                {BEGIN COMMENT;}
<COMMENT>.|{whitespace}             ;
<COMMENT>\n                         {yylineno++;}
<COMMENT>"*/"                       {BEGIN INITIAL;}
<COMMENT>"/*"                       {yyerror("Improper Comment");yyterminate();}
<COMMENT><<EOF>>                    {yyerror("Improper Comment");yyterminate();}
"//"                                {BEGIN SLCOMMENT;}
<SLCOMMENT>.                        ;
<SLCOMMENT>\n                       {yylineno++; BEGIN INITIAL;}


 /* Operators */

"+"                                 {return ADD;}
"-"                                 {return SUBTRACT;}
"*"                                 {return MULTIPLY;}
"/"                                 {return DIVIDE;}
"%"                                 {return MOD;}
"="                                 {return ASSIGN;}
"--"                                {return DECREMENT;}
"++"                                {return INCREMENT;}

"+="                                {return ADD_ASSIGN;}
"-="                                {return SUB_ASSIGN;}
"*="                                {return MUL_ASSIGN;}
```

```
"/="                                {return DIV_ASSIGN;}
"%="                                {return MOD_ASSIGN;}

">"                                 {return GREATER_THAN;}
"<"                                 {return LESSER_THAN;}
">="                                {return GREATER_EQ;}
"<="                                {return LESS_EQ;}
"=="                                {return EQUAL;}

"||"                                {return LG_OR;}
"&&"                                {return LG_AND;}
"!"                                 {return NOT;}
"!="                                {return NOT_EQ;}


 /* Strings and Characters */

\"[^\"\n]*$                         {ErrFlag=1; yyterminate();}
\"[^\"\n]*\" {
  if(yytext[yyleng-2]=='\\') {
    yyless(yyleng-1);
    yymore();
  }
  else
  {
    yylval.tbEntry = InsertEntry(ConstantTable,yytext,yylval.dval,"CHAR",yylineno,0);
    return STRING;
  }
}

\'[^\'\n]\'                                                         {  yylval.tbEntry  =
InsertEntry(ConstantTable,yytext,yylval.dval,"CHAR",yylineno,0); return STRING;}


 /* Punctuators */


"\n"                                {yylineno++;}
.                                   {return yytext[0];}

%%
```

There is no change in lex code with respect to identifying of tokens. But while inserting constants and variables it stores the pointer to the location of constant or the variable in yylval which is passed to the parser. Thus parser , lexical analyser , and semantic analyser and intermediate

code generator share the symbol table. The script still identifies lexical errors and reports them to the parser which then throws a syntax error.

Parser,Semantic Analyser and Intermediate Code Generator

```
/* The file describes the grammar for our parser */
%{
    // Header files
    #include<stdio.h>
       #include<stdlib.h>
       #include "tables.h"
    #include<limits.h>
    #include<ctype.h>
    #include<string.h>

    // Initialising Symbol table and constant table
    entry **SymbolTable = NULL;
    entry **ConstantTable = NULL;

    int yyerror(char *msg);
    int checkScope(char *value);
    int typeCheck(entry*,entry*,char*);
    int checkFunc(char*);
    char* curr_data_type;
    int yylex(void);
    int is_loop = 0;
    int curr_nest_level = 1;
    int return_exists = 0;
    int is_param = 0;
    extern int yylineno;
    extern char* yytext;


    int is_declaration = 0;
    int is_function = 0;
    char* func_type;
    char *param_list[10];
     char *arg_list[10];

    int p_idx = 0;
    int a_idx = 0;
    int p=0;

    int stack[100] = {0};
    int top = 0;

    char ICGstack[100][20];
```

```
    int ICGtop = 0;

    double ICGvaluestack[100];
    int ICGvaluetop = 0;

    int Labelstack[10];
    int Labeltop = 1;

    int is_call = 0;

    void push(char *text);

    void gencode();
    void gencode_unary();
    void gencode_if_statement();
    void gencode_if_if();
    void gencode_if_else();
    void gencode_while();
    void gencode_while_block();
    void gencode_for_modif();
    void gencode_for_eval();
    void gencode_for();
    void gencode_function(char *lexeme);
    void gencode_return();
    void gencode_param();
    void gencode_array(char *data_type);
    void gencode_doWhile_eval();
    void gencode_doWhile_block();

    int Registerlabel = 0;
    int line_instruction = 0;

    int Declarationlabel = 0;
    double valAssign;

    FILE *output;

    int is_for = 0;

    int loop_constants[2];
    int temporary = 0;
    int is_array = 0;

%}

// Data types of tokens
%union{
        char *str;
        entry *tbEntry;
        double dval;
```

```
}

/* Operators */
%token ADD SUBTRACT MULTIPLY DIVIDE ASSIGN ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN
MOD_ASSIGN  MOD
/* Relational Operators */
%token GREATER_THAN LESSER_THAN LESS_EQ GREATER_EQ NOT_EQ EQUAL
/* Keywords */
%token VOID IF ELSE FOR DO WHILE GOTO BREAK CONTINUE RETURN
/* Data types */
%token INT SHORT LONG CHAR FLOAT
/* Logical Operators */
%token LG_OR LG_AND NOT
/* Assignment Operators */
%token DECREMENT INCREMENT
/* Constants */
%token <tbEntry> HEX_CONSTANT DEC_CONSTANT INT_CONSTANT STRING
/* Identifier */
%token <tbEntry> IDENTIFIER


 %type <tbEntry> identifier varDecId
 %type <tbEntry> mutable call factor expression simpleExpression andExpression
sumExpression unaryExpression unaryRelExpression term immutable relExpression
const_type


// Start Symbol of the grammar
%start program

/* Precedence of Operators */
%left ','
%right ASSIGN
%left LG_OR
%left LG_AND
%left EQUAL NOT_EQ
%left LESSER_THAN GREATER_THAN LESS_EQ GREATER_EQ
%left ADD SUBTRACT
%left MULTIPLY DIVIDE MOD
%right NOT


%nonassoc IFX
%nonassoc ELSE
%%

    program : declarationList;

    declarationList : declarationList declaration | declaration;
```

```
    declaration : varDeclaration | funDeclaration ;

    varDeclaration : typeSpecifier varDeclList ';' {is_declaration = 0;} ;

    varDeclList : varDeclList ',' varDeclInitialize | varDeclInitialize;

    varDeclInitialize : varDecId | varDecId assign_symbol simpleExpression {gencode();
$1->value = valAssign; typeCheck($1,$3,"="); is_declaration=1;} ;
    varDecId : identifier {$$=$1;} | identifier '[' INT_CONSTANT  ']' { if($3->value <
1){yyerror("Arrays can't have dimension lesser than 1"); return -1;} $$=$1;
$1->is_array = 1; $1->array_dim = (int)$3->value;};
    typeSpecifier : typeSpecifier pointer
                  | INT {curr_data_type = strdup("INT");  is_declaration = 1; }
                  | VOID {curr_data_type = strdup("VOID");  is_declaration = 1; }
                  | CHAR {curr_data_type = strdup("CHAR");  is_declaration = 1;}
                        | FLOAT {curr_data_type = strdup("FLOAT");  is_declaration =
1;}
                  ;

    assign_symbol : ASSIGN {is_declaration = 0; push("=");}

    pointer : MULTIPLY pointer | MULTIPLY;


    funDeclaration : typeSpecifier
                     identifier             {

                                                func_type = curr_data_type;
                                                            is_declaration = 0;

                                            }
                     '(' params ')'         {

fill_parameter_list($2,param_list,p_idx);
                                                p_idx = 0;
                                                is_function = 1;
                                                            int flag =
set_is_function(SymbolTable,$2->lexeme);
                                                if(flag == 0){return -1;}
                                                p=1;
                                                gencode_function($2->lexeme);
                                            }
                     compoundStmt           { is_function = 0;
                                              if(!return_exists &&
strcmp(func_type,"VOID") != 0)
                                              {
                                                printf("Warning : Function %s must have
a return type\n",$2->lexeme);

                                              }
```

```
                                                fprintf(output,"func end\n");
                                            };

     // Rules for parameter list
    params : paramList | ;
    paramList : paramList ',' paramTypeList | paramTypeList;
    paramTypeList : {is_param = 1; } typeSpecifier

                        paramId   {
                                                param_list[p_idx] = (char
*)malloc(sizeof(curr_data_type));

strcpy(param_list[p_idx++],curr_data_type);
                                                is_declaration = 0;

                                    };
    paramId : identifier | identifier '[' ']';
    // Types os statements in C
    statement : expressionStmt  | compoundStmt  | selectionStmt | iterationStmt |
jumpStmt | returnStmt | breakStmt | varDeclaration ;

    // compound statements produces a list of statements with its local declarations
    compoundStmt : {curr_nest_level++; stack[top] = curr_nest_level; top+=1;
insertNestStart(curr_nest_level,yylineno);} '{' statementList '}' {curr_nest_level++;
insertNestEnd(stack[top-1],yylineno);top-=1;};
    statementList : statementList statement
                  |  ;
    // Expressions
    expressionStmt : expression ';' {ICGtop = 0;} | ';' {ICGtop=0;} ;
    selectionStmt : IF '(' simpleExpression ')' {gencode_if_statement();} compoundStmt
else
                  ;

    else : {gencode_if_else();}ELSE statement  | ;

    iterationStmt : WHILE '(' simpleExpression ')' {gencode_while();} {is_loop = 1;}
statement { gencode_while_block(); is_loop = 0;}
                  | DO {is_loop = 1;}  {gencode_doWhile_block();}statement  WHILE '('
expression ')'  ';' {is_loop = 0; {gencode_doWhile_eval();}}
                  | FOR '(' optExpression ';' optExpression {gencode_for_eval();} ';'
{is_for = 1;} optExpression {is_for = 0;}  ')'{is_loop = 1;} statement {is_loop =
0;gencode_for_modif();};
    // Optional expressions in case of for
    optExpression : expression | ;

    jumpStmt : GOTO identifier ';' | CONTINUE ';' {if(!is_loop) {yyerror("Illegal use
of continue"); return -1;} fprintf(output,"goto L%d\n",loop_constants[1]); };
    returnStmt : RETURN ';'  { if(is_function) { if(strcmp(func_type,"VOID")!=0)
yyerror("return type (VOID) does not match function type"); return -1;}}
```

```
                | RETURN expression {
                                        return_exists = 1;
                                        if(strcmp(func_type,$2->data_type)!=0)
                                        {
                                          yyerror("return type does not match function
type");
                                          return -1;
                                        }
                                        else
                                          gencode_return();
                                      } ;
    breakStmt : BREAK ';' {if(!is_loop) {yyerror("Illegal use of break"); return -1;}
fprintf(output,"goto L%d\n",loop_constants[0]); };

    expression : mutable ASSIGN {push("=");}  expression {typeCheck($1,$4,"=");$$ =
$1;gencode(); $1->value = valAssign;}
                | mutable ADD_ASSIGN {push("+=");} expression {typeCheck($1,$4,"+=");$$
= $1;gencode();$1->value = valAssign;}
                | mutable SUB_ASSIGN {push("-=");} expression {typeCheck($1,$4,"-=");$$
= $1;gencode();$1->value = valAssign;}
                | mutable MUL_ASSIGN {push("*=");} expression {typeCheck($1,$4,"*=");$$
= $1;gencode(); $1->value = valAssign;}
                | mutable DIV_ASSIGN {push("/=");} expression {typeCheck($1,$4,"/=");$$
= $1;gencode(); $1->value = valAssign;}
                | mutable INCREMENT {typeCheck($1,$1,"++"); $$ = $1; $1->value =
$1->value + 1; fprintf(output,"%s = %s + 1\n",$1->lexeme,$1->lexeme);}
                | mutable DECREMENT {typeCheck($1,$1,"--"); $$ = $1; $1->value =
$1->value - 1; fprintf(output,"%s = %s - 1\n",$1->lexeme,$1->lexeme);}
                | simpleExpression { $$ = $1;}
                ;

    simpleExpression : simpleExpression LG_OR  andExpression {typeCheck($1,$3,"||"); $$
= $1;{push("||");}gencode();}
                     | andExpression { $$ = $1;};

    andExpression : andExpression LG_AND  unaryRelExpression { typeCheck($1,$3,"&&");$$
= $1;{push("&&");}gencode();}
                  | unaryRelExpression { $$ = $1;} ;

    unaryRelExpression : NOT unaryRelExpression { typeCheck($2,$2,"!u");$$ =
$2;{push("!");} }
                       | relExpression { $$ = $1;} ;

    relExpression : sumExpression GREATER_THAN  sumExpression {typeCheck($1,$3,">");$$
= $1;{push(">");}gencode();}
                  | sumExpression LESSER_THAN  sumExpression {typeCheck($1,$3,"<");$$ =
$1;{push("<");}gencode();}
                  | sumExpression LESS_EQ  sumExpression {typeCheck($1,$3,"<=");$$ =
$1;{push("<=");}gencode();}
                  | sumExpression GREATER_EQ  sumExpression {typeCheck($1,$3,">=");$$ =
```

```
$1;{push(">=");}gencode();}
                  | sumExpression NOT_EQ  sumExpression
{typeCheck($1,$3,"!=");{push("!=");} gencode();}
                  | sumExpression EQUAL  sumExpression {typeCheck($1,$3,"==");$$ = $1;
{push("==");} gencode();}
                  | sumExpression { $$ = $1;}
                  ;
    sumExpression : sumExpression ADD  term {typeCheck($1,$3,"+");$$ =
$1;{push("+");}gencode();}
                  | sumExpression SUBTRACT term {typeCheck($1,$3,"-");$$ =
$1;{push("-");} gencode(); }
                  | term { $$ = $1;}
                  ;



    term : term MULTIPLY  unaryExpression {typeCheck($1,$3,"*");$$ = $1;
{push("*");}gencode();}
         | term DIVIDE  unaryExpression {typeCheck($1,$3,"/");$$ = $1;
{push("/");}gencode();}
         | unaryExpression { $$ = $1;}
         ;

    unaryExpression : ADD   factor { typeCheck($2,$2,"+u");$$ = $2;
{push("+");}gencode_unary();}
                    | SUBTRACT  factor { typeCheck($2,$2,"-u");$$ = $2;
{push("-");}gencode_unary();}
                    | factor { $$ = $1;} ;


    factor : immutable {$$ = $1;} | mutable {$$ = $1;};
    mutable : identifier {if(checkScope(yylval.str) == 0){ return -1;}; $$ = $1;}|
identifier '[' {is_array=1;}simpleExpression {is_array=0;} ']' {if($4->value < 0 ||
$4->value >= $1->array_dim ){yyerror("Exceeds Array Dimensions\n"); return -1; } $$ =
$1; gencode_array($1->data_type);}
    immutable : '(' expression ')' { $$ = $2;}| call {$$=$1;} | const_type {$$=$1;} ;
    call : identifier  '(' args ')' {
                                      if(checkFunc($1->lexeme) == 0)
                                        {return -1;};
                                      $$ = $1;
                                      check_parameter_list($1,arg_list,a_idx);
                                      a_idx = 0;
                                      fprintf(output,"refparam result\n");
                                      {fprintf(output,"call
%s,%d\n",$1->lexeme,$1->num_params);}
                                      is_call = 1;
                                    }
    args : argList  | ;

    argList : argList ',' arg
```

```
            | arg ;

    arg : expression     {
                            arg_list[a_idx] = (char *)malloc(sizeof($1));
                            strcpy(arg_list[a_idx++],$1->data_type);

                            gencode_param();

                         }
        ;

    const_type : DEC_CONSTANT { $$ = $1;push($1->lexeme);}
               | INT_CONSTANT { $$ = $1;push($1->lexeme);}
               | HEX_CONSTANT { $$ = $1;push($1->lexeme);}
               | STRING       { $$ = $1;push($1->lexeme);}
               ;
    identifier : IDENTIFIER {
                                    if(is_declaration){
                                    // $1 =
InsertEntry(SymbolTable,yytext,INT_MAX,curr_data_type,yylineno,curr_nest_level);
                                    $$ = $1;
                    is_function = 0;
                    push($1->lexeme);
                                    }
                                    else
                                    {
                                    $1 = Search(SymbolTable,yytext);
                                    $$ = $1;
                    if($1 == NULL)
                    {
                        yyerror("Variable Not Declared");
                        return -1;
                    }
                    push($1->lexeme);
                                    }
                          };
%%


int checkFunc(char* lexeme)
{
    entry *res = searchFunc(SymbolTable,lexeme);
    if(res != NULL)
    {
        res = InsertSearch(SymbolTable,lexeme,curr_nest_level);
        if(res != NULL)
        {
            yyerror("Defined as variable in this scope, calling not allowed");
            return 0;
        }
```

```
        else
        {
            return 1;
        }

    }
    else
    {
        yyerror("No such declaration\n");
        return 0;
    }
}

int typeCheck(entry *a,entry *b,char *c){

       if(strcmp(a->data_type,b->data_type)!=0){
              yyerror("Type Mismatch");
              exit(0);
       }

       else
              return 1;
}
void disp()
{
    printf("\n\tSymbol table");
    Display(SymbolTable);
    // printf("\n\tConstant table");
    //DisplayConstant(ConstantTable);

    printf("\n\n_____________________________________________\n\n");
    printf("\t Intermediate Code Generation\n\n");
    printf("_____________________________________________\n\n");
    system("cat ICG.code");
    printf("\n_____________________________________________\n");

}

int checkScope(char *val)
{

    char *extract = (char *)malloc(sizeof(char)*32);
    int i;
    // Don't touch this CRUCIAL AS FUCK
    for(i = 0; i < strlen(val) ;i=i+1)
    {
        //printf("%d\n",i);
        if((isalnum(*(val + i)) != 0) || (*(val + i)) == '_')
        {
            *(extract + i) = *(val + i);
```

```
            }
            else
            {
                *(extract + i) = '\0';
                break;
            }
        }
        entry *res = ScopeSearch(SymbolTable,extract,curr_nest_level);
        // First check if variable exists then check for nesting level
        if (res == NULL)
        {
            yyerror("Variable Not Declared\n");
            exit(0);
            return 0;
        }
        else
        {
            int level = res->nesting_level;
            int endLine = -1;
            int startLine = -1;
            if(Nester[level] == NULL)
            {
                startLine = 0;
                endLine = yylineno + 100;
            }
            else
            {
                startLine = Nester[level]->line_start;
                endLine = Nester[level]->line_end;
            }


            if((yylineno <= endLine && yylineno >= startLine))
            {

                return 1;
            }
            else
            {

                yyerror("Variable Out Of Scope\n");
                exit(0);
                return 0;
            }
        }

    }

    #include "lex.yy.c"
    int main(int argc , char *argv[]){
```

```
    system("clear");

    SymbolTable = CreateTable();
    ConstantTable = CreateTable();
    nested_homekeeping();

    // Open a file for parsing
    yyin = fopen(argv[1], "r");

    output = fopen("ICG.code","w");

    if(!yyparse())
    {
        printf("\nParsing complete.\n");
        fprintf(output,"exit\n");
        fclose(output);
        fclose(yyin);
        disp();
        return 1;

    }
    else
    {
        printf("\nParsing failed.\n");
    }
    fclose(yyin);
    fclose(output);
    // system("clear");

    return 0;
}


int yyerror(char *msg)
{
  // Function to display error messages with line no and token
    printf("Line no: %d Error message: %s Token: %s\n", yylineno, msg, yytext);
    return 0;
}

void push(char *text)
{
    strcpy(ICGstack[ICGtop++],text);
}

void pushvalue(double text)
{
    ICGvaluestack[ICGvaluetop++] = text;
}
```

```
void gencode()
{


    if(is_for == 1)
    {
        gencode_for();
    }
    else
    {

    char *op1 = ICGstack[--ICGtop];
    char *op2 = ICGstack[--ICGtop];
    char *op3 = ICGstack[--ICGtop];

    if( strcmp(op2,"=")== 0)
    {
        if(is_call == 0)
        {
            fprintf(output,"%s = %s\n",op3,op1);
            if(isalpha(op1[0])){
              if(ScopeSearch(SymbolTable,op1,curr_nest_level)==NULL)
                valAssign = ICGvaluestack[--ICGvaluetop];
              else
                valAssign = ScopeSearch(SymbolTable,op1,curr_nest_level)->value;
            }
            else
              valAssign = atof(op1);
            is_call = 0;

        }
        else
        {
           fprintf(output,"%s = result\n",op3);
           --ICGtop;
        }
    }
    else if(strcmp(op2,"+=") == 0)
    {
       if(isalpha(op1[0])){
              if(ScopeSearch(SymbolTable,op1,curr_nest_level)==NULL)
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value +
ICGvaluestack[--ICGvaluetop];
              else
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value +
ScopeSearch(SymbolTable,op1,curr_nest_level)->value;
            }
            else
        {
```

```
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value +
atof(op1);
        }
        fprintf(output,"%s = %s + %s\n",op3,op3,op1);

    }
    else if(strcmp(op2,"-=") == 0)
    {
       if(isalpha(op1[0])){
              if(ScopeSearch(SymbolTable,op1,curr_nest_level)==NULL)
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value -
ICGvaluestack[--ICGvaluetop];
              else
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value -
ScopeSearch(SymbolTable,op1,curr_nest_level)->value;
            }
            else
        {
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value -
atof(op1);
        }
        fprintf(output,"%s = %s - %s\n",op3,op3,op1);
    }
    else if(strcmp(op2,"*=") == 0)
    {
       if(isalpha(op1[0])){
              if(ScopeSearch(SymbolTable,op1,curr_nest_level)==NULL)
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value *
ICGvaluestack[--ICGvaluetop];
              else
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value *
ScopeSearch(SymbolTable,op1,curr_nest_level)->value;
            }
            else
        {
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value *
atof(op1);
        }
        fprintf(output,"%s = %s * %s\n",op3,op3,op1);
    }
    else if(strcmp(op2,"/=") == 0)
    {
       if(isalpha(op1[0])){
              if(ScopeSearch(SymbolTable,op1,curr_nest_level)==NULL)
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value /
ICGvaluestack[--ICGvaluetop];
              else
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value /
ScopeSearch(SymbolTable,op1,curr_nest_level)->value;
            }
```

```
        else
        {
              valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value /
atof(op1);
        }
        fprintf(output,"%s = %s / %s\n",op3,op3,op1);
    }
    else
    {

        char temp[3] = "t0\0";
        temp[1] = (char)(Registerlabel + '0');
        temp[2] = '\0';
        Registerlabel++;
        double var3,var2;

        if(isalpha(op3[0])){
              if(ScopeSearch(SymbolTable,op3,curr_nest_level)==NULL)
                var3 = ICGvaluestack[--ICGvaluetop];
              else
                var3 = ScopeSearch(SymbolTable,op3,curr_nest_level)->value;
        }
        else
          var3 = atof(op3);

        if(isalpha(op2[0])){
              if(ScopeSearch(SymbolTable,op2,curr_nest_level)==NULL)
                var2 = ICGvaluestack[--ICGvaluetop];
              else
                var2 = ScopeSearch(SymbolTable,op2,curr_nest_level)->value;
        }
        else
          var2 = atof(op2);

        fprintf(output,"%s = %s %s %s\n",temp,op3,op1,op2);
        if(strcmp(op1,"+") == 0)
            valAssign = var3 + var2;
        else if(strcmp(op1,"-") == 0)
            valAssign = var3 - var2;
        else if(strcmp(op1,"*") == 0)
            valAssign = var3 * var2;
        else if(strcmp(op1,"/") == 0)
            valAssign = var3 / var2;

        push(temp);
        pushvalue(valAssign);

    }
    }
```

```
    line_instruction++;
}

void gencode_for()
{


    char *op1 = ICGstack[--ICGtop];
    char *op2 = ICGstack[--ICGtop];
    char *op3 = ICGstack[--ICGtop];

    FILE *output1 = fopen("modtemp.code","a");

    if( strcmp(op2,"=")== 0)
    {
        fprintf(output1,"%s = %s\n",op3,op1);
    }
    else if(strcmp(op2,"+=") == 0)
    {
       if(isalpha(op1[0])){
              if(ScopeSearch(SymbolTable,op1,curr_nest_level)==NULL)
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value +
ICGvaluestack[--ICGvaluetop];
              else
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value +
ScopeSearch(SymbolTable,op1,curr_nest_level)->value;
            }
        else
        {
               valAssign =  atof(op1);
        }
        fprintf(output1,"%s = %s + %s\n",op3,op3,op1);
    }
    else if(strcmp(op2,"-=") == 0)
    {
       if(isalpha(op1[0])){
              if(ScopeSearch(SymbolTable,op1,curr_nest_level)==NULL)
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value -
ICGvaluestack[--ICGvaluetop];
              else
                valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value -
ScopeSearch(SymbolTable,op1,curr_nest_level)->value;
            }
        else
        {
               valAssign = atof(op1);
        }
        fprintf(output1,"%s = %s - %s\n",op3,op3,op1);
    }
    else if(strcmp(op2,"*=") == 0)
```

```
    {
       if(isalpha(op1[0])){
             if(ScopeSearch(SymbolTable,op1,curr_nest_level)==NULL)
               valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value *
ICGvaluestack[--ICGvaluetop];
             else
               valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value *
ScopeSearch(SymbolTable,op1,curr_nest_level)->value;
           }
        else
        {
              valAssign = atof(op1);
        }
        fprintf(output1,"%s = %s * %s\n",op3,op3,op1);
    }
    else if(strcmp(op2,"/=") == 0)
    {
       if(isalpha(op1[0])){
             if(ScopeSearch(SymbolTable,op1,curr_nest_level)==NULL)
               valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value /
ICGvaluestack[--ICGvaluetop];
             else
               valAssign = ScopeSearch(SymbolTable,op3,curr_nest_level)->value /
ScopeSearch(SymbolTable,op1,curr_nest_level)->value;
           }
        else
        {
              valAssign = atof(op1);
        }
        fprintf(output1,"%s = %s / %s\n",op3,op3,op1);
    }
    else
    {

        char temp[3] = "t0\0";
        temp[1] = (char)(Registerlabel + '0');
        temp[2] = '\0';
        Registerlabel++;

        double var3,var2;

        if(isalpha(op3[0])){
             if(ScopeSearch(SymbolTable,op3,curr_nest_level)==NULL)
               var3 = ICGvaluestack[--ICGvaluetop];
             else
               var3 = ScopeSearch(SymbolTable,op3,curr_nest_level)->value;
        }
        else
          var3 = atof(op3);
```

```
            if(isalpha(op2[0])){
                  if(ScopeSearch(SymbolTable,op2,curr_nest_level)==NULL)
                    var2 = ICGvaluestack[--ICGvaluetop];
                  else
                    var2 = ScopeSearch(SymbolTable,op2,curr_nest_level)->value;
            }
            else
              var2 = atof(op2);

            if(strcmp(op1,"=") != 0)
            {
            fprintf(output1,"%s = %s %s %s\n",temp,op3,op1,op2);
            if(strcmp(op1,"+") == 0)
                valAssign = var3 + var2;
            else if(strcmp(op1,"-") == 0)
                valAssign = var3 - var2;
            else if(strcmp(op1,"*") == 0)
                valAssign = var3 * var2;
            else if(strcmp(op1,"/") == 0)
                valAssign = var3/var2;
            }
            push(temp);
            pushvalue(valAssign);

        }
        line_instruction++;
        fclose(output1);
    }

    void gencode_unary()
    {
        line_instruction++;
        char *op1 = ICGstack[--ICGtop];
        char *op2 = ICGstack[--ICGtop];

        char temp[3] = "t0\0";
        temp[1] = (char)(Registerlabel + '0');
        temp[2] = '\0';
        Registerlabel++;

        fprintf(output,"%s = %s %s\n",temp,op1,op2);

        push(temp);
        line_instruction++;
    }

    void gencode_if_statement()
    {
        Labelstack[Labeltop++] = ++Declarationlabel;
        fprintf(output,"if %s goto L%d\n",ICGstack[--ICGtop],Declarationlabel);
```

```
    gencode_if_if();

}

void gencode_if_if()
{
    ++Declarationlabel;
    fprintf(output,"goto L%d\n",Declarationlabel);
    fprintf(output,"L%d :\n",Labelstack[--Labeltop]);
    Labelstack[Labeltop++] = Declarationlabel;
}

void gencode_if_else()
{
    fprintf(output,"L%d :\n",Labelstack[--Labeltop]);
}

void gencode_while()
{

    fprintf(output,"L%d :\n",++Declarationlabel);
    loop_constants[0] = Declarationlabel;
    Labelstack[Labeltop++] = Declarationlabel;
    Labelstack[Labeltop++] = ++Declarationlabel;
    fprintf(output,"if %s goto L%d\n",ICGstack[--ICGtop],Declarationlabel);

    ++Declarationlabel;
    fprintf(output,"goto L%d\n",Declarationlabel);
    loop_constants[1]= Declarationlabel;
    fprintf(output,"L%d :\n",Labelstack[--Labeltop]);
    Labelstack[Labeltop++] = Declarationlabel;
}

void gencode_while_block()
{
    int l_otherwise = Labelstack[--Labeltop];
    int l_while = Labelstack[--Labeltop];
    fprintf(output,"goto L%d\n",l_while);
    fprintf(output,"L%d :\n",l_otherwise);
}

void gencode_for_eval()
{
    fprintf(output,"L%d :\n",++Declarationlabel);
    loop_constants[0] = Declarationlabel;
    Labelstack[Labeltop++] = Declarationlabel;
    Labelstack[Labeltop++] = ++Declarationlabel;
    fprintf(output,"if %s goto L%d\n",ICGstack[--ICGtop],Declarationlabel);
```

```
    ++Declarationlabel;
    fprintf(output,"goto L%d\n",Declarationlabel);
    loop_constants[1] = Declarationlabel;
    fprintf(output,"L%d :\n",Labelstack[--Labeltop]);
    Labelstack[Labeltop++] = Declarationlabel;
}

void gencode_for_modif()
{

    FILE *output1 = fopen("modtemp.code","r");

    char c = fgetc(output1);
    while(c != EOF)
    {
        fprintf(output,"%c",c);
        c = fgetc(output1);
    }

    fclose(output1);

    system("rm modtemp.code");

    int l_otherwise = Labelstack[--Labeltop];
    int l_for = Labelstack[--Labeltop];

    fprintf(output,"goto L%d\n",l_for);
    fprintf(output,"L%d :\n",l_otherwise);

}

void gencode_function(char *lexeme)
{
    fprintf(output,"func begin %s\n",lexeme);
}

void gencode_return()
{
    fprintf(output,"return %s\n",ICGstack[--ICGtop]);
}

void gencode_param()
{
    fprintf(output,"Param %s\n",ICGstack[--ICGtop]);
}
void gencode_array(char *data_type)
{

    char temp[3] = "t0\0";
        temp[1] = (char)(Registerlabel + '0');
```

```
            temp[2] = '\0';
            Registerlabel++;

        char temp1[3] = "t0\0";
            temp1[1] = (char)(Registerlabel + '0');
            temp1[2] = '\0';
            Registerlabel++;



        char *op1 = ICGstack[--ICGtop];
        char *op2 = ICGstack[--ICGtop];



        strcat(op2,"[");
        if(strcmp(data_type,"INT") == 0 || strcmp(data_type,"FLOAT")==0)
            fprintf(output,"%s = 4 * %s\n",temp,op1);
        strcat(op2,temp);
        strcat(op2,"]");

        fprintf(output,"%s = %s\n",temp1,op2);

        push(temp1);

        //fprintf(output,"ARRAY %s\n",ICGstack[--ICGtop]);
    }

    void gencode_doWhile_block()
    {
        fprintf(output,"L%d :\n",++Declarationlabel);
        loop_constants[0] = Declarationlabel;
        loop_constants[1] = Declarationlabel + 1;
    }

    void gencode_doWhile_eval()
    {
        fprintf(output,"if %s goto L%d\n",ICGstack[--ICGtop],loop_constants[0]);
        fprintf(output,"goto L%d\n",++Declarationlabel);
        fprintf(output,"L%d :\n",Declarationlabel);
    }
```

The above code is the yacc script semantic checking of the code. Code generation rules related to each production are written opposite to it in a block of code comprised between flower braces.

The below file is a tables.h which deals with functions related to entries within the tables.

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

#define SIZE 100

extern int yyerror(char *msg);
extern int curr_nest_level;
extern int is_function;
extern int yylineno;

struct table_entry{

    int line_number;
    char *lexeme;
    double value;
    char* data_type;
    int is_function;
    int is_array;
    int nesting_level;
    int array_dim;
    char* parameter_list[10];
    int num_params;
    int *arr_vals;
    struct table_entry *next;
};

struct nested{
    int nesting_level;
    int line_start;
    int line_end;
};

struct nested **Nester = NULL;

void nested_homekeeping()
{
  Nester = malloc(sizeof(struct nested*)*SIZE);

  int i;
  for(i = 0;i<SIZE;i++)
  {
    Nester[i] = NULL;
  }
  struct nested *temp = NULL;
```

```
  temp = malloc(sizeof(struct nested));
  temp->nesting_level = 1;
  temp->line_start = 0;
  temp->line_end = 999999;
  Nester[1] = temp;
}

void insertNestStart(int nesting_level,int line_start)
{
  struct nested *temp = NULL;
  temp = malloc(sizeof(struct nested));
  temp->nesting_level =  nesting_level;
  temp->line_start = line_start;
  temp->line_end = 999999;

  Nester[nesting_level] = temp;
}


void insertNestEnd(int nesting_level,int line_end)
{
  if(Nester[nesting_level] != NULL)
  {
    Nester[nesting_level]->line_end=line_end ;
  }
  else
  {
    return;
  }

}


typedef struct table_entry entry;

void Display(entry** TablePointer);
int hash(char *lexeme)
{
  int hash = 0,i=0;

  for(i=0; i < strlen(lexeme); i++)
    hash += lexeme[i];

  return hash % SIZE;
}

entry** CreateTable()
{
  entry **TablePointer = NULL;
```

```
  TablePointer = malloc(sizeof(entry*)*SIZE);

  if(TablePointer == NULL)
    return NULL;

  int i;
  for(i=0;i<SIZE;i++)
    TablePointer[i] = NULL;

  return TablePointer;
}

entry* searchFunc(entry** TablePointer,char *lexeme)
{
  int temp = hash(lexeme);
  entry *head = NULL;
  head = TablePointer[temp];
  //Display(TablePointer);
  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0 && head->is_function == 1)
    {
      return head;
    }
    else
      head = head->next;
  }
  if(head == NULL)
    return NULL;
  return head;
}

entry* Search(entry** TablePointer, char *lexeme)
{
  int temp = hash(lexeme);
  entry *head = NULL;
  head = TablePointer[temp];
  //Display(TablePointer);
  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0)
    {
      return head;
    }
    else
      head = head->next;
  }
  if(head == NULL)
    return NULL;
  return head;
```

```
}

entry* InsertSearch(entry** TablePointer, char *lexeme,int currScope)
{

  int temp = hash(lexeme);
  entry *head = NULL;
  head = TablePointer[temp];
  //Display(TablePointer);
  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0 && currScope == head->nesting_level)
    {
      return head;
    }
    else
      head = head->next;
  }
  if(head == NULL)
    return NULL;
  return head;
}

entry* ScopeSearch(entry** TablePointer, char *lexeme,int currScope)
{
  int temp = hash(lexeme);
  entry *head = NULL;
  head = TablePointer[temp];


  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0)
    {
      if(Nester[head->nesting_level]->line_end < yylineno)
      {
        head = head->next;
        continue;
      }
      else
      {
        return head;
      }
    }
    else
    {
      head = head->next;
    }

  }
```

```
  return NULL;
}

int funcSearch(entry** TablePointer,char *lexeme,int currLine)
{
  int temp = hash(lexeme);
  entry *head = NULL;
  entry **array = NULL;
  int raiseFlag = 0;

  array = malloc(sizeof(entry*)*SIZE);

  if(array == NULL)
    return 0;

  int i;
  for(i=0;i<SIZE;i++)
    array[i] = NULL;
  i = 0;
  head = TablePointer[temp];
  //Display(TablePointer);
  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0 )
    {
      array[i] = head;
      head=head->next;
    }
    else
      head = head->next;
  }
  for(i=0;i<SIZE;i++)
  {

    entry* tempPoint = array[i];
    if(tempPoint == NULL)
      continue;
    else if(tempPoint->is_function == 1)
    {
      raiseFlag = 1;
      break;
    }
  }
  if(raiseFlag == 1)
  {
    head = TablePointer[temp];
    entry* prev = NULL;

    while(head != NULL)
    {
```

```
      if(strcmp(head->lexeme,lexeme) == 0 && head->line_number == currLine)
      {
        if(prev != NULL)
        {
          prev->next = head->next;
          free(head);
        }
        else
        {
          TablePointer[temp] = NULL;
          free(head);
        }

        return 0;
      }
      else
      {
        prev = head;
        head = head->next;
      }

    }
    return 0;
  }
  else
  {
    return 1;
  }


}

int set_is_function(entry** TablePointer, char *lexeme)
{
       if(funcSearch(TablePointer,lexeme,yylineno))
  {
    entry* Entry = Search(TablePointer,lexeme);
         if (Entry == NULL)
           return 0;
         else
           Entry->is_function = 1;
    return 1;
  }
  else
  {
    yyerror("Duplicate Functions Not allowed\n");
    return 0;
  }
```

```
}



entry* InsertEntry(entry** TablePointer, char *lexeme,double value,char* DataType,int
line_number ,int nesting_level)
{
  int temp = hash(lexeme);
  if(InsertSearch(TablePointer,lexeme,curr_nest_level) != NULL)
  {
    yyerror("Duplicate Variable Declaration not allowed\n");
    return TablePointer[temp];
  }

  else
  {

    entry *head = NULL;

    head = TablePointer[temp];

    entry *tempPoint = NULL;
    tempPoint = malloc(sizeof(entry));
    tempPoint->lexeme = strdup(lexeme);
    tempPoint->value = value;
    tempPoint->data_type = strdup(DataType);
    tempPoint->line_number = line_number;
    tempPoint->nesting_level = nesting_level;
    tempPoint->next = NULL;

    if (head == NULL)
    {
      TablePointer[temp] = tempPoint;
    }
    else
    {
      tempPoint->next = TablePointer[temp];
      TablePointer[temp] = tempPoint;
    }

  }
  return TablePointer[temp];

}


void fill_parameter_list(entry* tableEntry, char **list, int n)
{
```

```
   int i;
   for(i=0; i<n; i++)
   {
    tableEntry->parameter_list[i] = (char *)malloc(sizeof(char));
    if(strcmp(list[i],"VOID") == 0)
    {
      yyerror("Parameters of type void not allowed\n");
      return;
    }
     strcpy(tableEntry->parameter_list[i],list[i]);
   }
   tableEntry->num_params = n;
}

int check_parameter_list(entry* tableEntry, char** list, int m)
{

       if(m != tableEntry->num_params)
       {
              yyerror("Number of parameters and arguments do not match");
    exit(0);
       }

       int i;
       for(i=0; i<m; i++)
       {
              if( strcmp(list[i],tableEntry->parameter_list[i]) !=0 ){
              yyerror("Parameter and argument types do not match");
    exit(0);
  }

       }

       return 1;
}


void Display(entry** TablePointer)
{
  int i =0;
  entry *temp = NULL;

  printf("\n\n");

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

  printf("\n\t(lexeme, \tData type, \tLine Number, \tisArray, \tArrayDimensions, \tisFunction,
\tNesting Level, \tnum_params\n");
```

```
  for(i=0;i<SIZE;i++)
  {
    temp = TablePointer[i];
    while(temp != NULL)
    {
      printf("\t(%s, \t\t%s, \t\t%d, \t\t%d, \t\t%d, \t\t\t%d, \t\t%d,
\t\t%d\t)\n",temp->lexeme,temp->data_type,temp->line_number,temp->is_array,temp->array_dim,temp
->is_function,temp->nesting_level, temp->num_params);
      int j;
      if(temp->num_params > 0)
      {
        printf("\tParameter List for %s\n",temp->lexeme);
        printf("\t( ");
        for(j=0; j < temp->num_params; j++)
        {
            printf("%s",temp->parameter_list[j]);
            printf("\t");
        }
        printf(" )\n");
      }

      temp = temp->next;
    }

  }

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

void DisplayConstant(entry** TablePointer)
{
  int i =0;
  entry *temp = NULL;

  printf("\n\n");

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

  printf("\n\t(lexeme,   value, Data type, Line Number\n");

  for(i=0;i<SIZE;i++)
  {
    temp = TablePointer[i];
    while(temp != NULL)
    {
      printf("\t(%5s, %5f, %9s,
%11d)\n",temp->lexeme,temp->value,temp->data_type,temp->line_number);
      temp = temp->next;
    }
}
```

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

The above code is the C code for tables.h which handles all functions relating to entries in the table. The functions used in the file are described in detail below -

1. **InsertEntry** is a function that inserts new variables and functions into the table. It checks for duplicate variables by carrying out a search and reports errors in case it finds a duplicate declaration.

2. Search Functions:

a) **Search** is a search function which searches for entries in Symbol Table based on the lexeme, it doesn't take any other criteria into consideration. This function is used to look for the most recent entry for the current lexeme, and is in employed in cases where only a simple lookup is required.

b) **InsertSearch** is a search function which searches for entries based on two criteria lexeme and current scope of the variable, this function is primarily used during declaration to check if any other variable with the same name exists in the current scope.

c) **funcSearch** is a function employed during declaration of functions, when a new function is declared we have to ensure that no other function with the same name exists in the program, and this function takes care of that, it searches the table for any other identifiers which might also be a function and then throws an error if the current declaration is a duplicate.

d) **searchFunc** is another utility function employed during function calls, it searches for any functions with the same name as the current function and enables appropriate actions in the caller function based on syntactical rules.

3. **set_is_function** is used to declare variables as functions if the declaration follows function declaration rules, it takes care of appropriate sanitization and sets is_function as 1 if the variable is a function.

4. **fill_parameter_list** is used to fill all the parameters for a declared function, this is done so that verification can be done at a later stage when the function is called. fill_parameter_list uses a linked list implementation to keep track of the declared parameters and then stores the linked list in the symbol table.

5. **check_parameter_list** is used to match the types of arguments and parameters. First the function matches the number of two and then the type.

Symbol table implementation

The above table contains all entries for variables and functions. The fields include name of the identifier, line number , boolean fields is_array , is_function and is_array (which are self explanatory) , data type , nesting level , array dimension , number of params and parameter list.

The implementation uses a hash table with open chaining. The constant table also uses the same structure.

Test Cases

Without Errors

Serial Number	Test Case	Expected Output	Status

1.	`#include <stdio.h>`
	`int main()`
	`{`
	`  int temp=5;`
	`  int arr[100];`
	`  arr[3]=2;`
	`  int k=arr[temp-3];`
	`  return 0;`
	`}`
		`func begin main`
		`temp = 5`
		`t0 = 4 * 3`
		`t1 = arr[t0]`
		`t1 = 2`
		`t2 = temp - 3`
		`t3 = 4 * t2`
		`t4 = arr[t3]`
		`k = t4`
		`return 0`
		`func end`
		`exit`
			PASS

2.	`#include <stdio.h>`
	`int main () {`
	`  int a = 10;`
	`  while( a < 20 ) {`
	`  a++;`
	`  if( a > 15) {`
	`  break;`
	`  }`
	`  }`
	`  return 0;`
	`}`
		`func begin main`
		`a = 10`
		`t0 = a < 20`
		`L1 :`
		`if t0 goto L2`
		`goto L3`
		`L2 :`
		`a = a + 1`
		`t1 = a > 15`
		`if t1 goto L4`
		`goto L5`
		`L4 :`
		`goto L1`
		`goto L3`
		`L5 :`
		`return 0`
		`func end`
		`exit`
		
		`func end`
		`exit`
			PASS

3.	`#include <stdio.h>`
	`int main () {`
	`  int a = 10;`
	`  do {`
	`  a = a + 1;`
	`  }while( a < 20 );`
	`  return 0;`
	`}`
		`func begin main`
		`a = 10`
		`L1 :`
		`t0 = a + 1`
		`a = t0`
		`t1 = a < 20`
		`if t1 goto L1`
		`goto L2`
		`L2 :`
		`return 0`
		`func end`
		`exit`
			PASS

4.	`#include <stdio.h>`
	`int main () {`
	`  int a;`
	`  int k;`
	`  for( a=10; a<20; a=a+1 ){`
	`  k = 1;`
	`  }`
	`  return 0;`
	`}`
		`func begin main`
		`a = 10`
		`t0 = a < 20`
		`L1 :`
		`if t0 goto L2`
		`goto L3`
		`L2 :`
		`k = 1`
		`t1 = a + 1`
		`a = t1`
		`goto L1`
		`L3 :`
		`return 0`
		`func end`
		`exit`
			PASS

5.	`#include <stdio.h>`
	`void sum(int a, int b)`
	`{`
	`  int c=0;`
	`  while(a>7)`
	`  {`
	`  b=6;`
	`  while(b>=5)`
	`  {`
	`  a=9;`
	`  }`
	`  }`
	`  return;`
	`}`
	
	`int main ( )`
	`{`
	`  int a=5;`
	`  int b=6;`
		`func begin sum`
		`c = 0`
		`t0 = a > 7`
		`L1 :`
		`if t0 goto L2`
		`goto L3`
		`L2 :`
		`b = 6`
		`t1 = b >= 5`
		`L4 :`
		`if t1 goto L5`
		`goto L6`
		`L5 :`
		`a = 9`
		`goto L4`
		`L6 :`
		`goto L1`
		`L3 :`
		`func end`
		`func begin main`
		`a = 5`
			PASS

	`while(a>7)`
	`{`
	`  b=6;`
	`  while(b>=5)`
	`  {`
	`  a=9;`
	`  }`
	`}`
	`  sum(a,b);`
	`return 0;`
	`}`
		b = 6
		t2 = a > 7
		L7 :
		if t2 goto L8
		goto L9
		L8 :
		b = 6
		t3 = b >= 5
		L10 :
		if t3 goto L11
		goto L12
		L11 :
		a = 9
		goto L10
		L12 :
		goto L7
		L9 :
		Param a
		Param b
		refparam result
		call sum,2
		return 0
		func end
		exit	

6.	`#include <stdio.h>`
	`int main()`
	`{`
	`  int x=1;`
	`  int y=2;`
	`  if(x==3)`
	`  {`
	`  y=y-2;`
	`  }`
	`  else`
	`  {`
	`  y=0;`
	`  }`
	`  return 0;`
	`}`
		func begin main
		x = 1
		y = 2
		t0 = x == 3
		if t0 goto L1
		goto L2
		L1 :
		t1 = y - 2
		y = t1
		L2 :
		y = 0
		return 0
		func end
		exit
			PASS

7.	`#include <stdio.h>`
	`int main()`
	`{`
	`  int a=5;`
	`  int b=6;`
	`  if(a<=7)`
	`  {`
	`  if(a==9)`
	`  {`
		func begin main
		a = 5
		b = 6
		t0 = a <= 7
		if t0 goto L1
		goto L2
		L1 :
		t1 = a == 9
		if t1 goto L3
		goto L4
			PASS

	b=b*8;
	b=9;
	}
	else
	{
	a=10;
	}
	}
	else
	{
	b=2;
	}
	return 0;
	}
		L3 :
		t2 = b * 8
		b = t2
		b = 9
		L4 :
		a = 10
		L2 :
		b = 2
		return 0
		func end
		exit	

8.	#include <stdio.h>
	int main()
	{
	int x=5;
	int y=6;
	while(x>7)
	{
	y=y+1;
	}
	return 0;
	}
		func begin main
		x = 5
		y = 6
		t0 = x > 7
		L1 :
		if t0 goto L2
		goto L3
		L2 :
		t1 = y + 1
		y = t1
		goto L1
		L3 :
		return 0
		func end
		exit
			PASS

With Errors

Serial Number	Test Case	Expected Output	Status

1.	void main(){
	int a;
	a = f(2,3);
	}
		Line no: 5 Error message: Variable Not Declared Token: f
		
		Parsing failed.
			PASS

2.	int a(int b, int c){
	return b;
	}
		Line no: 7 Error message: Duplicate Functions Not allowed
		Token:)
			PASS

	`int a(int d){` `  return d;}`	Parsing failed.	

3.	`int a(int b, int c){`
	`  return b;`
	`}`
	
	`void main(){`
	`  int c = a(3.5,2);`
	`}`
		Line no: 8 Error message: Parameter and argument types do not match Token:)
		
		Parsing failed.
			PASS

4.	`int main(){`
	`  return 3.5;`
	`}`
		Line no: 5 Error message: return type does not match function type Token: }
		
		Parsing failed.
			PASS

5.	`void main(){`
	`  continue;`
	`}`
		Line no: 4 Error message: Illegal use of continue Token: ;
		
		Parsing failed.
			PASS

6.	`int main()`
	`{`
	`  int a;`
	`  a = c + 1;`
	`  {`
	`  int b;`
	`  }`
	`  a = b;`
	`  int a;`
	`  int b[10];`
	`  return 1;`
	`}`
		Line no: 8 Error message: Variable Not Declared Token: c
		
		Parsing failed.
			PASS

Functionality and Implementation

The below section describes how the code generates a three address representation for a subset of construct of C language.

1. Variable Declaration

- Declaration statements with initialisations are of the type *typeSpecifier identifier = simpleExpression.* The function gencode() is called after parsing the entire statement. Gencode pushes identifier , assign and simple expression value onto the stack. Thus the stack can now be converted into an instruction of the type as shown in the example below.
  Example -
  int a = 10 + 5; is converted to
  t0 = 10 + 5
  a = t0
- Declaration statements with no initialisation are skipped in intermediate code generation. When the variable is used in an expression it is taken into account.
- Comma declaration can be handled by the same code as for initialisations because ICG does not take any actions for plain declarations. So the semantic rules are only defined for the productions of VarDecInitialise as can be seen in the code above.

2. Function Definitions

- Whenever the user defines a function we denote the start of the function by writing “func begin <name_of_the_function>” and similarly the end is denoted by “func end”.
- The block of statements inside the function are executed as per the rules for each of the statements in the block.
- Example -
  int fsum(int a,int b) {
  int c = a-b;
  int d=a+b;
  return c+d;
  }
  is converted to -
  func begin fsum
  t0 = a - b
  c = t0
  t1 = a + b
  d = t1
  t2 = c + d
  return t2
  func end

3. Call expressions :

- Call expressions encountered during any test case are processed as follows. Whenever a call routine in encountered it changes to call , <no of params> and the params are listed before the function is called. As it can be seen below function f has one parameter which is 6 and then call f,1 to indicate call to f with one parameter which is 6.
- Example -
  int f(int c){ int a =9; int b = a+ 5; return a+b+c;}
  void main(){ f(6);} is converted to
  func begin f
  a = 9
  t0 = a + 5
  b = t0
  t1 = a + b
  t2 = t1 + c
  return t2
  func end
  func begin main
  Param 6
  refparam result
  call f,1
  func end

  In case of any arguments mismatch or function not declared the semantic checker throws an error.

4. While/Do while statements :

- While statement can be divided as set of block statements and the condition. The program uses a label to redirect it to the block of statements on satisfying the condition.
- Example-
  int main() {int i =0,sum = 0; while(i < 10){ sum += i; i += 1;} return 1;}
  This gets converted to
  func begin main
  i = 0
  sum = 0
  t0 = i < 10
  L1 :
  if t0 goto L2
  goto L3
  L2 :
  sum = sum + i
  i = i + 1
  goto L1
  L3 :
  return 1

func end
exit

- Do while statements are similar in nature to while statements.except that the comparison is done in the end.
- Example-
  int main() { int i = 0; do{ i = i + 1;} while (i < 10); return 0;}
  This gets converted to
  func begin main
  i = 0
  L1 :
  t0 = i + 1
  i = t0
  t1 = i < 10
  if t1 goto L1
  goto L2
  L2 :
  return 0
  func end
  exit

5. Expressions :

- Expression evaluation is carried out in terms of three address code. For every operand parsed the stack evaluates the value based on the operators and the operand. The result is pushed onto the stack for further evaluation if necessary. The evaluation is carried out in a left to right manner with precedence rules taken into account by the grammar.
- In case of expressions involving identifiers and array elements the value is fetched from the table and used to evaluation.
- Two stacks are maintained one is called the ICGstack and the other ICGvaluestack. As the name suggests ICGvaluestack saves the value of expression and ICGstack saves the temporary variables.
- For assignment expressions the value is updated back to the table.
- Example -
  void main(){ int a = 6;int b = a + 4; int c; c = a + a + 9;} is converted to
  func begin main
  a = 6
  t0 = a + 4
  b = t0
  t1 = a + a
  t2 = t1 + 9
  c = t2
  func end

6. For statements

- For expressions are iterative statements that execute a block of code if the condition is true. The loop gets converted to a condition involving if statements which either exits the

block or executes them. The increment or decrement statements are written after the comparison condition and then the loop jumps to the condition evaluation after executing block of code. The explanation will be made clearer by the example below.

- Example-
  void main(){ int i=0; for(i=0;i<10;i+=1) int c = c + 1;}
  func begin main
  i = 0
  i = 0
  t0 = i < 10
  L1 :
  if t0 goto L2
  goto L3
  L2 :
  t1 = c + 1
  c = t1
  i = i + 1
  goto L1
  L3 :
  func end
- As seen in the example above , the condition i<10 is stored in t0. If this condition is satisfied the code execution jumps to L2 otherwise L3. To name these labels a labelStack is maintained.

7. If statements

- If statements can be converted to three address codes as described further. The condition is stored in a temporary variable which is used to decide the flow of the programs.
- Example -
  void main(){ int i = 5; if(i>=5){ i*= 4;} else { i /= 4;}}
  func begin main
  i = 5
  t0 = i >= 5
  if t0 goto L1
  goto L2
  L1 :
  i = i * 4
  L2 :
  i = i / 4
  func end
- If the condition i>=5 is true the code execution jumps to label L1 otherwise jumps to L2. The code after the if condition is written before func end. To calculate the labels a labelStack is maintained.

Results and Future Work

Results

We were able to successfully generate three address code for a subset of constructs for C language.The results of the effort are collected here. The input is a simple C program and the output is the three address code and the symbol table.

Input 1:with array

```c
#include <stdio.h>
int main()
{
    int temp=5;
    int arr[100];
    arr[3]=2;
    int k=arr[temp-3];
    return 0;
}
```

Output 1 :

```
-----------------------------------------------

          Intermediate Code Generation

-----------------------------------------------

func begin main
temp = 5
t0 = 4 * 3
t1 = arr[t0]
t1 = 2
t2 = temp - 3
t3 = 4 * t2
t4 = arr[t3]
k = t4
return 0
func end
exit
```

The above screenshot displays the three address code for array access where the access is based on calculations from base address and offset. And offset is in turn based on size of the type of the array.

Input 2: Test Case for functions

```c
#include <stdio.h>
void sum(int a, int b)
{
    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;
}
```

Output 2:

```
------------------------------------------------

         Intermediate Code Generation

------------------------------------------------

func begin sum
c = 0
t0 = a > 7
L1 :
if t0 goto L2
goto L3
L2 :
b = 6
t1 = b >= 5
L4 :
if t1 goto L5
goto L6
L5 :
a = 9
goto L4
L6 :
goto L1
L3 :
func end
func begin main
a = 5
b = 6
t2 = a > 7
L7 :
if t2 goto L8
goto L9
L8 :
b = 6
t3 = b >= 5
L10 :
if t3 goto L11
goto L12
L11 :
a = 9
goto L10
L12 :
goto L7
L9 :
Param a
Param b
refparam result
call sum,2
return 0
func end
exit
```

The above screenshot shows the three address code for function calls and declaration of more than one function in a program.

Input 3: Test case for break statements

```c
#include <stdio.h>
 int main () {
   int a = 10;
   while( a < 20 ) {
      a++;
    if( a > 15) {
        break;
      }
   }
   return 0;
}
```

Output 3:

```
------------------------------------------------

          Intermediate Code Generation

------------------------------------------------

func begin main
a = 10
t0 = a < 20
L1 :
if t0 goto L2
goto L3
L2 :
a = a + 1
t1 = a > 15
if t1 goto L4
goto L5
L4 :
goto L1
goto L3
L5 :
return 0
func end
exit
```

The above screenshot shows the three address code for loop control statements like break and continue.

Input 4: Test case for do while statements

```c
#include <stdio.h>
int main () {
   int a = 10;
   do {
      a = a + 1;
   }while( a < 20 );
    return 0;
}
```

Output 4:

```
------------------------------------------------

         Intermediate Code Generation

------------------------------------------------

func begin main
a = 10
L1 :
t0 = a + 1
a = t0
t1 = a < 20
if t1 goto L1
goto L2
L2 :
return 0
func end
exit
```

The above screenshot above shows the three address code for do..while statements. The control is transferred appropriately using labels.

Input 5: Test case for for statements

```c
#include <stdio.h>
 int main () {
   int a;
   int k;
   for( a = 10; a < 20; a = a + 1 ){
      k = 1;
   }
   return 0;
}
```

Output 5:

```
-------------------------------------------------

          Intermediate Code Generation

-------------------------------------------------

func begin main
a = 10
t0 = a < 20
L1 :
if t0 goto L2
goto L3
L2 :
k = 1
t1 = a + 1
a = t1
goto L1
L3 :
return 0
func end
exit
```

The above screenshot shows the three address code for for statements using appropriate labels.

Input 6: Test case for if..else statements

```c
#include <stdio.h>
int main()
{
    int x=1;
    int y=2;
    if(x==3){
        y=y-2;
    }
    else{
            y=0;
        }
    return 0;
}
```

Output 6 :

```
-------------------------------------------------

          Intermediate Code Generation

-------------------------------------------------

func begin main
x = 1
y = 2
t0 = x == 3
if t0 goto L1
goto L2
L1 :
t1 = y - 2
y = t1
L2 :
y = 0
return 0
func end
exit
```

The above screenshot shows the three address codes for if else statements.

Input 7: Test case for nested if..else statements

```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;
}
```

Output 7:

```
-----------------------------------------------

        Intermediate Code Generation

-----------------------------------------------

func begin main
a = 5
b = 6
t0 = a <= 7
if t0 goto L1
goto L2
L1 :
t1 = a == 9
if t1 goto L3
goto L4
L3 :
t2 = b * 8
b = t2
b = 9
L4 :
a = 10
L2 :
b = 2
return 0
func end
exit
```

Above screenshot shows the three address code representation for nested if..else statements.

Input 8: Test case for while statements

```c
#include <stdio.h>
int main()
{
    int x=5;
    int y=6;
    while(x>7){
        y=y+1;
    }
    return 0;
}
```

Output 8:

```
-----------------------------------------------

          Intermediate Code Generation

-----------------------------------------------

func begin main
x = 5
y = 6
t0 = x > 7
L1 :
if t0 goto L2
goto L3
L2 :
t1 = y + 1
y = t1
goto L1
L3 :
return 0
func end
exit
```

The above screenshot shows three address code for while statements.

Future work

The yacc script we have used for the intermediate code generation implements a subset of the C language, further work can be done to include more features, we had to implement a small subset because of time constraints.

More work can be done on lex script and yacc script to work with type casting. Further work under this project includes the implementation of code optimiser and further phases.

References

1. https://www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/Ccfg.html
2. http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf
3. http://dinosaur.compilertools.net/yacc/